Projecting Computer Languages for a Protean Interaction

Projeter les Langages Informatiques pour une Interaction Protéiforme

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 :
Sciences et technologies de l'information et de la communication (STIC)

Spécialité de doctorat : Informatique

Graduate School : Informatique et sciences du numérique

Référent : Faculté des Sciences d'Orsay

Thèse préparée dans l'unité de recherche **LISN (Université Paris-Saclay, CNRS)**,
sous la direction de **Michel BEAUDOUIN-LAFON**, Professeur à l'Université Paris-Saclay.

Thèse soutenue à Paris-Saclay, le 18 mars 2024, par

Camille GOBERT

Composition du jury

Membres du jury avec voix délibérative

Anastasia BEZERIANOS Professeure, Université Paris-Saclay	Présidente
Amy KO Professeure, University of Washington	Rapporteure & Examinatrice
Stéphane CONVERSY Professeur, ENAC-LII & Université Fédérale de Toulouse	Rapporteur & Examinateur
Alan DIX Professeur, Swansea University	Examinateur
Tomas PETRICEK Maître de conférence, Charles University	Examinateur

Titre : Projeter les Langages Informatiques pour une Interaction Protéiforme

Mots clés : Interaction, Langage informatique, Projection, Programmation, *i*-LaTeX, Lorgnette

Résumé : Interagir avec les ordinateurs nous amène souvent à utiliser des langages informatiques tels que Python pour créer des programmes et LaTeX pour rédiger des documents techniques. Au cours des dernières décennies, ces langages se sont mis à occuper une place de plus en plus importante dans des domaines divers allant des sciences aux arts. Ils sont désormais enseignés aux élèves du monde entier et deviennent une compétence couramment attendue sur le marché du travail, creusant le fossé entre les citoyens qui les parlent et les autres. Or, pour l'essentiel, notre interaction avec ces langages est identique à ce qu'elle était il y a cinquante ans : lire et écrire du texte brut. Bien que diverses alternatives aient été proposées, elles ont souvent été développées avec une approche moderniste, en rupture avec les langages et pratiques répandus, résultant en des systèmes bornés aux cercles académiques et à quelques communautés spécifiques. Par conséquent, les langages informatiques se retrouvent entravés par des visions qui essentialisent notre interaction avec eux, encourageant ainsi à s'en distancier, au risque de perdre une partie de la maîtrise à laquelle ils nous donnent accès face à des interfaces toujours plus « simples » et à la synthèse « intelligente » de code. Dans cette thèse, je m'oppose à ce point de vue, que je considère néfaste pour la recherche en interaction humain-machine et la démocratie à l'heure du tout numérique. À cette fin, je développe un ensemble d'arguments qui soulignent qu'il est possible — et même bénéfique — de rendre plurielle notre interaction avec les langages informatiques, y compris avec les langages déjà existants, en phase avec l'héritage et la diversité inhérents à l'informatique du XXIe siècle. Je développe d'abord une nouvelle théorie de l'interaction avec les langages informatiques qui montre qu'aucun de ces langages n'est intrinsèquement lié à une représentation ou à un type d'interaction spécifique. Pour cela, je déconstruis la notion de *langage informatique* en cinq aspects fondamentaux afin d'isoler l'interaction des autres éléments constitutifs de ces langages, formant ainsi un modèle de langage informatique plus holistique que ceux déjà existants. J'identifie alors quatre niveaux d'interaction avec ces langages et montre que l'on peut les hybrider et choisir de *projeter* un même fragment de code sur différentes représentations afin de laisser le choix de celle offrant l'interaction la plus adaptée aux utilisateurs finaux. J'applique ensuite cette vision des langages informatiques à deux problèmes de recherche à l'aide d'une méthodologie de conception centrée sur l'utilisateur : faciliter la conception de documents avec le langage LaTeX, et aider les programmeurs à créer leurs propres projections afin de mieux s'approprier leurs éditeurs de texte. Dans le cadre de chaque problème, j'identifie les problèmes et limites rencontrés par les utilisateurs des langages concernés à l'aide d'une étude formative, développe un prototype d'éditeur de texte augmenté de projections supplémentaires, et évalue celui-ci à l'aide d'études utilisateur à la fois qualitatives et quantitatives. Les résultats montrent que le fait de compléter le texte par d'autres représentations nous aide à comprendre et à modifier le code plus rapidement et avec une charge de travail moindre, et que ces représentations peuvent être crées à moindre coût en recomposant des éléments réutilisables d'une projection à une autre. En conclusion, je montre qu'envisager l'interaction avec les langages informatiques sous la forme de projections permet de rendre celle-ci plus protéiforme, une approche qui, selon cette thèse, est théoriquement fondée, techniquement possible et empiriquement souhaitable. Cette approche se prête à la tâche urgente d'équiper un public citoyen toujours plus large avec de nouveaux outils intellectuels et techniques afin de les aider à comprendre et à s'approprier les langages informatiques au cœur du fonctionnement de nos sociétés.

Title : Projecting Computer Languages for a Protean Interaction

Keywords : Interaction, Computer language, Projection, Programming, *i*-LaTeX, Lorgnette

Abstract : To interact with computers, we often rely on computer languages such as Python for creating programs and LaTeX for writing technical documents. In the past few decades, these languages have become increasingly used in a variety of fields ranging from science to arts. They are now being taught to millions of pupils and have become a staple skill in the job market, widening the gap between those who are computer literates and the others. However, for the most part, our interaction with these languages has remained similar to what it used to be fifty years ago : reading and writing code as plain text. Although various alternatives have been introduced, they have often been developed with a modernist approach, in isolation from popular languages and widespread workflows, resulting in systems that hardly cross the borders of academic circles and niche communities. As a result, our interaction with computer languages is now hampered by essentialist views that encourage us to move away from them, at the risk of losing some of the agency they give us, as if dealing with code was nothing but a burden from the past compared to ever more "simple" user interfaces and "intelligent" code synthesis. In this thesis, I argue against this view that I consider harmful to human-computer interaction research and computer-driven democracies. To do so, I introduce a number of arguments that show that it is indeed possible—and even beneficial—to make our interaction with computer languages more diverse, including with the many languages that already exist that are inherent to the diversity of computing in the 21st century. I first develop a new theory of interaction with computer languages that shows that no such language is inherently bound to a specific representation or type of interaction. To that end, I deconstruct the notion of *computer language* into five fundamental aspects to isolate interaction from the other constituents of these languages, yielding a more holistic model than those that already exist. I then use this model to identify four different levels of interaction with computer languages, which can be hybridised, and show that a single piece of code can very well be *projected* onto several representations to let end-users decide which representation supports the form of interaction most appropriate for them. I then apply this view of computer languages to two research problems using user-centred design methodologies : helping users author documents written in LaTeX and helping programmers appropriate their text editors by crafting their own projections. For each problem, I assess the limitations of existing solutions with the help of a formative study; I develop a prototype of a text editor equipped with additional projections; and I evaluate it with both qualitative and quantitative user studies. The results show that that complementing text with other representations helps us understand and modify code faster and with a lower workload and that these representations can be created by recomposing existing parts that can be reused from one projection to another. In conclusion, I show that considering interaction with computer languages as projections makes it more protean, an approach which, according to this thesis, is theoretically grounded, technically possible and empirically desirable. It lends itself to the urgent task of equipping an ever-growing public of citizens with new intellectual and technical tools to help them understand and appropriate the computer languages that rule the societies we live in.

This document was written in LaTeX and generated in April 2024 using pdfTeX and BibLaTeX. It uses a custom template created for the occasion by the author of this work. It is typeset with the Minion Pro and IBM Plex Mono typefaces.

Résumé

La notion de *langage informatique* est au cœur de notre interaction avec les ordinateurs. Par exemple, les langages Python et JavaScript font parti des langages les plus utilisés pour concevoir des programmes informatiques, tandis que les langages LaTeX et Markdown sont massivement employés afin de rédiger des documents techniques. Au cours des dernières décennies, ces langages se sont mis à occuper une place de plus en plus importante dans des domaines aussi divers que la recherche scientifique et la création artistique. Ils sont désormais enseignés aux élèves du monde entier et deviennent une compétence couramment attendue sur le marché du travail, creusant le fossé entre les citoyens qui les parlent et les autres.

Or, pour l'essentiel, notre interaction avec ces langages est identique à ce qu'elle était il y a cinquante ans : lire et écrire du texte brut. Bien que diverses alternatives aient été proposées, elles ont souvent été développées en rupture avec les langages et pratiques répandus, résultant en des systèmes bornés aux cercles académiques et à quelques communautés spécifiques. Par conséquent, les langages informatiques se retrouvent désormais entravés par des visions qui essentialisent notre interaction avec eux, encourageant ainsi à s'en distancier, au risque de perdre une partie de la maîtrise à laquelle ils nous donnent accès face à des interfaces toujours plus « simples » et à la synthèse « intelligente » de code.

Dans cette thèse, je m'oppose à cette vision des langages informatiques, que je considère néfaste pour la recherche en interaction humain-machine et la démocratie à l'heure du tout numérique. À cette fin, je développe un ensemble d'arguments qui soulignent qu'il est possible — et même bénéfique — de rendre plurielle notre interaction avec les langages informatiques, y compris avec les langages déjà existants, en phase avec l'héritage et la diversité inhérents à l'informatique du XXIe siècle. Ces arguments, théoriques puis empiriques, s'articulent en deux parties distinctes.

Dans la première partie de cette thèse, je développe une nouvelle théorie de l'interaction avec les langages informatiques qui montre qu'aucun de ces langages n'est intrinsèquement lié à une représentation ou à un type d'interaction spécifique. Pour cela, je déconstruis la notion de langage informatique en cinq aspects fondamentaux — concepts, spécification, implémentation, interaction et contexte —, isolant ainsi la façon dont on interagit avec un langage des autres éléments qui le caractérisent. Cette décomposition en cinq aspects

me conduit à introduire un nouveau modèle des langages informatiques, plus holistique que ceux déjà présents dans la littérature, dont je déduis deux contributions théoriques.

Premièrement, je postule que le rôle de l'aspect interactif d'un langage informatique est de nous donner accès aux quatre autres aspects du langage, formant ainsi une nouvelle taxonomie de l'interaction avec un langage dotée de quatre niveaux : l'interaction graphémique, l'interaction morphosyntaxique, l'interaction sémantique et l'interaction pragmatique. Je montre également que les systèmes interactifs hybrident couramment plusieurs de ces niveaux, que j'illustre avec une panoplie d'exemples mélangeant interfaces classiques et techniques issues de l'état de l'art.

Deuxièmement, les données encodant le langage dans la mémoire d'un ordinateur nous étant inaccessibles telles quelles, je propose d'utiliser le concept de *projection* pour qualifier le phénomène par lequel le code se retrouve *projeté* sur le substrat avec lequel nous interagissons réellement. Afin de précisément caractériser la notion de projection, j'introduis un glossaire de concepts et un ensemble de sept propriétés. Celles-ci définissent un espace de conception de projections pouvant aussi bien être utilisé pour analyser les projections de systèmes existants que pour explorer de nouvelles opportunités de manière systématique. La projection de code informatique n'étant pas limitée à un unique substrat, c'est-à-dire à une approche *uniforme* de l'interaction, je conclus qu'il n'existe pas de barrière théorique à une approche *protéiforme*, dans laquelle un même fragment de code se retrouve projeté sur différents substrats, laissant l'utilisateur libre de choisir parmi l'ensemble des projections du code sur lequel il travaille, à la façon d'une panoplie d'outils mis à la disposition de sa curiosité et de son expertise.

Dans la seconde partie de cette thèse, j'applique cette approche protéiforme de l'interaction avec les langages informatiques à deux problèmes de recherche concrets à l'aide d'une méthodologie de conception centrée sur l'utilisateur. Je me concentre plus spécifiquement sur le fait de complémenter une projection textuelle de l'ensemble du code avec une ou plusieurs projections alternatives des fragments de texte les plus à même d'en bénéficier. Ainsi, dans le cadre de chaque problème de recherche, j'identifie les problèmes et limites rencontrés par les utilisateurs des langages concernés à l'aide d'une étude formative ; je développe un prototype d'éditeur de texte doté de projections supplémentaires ; et j'évalue celui-ci à l'aide d'études utilisateur qualitatives et quantitatives et de mises en situation des prototypes.

Je m'intéresse d'abord au problème de la conception de document avec le langage LaTeX. Afin d'identifier les difficultés auxquelles font face les utilisateurs du langage, je présente les résultats d'une analyse thématique de 11 interviews d'utilisateurs de LaTeX. Je présente alors *i*-LaTeX, un prototype d'éditeur LaTeX dans lequel il est possible d'interagir avec certains fragments du langage identifiés dans l'analyse thématique à l'aide de représentations

alternatives du code que l'utilisateur peut afficher à l'intérieur du document généré, un type de projection appelé *représentation transitionnelle*. À l'aide d'une évaluation contrôlée et d'une étude longitudinale, je montre que ces représentations permettent aux utilisateurs de concevoir leurs documents en adoptant des stratégies faisant intervenir plusieurs projections, leur permettant ainsi d'effectuer certaines tâches courantes plus rapidement et avec une charge de travail moindre.

Je m'intéresse ensuite à la question de l'appropriation des éditeurs de texte par leurs utilisateurs. Afin de mieux comprendre de quelle manière les programmeurs souhaiteraient pouvoir interagir avec le code qu'ils lisent et écrivent sous forme de texte, j'analyse un ensemble de 62 projections collectées via un atelier de conception participative organisé avec 9 programmeurs et une revue de systèmes publiés dans la littérature. Je présente ensuite LORGNETTE, un système permettant d'instrumenter un éditeur de texte existant afin de permettre à ses utilisateurs d'y ajouter de nouvelles projections en associant une interface graphique de leur choix à un motif arbitraire dans le code de manière libre et idiosyncratique. J'illustre la capacité de LORGNETTE à enrichir l'interaction avec plusieurs langages existants en l'utilisant pour créer des projections complémentaires au texte dans cinq situations différentes, telles qu'une une grille interactive permettant de manipuler la structure d'un tableau en Markdown et un formulaire permettant de configurer des propriétés graphiques en Python et en JSON.

En conclusion, ce travail suggère qu'il est possible de réenvisager notre interaction avec les langages informatiques à travers le concept de *projection protéiforme*, une approche qui, selon cette thèse, est théoriquement fondée, techniquement possible et empiriquement souhaitable. L'approche présentée ici se veut résolument postmoderne. Elle vise à s'accommoder aux systèmes et aux encodages existants, malgré leurs éventuels défauts, en tant que ceux-ci constituent le matériau réel et durable avec lequel nous sommes contraints d'interagir et face auxquels les approches modernistes nécessitant de faire table rase du passé se heurtent. Elle se prête ainsi à la tâche urgente d'équiper un public citoyen toujours plus large avec de nouveaux outils intellectuels et techniques afin de les aider à comprendre et à s'approprier les langages informatiques au cœur du fonctionnement de nos sociétés.

Acknowledgments

I would never have studied computer science, let alone written a Ph.D. thesis, if I had not met all the people who shaped the journey that led me to where I am today. As this journey started a long time ago, soon after I started using the first computer I owned, accounting for all those who played a significant role in it requires a little history.

My interest in computer science began in early middle school when I discovered programming as I spent countless hours taking my first steps into the worlds of web and video game development—a freedom my parents gave me that I will eternally be grateful for. I will never forget the joy I felt the first time I wrote a webpage using HTML and CSS or watched my computer successfully interpret a piece of PHP or Ruby code I tweaked out of curiosity at a time when most of these languages' concepts were unknown to me. Yet, despite entering high school with a keen interest in mathematics and science, I had already decided that although science was fun, my goal was to study and work in graphic and interaction design. What a delusion when, in the Summer of 2013, I discovered that each of my applications to two preparatory schools in arts—*Mises À Niveau en Arts Appliqués* in French—I applied to had been rejected. My only other plan, back then, came from a clever recommendation from my high school teachers: always apply to a bachelor you might be interested in at the university, *just in case*. This is how, a decade ago, I enrolled in the first year of a B.Sc. in computer science at Aix-Marseille University, located in the South of France, where I come from.

I started learning computer science without any expectation, initially thinking I would only spend one year at the university before applying to the same preparatory schools again. I could never have been so wrong. Not only did I immediately like the topics I discovered there, but I also quickly found out about the peculiar relationship to knowledge there is in academia, which is something that I never ceased to admire since then. For this reason, I am thankful to all the professors and researchers I met during these years, whose passion for sharing their knowledge rather than (just) helping students get good grades convinced me that I should stay a little longer. Two of them, in particular, played an essential role in helping me get to where I am today. The first one is Régis Barbanchon, who was in charge of the C programming class I attended in Spring 2014. Régis was the first person I met who insisted so much on the importance of naming conventions and code organisation.

He sparked my interest in the role of design in programming, which slowly matured in me to become the topic of the thesis you are reading. The second one is Frédéric Béchet, who was in charge of a web programming class I attended in Spring 2015. By trusting me enough to offer me an internship opportunity in his research team in the middle of my B.Sc., during which I designed and developed the user interface of a semi-automated annotation tool for textual corpora (using a gigantic amount of inefficient jQuery code!), Frédéric made me discover the daily life of an academic research lab for the first time, before I even considered doing research.

As the end of my B.Sc. approached, I decided I did not want to return to my initial plan, at least not immediately. At the time, the most obvious option for me was to pursue a two-year M.Sc. program before working my way to interaction design. Yet, my discovery of the field of human-computer interaction (HCI) during an exchange semester at Uppsala University in Sweden and the opportunity I had to join the École normale supérieure of Paris (ENS) the year after made me stay around academia for four more years instead. The École normale supérieure is a small world where intellect meets craziness, and I can confidently say that everything I shared with all the students I met and all the friends I made there, from highly intellectual discussions to highly absurd pranks, paved the way to my decision to stay a little longer in this ecosystem and pursue a Ph.D. I am thankful for the trust I received from my academic advisor there, Timothy Bourke, who always supported my interest in HCI research (which is rather uncommon for an ENS student), as well as from the supervisors of the three research internships I did back then—Géry Casiez in 2017, Antti Oulasvirta and Kashyap Todi in 2018, and Michel Beaudouin-Lafon in 2020—and all the fellow researchers I met along the way, who helped me discover and explore multiple aspects of the research field I eventually settled in, which is, to some extent, the academic flavour of the job I had in mind back in high school.

At the end of 2019, I contacted Michel Beaudouin-Lafon to express my interest in doing a Ph.D. under his supervision. Michel was very kind and receptive to my initial ideas and helped me brainstorm and write the research subject this thesis is about. With his help, my Ph.D. was soon funded for three years, and the final part of my ten-year journey in higher education began in October 2020. Although doing a Ph.D. can be a little tedious and make you feel a little lonely at times, I really, really enjoyed the four years I spent in the ex)situ group of the LISN lab, surrounded by many great people all along the way.

Thank you, Michel, for giving me so much freedom in my work, yet being so responsive every time I need your help, without ever pressuring me or steering my research towards a direction I was not interested in. Thank you, Wendy Mackay, for welcoming me to your research group and for all the knowledgeable recommendations you are an expert in giving. Thank you,

Janin Koch, Fanis Tsandilas, Sarah Fdili Alaoui and Nicolas Taffin, for all our great discussions on both academic and non-academic matters. Thank you, all my fellow students, and in particular Téo Sanchez, Elizabeth Walton, Miguel Renom, Han Han, Martin Tricaud, Alexandre Battut, Arthur Fages, Junhang Yu, Capucine Nghiem, Romane Dubus, Alexandre Kabil and Yann Trividic, for all the reliable help, the lengthy chats and the beers we shared in Paris and abroad. I redouble my thanks to you, Martin, with whom I shared an office for a while, for the hours you spent explaining, exploring and debating ideas with me; for the many papers I could not have discovered had you not told me about them; and for enduring my countless jokes on your crazy love for category theory. May you succeed in improving HCI with monads and adjunctions! In the same spirit, thank you, Adrien Chaffangeon Caillet, Joanna McGrenere, Thomas Baudel, Simon Talaga, Edwige Chauvergne and Matthew Beaudouin-Lafon, for your helpful thoughts and references. Thank you, participants to my user studies, for helping science and myself move forward; and thank you, internet strangers from StackOverflow and elsewhere, for answering the many questions I kept facing.

Since doing a Ph.D. is not only about research, I would like to highlight a few more people I had the pleasure of collaborating with. Thank you, Caroline Appert and Ravi Chugh, for kindly examining my doctoral work at the first half of the journey. Thanks to the support staff of the lab, who help us all do our job, and more specifically to Sonia Vanot, for your smiley help with mission planning and administrative issues. Thank you, James Eagan and Michel, for trusting me to teach your courses—and cheers to my fellow teaching assistants, Mehdi Chakhchoukh, Raphaël James and Yiran Zhang. Thank you, Camille Challant and the rest of the JDD 22 team, for the fun times we had organising the Ph.D. days of the lab in 2022.

Finally, since life is—fortunately!—not only about doing a Ph.D., I would like to extend all my gratefulness to the people who played a significant role in the rest of my life in the past few years. Thanks to all my friends, and in particular to Anatole D., Anna M., Lucas P., Mathieu F., Benoit S., Luc S., Étienne R., Clément R., Maël R., Gaspard M. and Énora le G., for being consistently here during this journey, for everything that our friendships have brought me, and for the mind-opening discussions about our respective fields of expertise and points of view. Thank you, Leila S. and Tiphaine L., for your peculiarity, love and care. Had you not introduced me to Pink Floyd and Colette, these long days (and nights) of work would not have been the same! Thanks to all my relatives who kindly supported me during this journey despite my limited presence. In particular, thanks to my parents, Anne-France B. and Claude G., for giving me so much love and autonomy, fueling my interest in design and never failing to support me in doing what makes me happy. I still hardly realise the chance I have of knowing you all, and I cannot express how much I love you, all and forever.

Contents

Figures

Tables

To my grandmother Marie,
tel un ultime bulletin de notes.

1

Introduction

It's a beautiful thing, the destruction of words.

— *Nineteen Eighty-Four* (Orwell, 1949, ch. 5)

In *Nineteen Eighty-Four*, a famous dystopian novel written by George Orwell published in 1949, England has become a totalitarian state where freedom and language are both under the government's control. The protagonist, Winston Smith, works at the *Ministry of Truth*, where he constantly rewrites historical documents to ensure they always align with the ever-changing position of the unique political party. There, he meets Syme, one of his colleagues working in a team of linguists tasked to write the official dictionary of Newspeak—the new official language of the state—, who eagerly explains to Winston the purpose of his work: "*You think, I dare say, that our chief job is inventing new words. But not a bit of it! We're destroying words—scores of them, hundreds of them, every day. We're cutting the language down to the bone*". Surprised by Winston's lack of enthusiasm, he goes on, eventually concluding—with a great deal of admiration for the idea—that "*in the end we shall make thoughtcrime literally impossible, because there will be no words in which to express it*".

Fortunately, we avoided the terrible fate depicted in Orwell's novel. Perhaps oppositely, the rise of computing gave birth to many, many new languages instead. Although computers were already in use when Orwell published *Nineteen Eighty-Four*, computing was merely at its infancy at the time. Yet, creating languages soon became necessary as computers quickly became more powerful and widespread, for people working on increasingly complex programs and data needed a practical support of thought and means to communicate with the machine, beyond writing long sequences of low level operations and binary digits. In little time, Fortran, Lisp, BASIC and many others were developed, becoming a core part of computer education, use and research. Computer languages were born and on the rise.

The way we interact with computers drastically evolved since the time of the first computer languages. Early work on this topic was led by visionaries such as Douglas Engelbart, whose work on *augmenting the human intellect* (Engelbart, 1962) with the help of computers pioneered a number of interactive systems such as hypertext, video conferencing and shared editing. Step by step, the field of human-computer interaction (HCI) gradually demonstrated that interacting with computers could go beyond typing and reading monospaced text. In the 1970s, computers soon became equipped with bitmap displays and mice, leading to the development of operating systems capable of displaying increasingly detailed graphical user interfaces in which users could point at the target they wish to interact with. This, in turn, gave birth to a myriad of graphical programs, such as word processors, spreadsheets, drawing applications and web browsers, some of which still exist today.[1] The development of interaction techniques at odds with established textual interaction, such as direct manipulation, intensified the schism between interacting with computers through languages and interacting with computers through metaphorical GUIs, best exemplified by the title of Shneiderman's classic article on direct manipulation interfaces: *Direct Manipulation: A Step Beyond Programming Languages* (Shneiderman, 1983).

1. For example, office software such as Microsoft Word and Microsoft Excel and image editing software such as Adobe Photoshop were first released in the 1980s and have both remained some of the most popular options in their respective categories since that time, with relatively little change to their user interfaces.

Nowadays, working with computer languages is often depicted as a technical necessity at best, and a burden from the past at worst. Many interactive systems hide underlying linguistic considerations from their users, who are rather encouraged to delegate their agency to *cloud services* and *artificial intelligence*, regardless of what these terms actually mean and what impact they have on their users and on the world—economically, ecologically and ethically. Eventually, a sensible fraction of the world's population might be living in societies in which computers have an all-time high importance in everybody's daily lives, even though these computers are mute strangers to many citizen, who must abide by their mechanised decision processes. I believe that this may eventually put our democracies at risk, when citizen do not or cannot speak the languages of the machines ruling their lives—a risk not so hypothetical given the current impact of our lack of understanding of contemporary artificial intelligence models already deployed in the wild, including when they decide who should be arrested or who should be killed (Benbouzid, 2019; Johnson, 2020).

While education and legal regulation may help counterbalance this trend, as hinted by the GDPR[2] in Europe and current work on regulating artificial intelligence in various regions of the world,[3] I believe that making computer languages more accessible and understood is a crucial step in giving citizen empowering tools to think and act for themselves in our computer-centric societies. Unfortunately, despite some small improvements, our interaction with many of the most popular and useful computer languages in use today is extremely similar to what it used to be in the 1970s—that is, editing monospaced text. I believe this contributes to making computer languages look like tools for experts and discourages people from learning and using them, even though there is no reason these languages could not be used in autonomy by everyone, as suggested by the remarkable success of programming classes and systems for children, who seem to be at ease with using computer languages from a young age without any formal training in STEM.

2. GDPR stands for *General Data Protection Regulation*, a major component of the European Union's legal texts on privacy and digital human rights that is in effect since 2018.
3. Recent examples include President Joe Biden's executive order on AI regulation issued in October 2023 in the US (🔗1) and the finalisation of the AI act in the European Union in December 2023 (🔗2).

To contribute in addressing this challenge rooted in HCI, I decided to direct the research work I conducted towards reconciling computer languages and human-computer interaction. As often in research, though, this line of work is neither new nor unique. In this case, other researchers have previously called for further research in programmer experience (PX), a subfield of user experience (UX) research focusing on programming system users (Morales et al., 2019), and initiated joint work on unifying research on programming languages and human-computer interaction (Chasins et al., 2021). Moreover, mixed communities have formed to explore this intersection, as illustrated by the Future of Coding (@3) and the LIVE and PAINT workshops at the ACM SPLASH conference (@4; @5).

In the context of my Ph.D., I specifically worked on complementing—rather than replacing—textual representations of computer languages with other representations of the code. The resulting work led to one publication in a national HCI conference (Gobert and Beaudouin-Lafon, 2021) and two publications in international HCI conferences (Gobert and Beaudouin-Lafon, 2022, 2023), which were all co-authored with my Ph.D. supervisor, Michel Beaudouin-Lafon. Informed by the outcomes of this work, I hereby make the following claim:

> *No computer language is inherently bound to a single representation, neither theoretically nor technically, and diversifying our interaction with computer code can help users understand and modify it.*

To support this claim, I present theoretical, technical and empirical contributions to the research field presented above, which are roughly divided into two parts. The first part, ranging from chapter 2 to 5, corresponds to the main theoretical contribution of this work. By first deconstructing, then reconstructing, the very notion of *computer language*, these chapters gradually describe a novel theory of interaction with computer languages. This theory decouples interaction from other aspects that constitute computer languages with the help of the concept of *projection*, which I progressively introduce to the reader by explaining *why* we need a new theory and a more holistic model of computer languages (chapters 2 and 3), *what* are the different ways we can interact with a computer language according to this theory (chapter 4), and *how* can we implement them in both novel and established code editing environments (chapter 5). The second part, ranging from chapters 6 to 7, presents two independent projects that I carried out that correspond to the main technical and empirical contributions of this work. By applying the aforementioned theory to two concrete research questions—how to improve our interaction with the LaTeX language, and how to help users tailor their text editors with custom representations of the code—using a user-centred design methodology, these two chapters demonstrate practical applications of the theory for rethinking the way we interact with computer languages, including existing ones. More specifically, the remaining eight chapters of this thesis are structured as follows.

In chapter 2, I start by defining what a computer language is, compare it with related terms, and report on various situations they have been used in, underlining how widespread they are and how diverse their users can be. I then present how computer languages have been studied in different fields of research, both human-centric and computer-centric, as well as in HCI, and point at several limitations of the theories HCI researchers have at their disposal to work on this topic.

In chapter 3, I propose to start filling this theoretical gap by decomposing the notion of computer language into five fundamental aspects that, I argue, must be part of every computer language: conceptualisation, specification, implementation, interaction and contextualisation. Put together, these aspects form a new holistic model of computer languages.

In chapter 4, I derive a new taxonomy of interaction with computer languages featuring four complementary levels: graphemic interaction, morphosyntactic interaction, semantic interaction and pragmatic interaction. I then show that these levels can be used as a new taxonomy for categorising both classic and state-of-the-art techniques for interacting with computer languages, and that they can be combined within a single code editing systems.

In chapter 5, I drill down on the notion of *projection*, a core concept for making interaction with computer languages possible according to this theory. I define the concept and its multiple properties, which form a set of dimensions that can be used to characterise existing systems for interacting with computer languages and explore new design opportunities in a systematic fashion. I further report on several milestones in historical implementations of this concept, leading me to position the strategy I used in my own work, which was motivated by postmodern visions of computing.

In chapter 6, I apply the theory introduced in the last three chapters by diversifying representations of LaTeX code to address some of the difficulties that I identified by interviewing 11 users of the language. I present *i*-LaTeX, a LaTeX editor with alternative representations of specific pieces of code that I developed, and report on evaluations of *i*-LaTeX with a controlled experiment and a longitudinal study.

In chapter 7, I reflect upon limitations of my work on LaTeX and switch to the problem of making alternative representations of text more appropriable by their users. I present LORGNETTE, a framework for augmenting text editors with alternative representations of the code that can be modified and created by end-users without requiring them to reprogram the entire editor. I demonstrate the value of LORGNETTE by using it to create visual tools to work with several computer languages in five different situations.

In chapter 8, I review the relations between the two parts of my work and discuss the benefits and limitations of the theory of interaction with computer languages proposed in this thesis, which I relate to the successes and failures I faced in developing and evaluating *i*-LaTeX and LORGNETTE. I then propose several directions to address them in the future, forming a possible research agenda to pursue the line of work presented in this thesis.

Finally, in chapter 9, I go back to the statement I made above and highlight how the contributions presented in this thesis support it. I conclude that computer languages, old and new, can all benefit from a mixture of representations and interaction techniques that complement each other, independently from how they are encoded, and present my vision on how this might affect how we use computers and their languages in education, industry and society.

2 Framing computer languages

At its core, this thesis is concerned with the study of our interaction with computer languages—but what is a computer language? How do they differ from other sorts of languages, such as natural languages? Who are the users of these languages; why do they use them; and why can it be challenging? What is more, what approaches and concepts have been used to study computer languages so far, and what can we learn from them to study these languages from a human-computer interaction perspective?

This chapter addresses these fundamental questions. Section 2.1 explains what computer languages are. It defines the term as it is used in this thesis, compares it to similar notions used in the literature, and gives a number of examples of computer languages. Section 2.2 further helps framing what these languages are by giving examples of what computer languages have been and are being used for in five major application domains. Doing so also highlights the diversity of computer language users, who range from scientists to artists, therefore giving an overview of user groups one may focus on when working on this topic. Finally, section 2.3 presents several human-centred and computer-centred perspectives from which computer languages have been studied in the past. It then argues that human-computer interaction lacks a theory of computer languages that captures the diversity of ways we can interact with these languages, which neither of the aforementioned theories seem appropriate for by itself, therefore leaving room for research on creating a dedicated theory, which I pursue in the following chapters.

2.1 WHAT IS A COMPUTER LANGUAGE?

There is no understood definition of what a computer language is. Depending on the context it is used in, and the person who uses it, the term may very well refer to different concepts. Some may argue that a computer language is just a synonym for a programming language, even though this term has no single definition either, as discussed by Ko (2016). Others may refer to theoretical considerations, such as having a formal grammar or being Turing-complete, or rather judge what is a computer language by the look of its notation.

To better delimit the object of study of this thesis, I must therefore start by defining what a computer language means in this work. This is the purpose of this first section, which combines an intentional approach with an extensional approach. First, I propose my own definition of computer language by characterising the key properties of this concept. Then, I contrast this definition by comparing the term with several others terms, some of which are more commonly used, in order to determine how they overlap with *computer language* and strengthen the definition with positive and negative examples.

2.1.1 DEFINITION

Language

A computer language, as its name suggest, is a particular kind of language. Before attempting to define the former, I must therefore start by defining the latter. In the context of this thesis, I define a language as a symbolic system for communicating information that has two constituents: (1) *symbols*, which are atoms of information that can be physically reified as, e.g., shapes, sounds or gestures; and (2) *rules*, which specify how symbols can be combined and how these combinations can be interpreted. This definition is in line with the main definitions of *language* found in several English dictionaries:

> *A systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures, or marks having understood meanings.*
>
> — Merriam-Webster dictionnary (2023)

> *A system of communication consisting of sounds, words, and grammar.*
>
> — Cambridge dictionnary (2023)

> *A system for the expression of thoughts, feelings, etc, by the use of spoken sounds or conventional symbols.*
>
> — Collins dictionnary (2023)

Computer language

Computer languages are languages which have a singular relationship with computers. Some other definitions of language found in dictionaries may hint at ways to qualify that relationship, such as by restraining who uses computer languages, or what they are used for:

> *The language of a particular nation or people.*
>
> — Collins dictionnary, 2023

> *A system of symbols and rules for writing instructions for computers.*
>
> — Cambridge dictionnary, 2023

However, I argue that neither of these definitions alone is satisfactory. Defining computer languages by their users is either too limiting, as not only computer scientists or software developers use them, or too vague, as more and more people are now becoming computer language literates. Defining computer languages by their purposes is fragile in time, as it is difficult to know in advance how a particular language will be used, and history has repeatedly shown that computer languages are sometimes used in unexpected ways, as we shall see in the subsequent sections of this chapter.

Instead, I argue that the defining characteristic of a computer language is that it is a symbolic systems meant to be understood and manipulated both by humans and computers:

> *A computer language is a language designed to be used by humans and interpreted by computers in the appropriate context.*

In addition, to avoid using purpose-specific terms such as *program*, I use the term *code* to refer to anything written in a computer language, i.e., anything that translates to a combination of the language's symbols. This definition of computer language has a number of important implications.

First, since a computer language is a particular kind of "*language*", it must include symbols and rules that can be materialised physically. This excludes symbols that only exist in one's mind, with no physical support, as well as collections of symbols not accompanied by rules specifying which combination of symbols is part of the language and which is not, and how to assign them some meaning.

Then, since a computer language is "*designed to be used by humans*", it must exist in a form we can indeed use as humans, relevant to our perceptual and cognitive abilities, which highly differs from those of machines. As a consequence, the definition excludes any language that is only used internally by a computer or a network of computers, without any human intervention, such as a number of communication protocols.

Finally, since a computer language must also be designed to be "*interpreted by computers in the appropriate context*", it must be possible to create a computer program that "speaks" the language by reading and writing code written in that language in a purposeful way. This notably excludes natural languages from this definition, as they were not *designed* to be interpreted by computers, even though we can write computer programs able to synthesise plausible sentences and extract information from such languages using natural language processing techniques. The precision regarding the "*appropriate context*" means that a computer language does *not* have to be universally understandable by *any* computer or computer program to be considered a computer language. If at least *one* program can understand a language that fulfils the other criteria, on the appropriate machine, and given the appropriate resources, it can be considered a computer language. In particular, this includes many domain-specific languages, no matter how niche they are, including those that may be created and used by a single person or a very restricted community, similar to some rare dialects spoken by humans.

2.1.2 RELATED TERMS

The term *computer language* is different yet related to several other terms, and most notably *programming language*, *markup language*, *domain-specific language*, *data format* and *protocol*.[4] Although it is part of the name of Elsevier's *Journal of Computer Languages*[5] and used in some publications,[6]

4. Note that this choice of terms is neither methodic nor exhaustive, but an empirical choice of mine, based on my own experience and review of the literature. Other terms that describe computer languages could have be included in this list, such as *pattern language* (McLean and Wiggins, 2010), *transformation language* (Cordy, 2006) and *computer code* (Liu et al., 2020; Ivanova et al., 2020).

5. The journal was formlerly titled *Computer Languages, Systems and Structures*, but changed name in 2018.

6. See, for example, Shieber (1984) and Simonyi (1995).

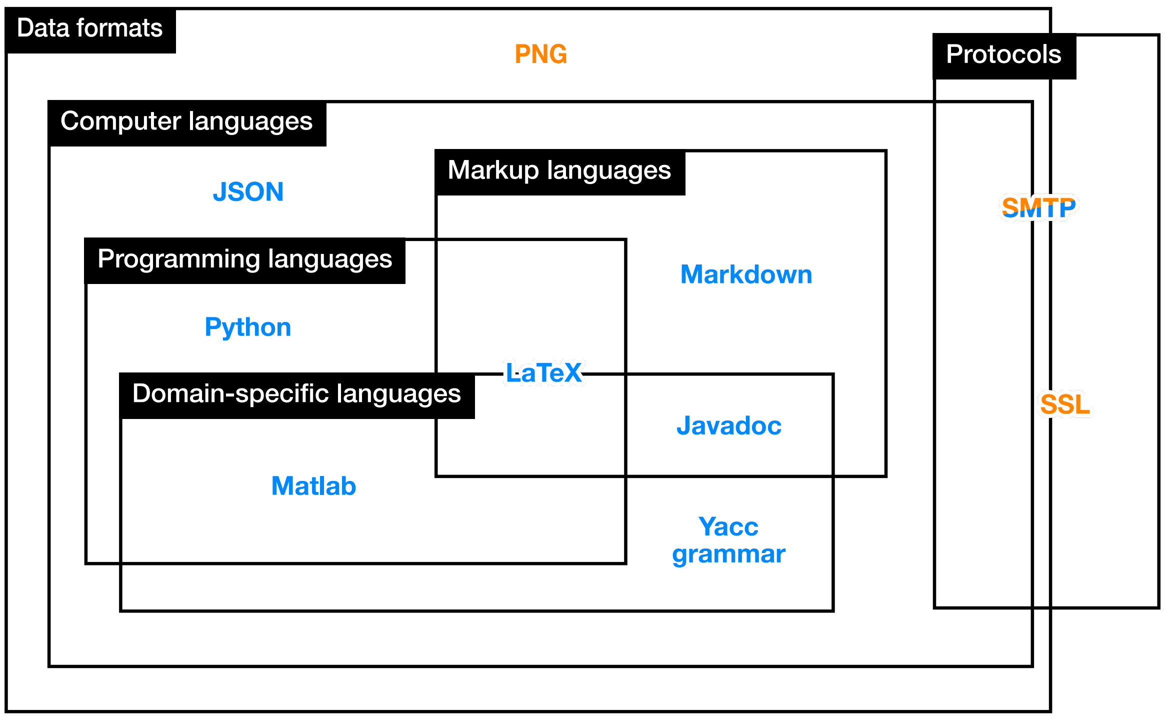


Figure 2.1. Venn diagram of the classes of languages represented by the different terms presented in section 2.1 as I describe them in the context of this thesis. Boxes represent sets of languages. Languages in blue represent positive examples of computer languages, and languages in orange represent negative examples. Since the definitions are somewhat porous, some languages are located at the edge between several classes to indicate that they may belong to one or the other depending on the context they are used in or the purpose they are used for. For example, LaTeX may or may not be considered specific to writing scientific and technical documents, and SMTP may only be considered as a computer language if a human manually reads or writes the protocol's messages.

Search term	**Number of matches**			
	Google Scholar	Semantic Scholar	Semantic Scholar (CS)	dblp
Computer language	98,100	15,300	13,100	2,572
Programming language	2,010,000	134,000	110,000	6,779
Markup language	235,000	15,300	13,000	685
Data format	267,000	34,200	23,100	1,107
Protocol	6,620,000	1,550,000	513,000	58,964

Table 2.1. Number of results returned by three search engines for scientific literature, provided different search terms related to computer languages. Each search term was surrounded with double quotes. Results under *Google Scholar* and *Semantic Scholar* are not specific to any field. Results under *Semantic Scholar (CS)* use a filter provided by the engine to only include results related to the field of computer science; and so do those under *dblp*, as the engine is designed to be specific to computer science research. All the searches were done in January 2024.

it appears to be less common in the scientific literature than most of these terms, as suggested by the data reported in Table 2.1. Yet, I argue that the notion of computer language as I define it is complementary to these other terms, in a way that is not well captured by any other term I know. In order to clarify what it encompasses and how all these terms are related, I briefly review the characteristics of these other classes of languages, along with a number of examples, as summarised in Figure 2.1.

Programming languages

Programming languages are computer languages designed for creating programs, i.e., sequences of instructions for controlling a computer that can either be executed directly by the hardware (for compiled languages) or by a proxy (for interpreted languages). Aho et al. (2006) report that since their inception in the 1940s, the way we program computers has drastically evolved, forming five generations of programming languages. In the early days of programming, *machine code* (generation 1), formed of simple instructions written in binary, was input through physical operations such as flipping switches. Machine code quickly evolved into *assembly languages* (generation 2), which included mnemonics and macros to abstract common sequences of instructions. Although they required some preprocessing, these languages were still formed of sequences of instructions representing the most basic operations available in the processing unit. However, the need of expressing more complex statements, such as mathematical formulae, and to write programs independently from the underlying architecture, led to the development of *high-level programming languages* (generations 3 to 5) such as Fortran, which need to be translated into machine code by more and more advanced compilers. Today, programming languages can take very diverse shapes, leading to various taxonomies, which may classify them by their generation, by the paradigms they are designed to support (imperative, functional, object-oriented, etc.), or even by the theoretical properties they exhibit (Turing-completeness, memory safeness, etc.) As a result, it is very hard to determine clear boundaries of what is and what is not a programming language, whose properties can, at best, serve as opinionated hints.

For example, some languages expose hardware-level information, such as different sizes of integers, e.g., `uint8_t` and `uint32_t` in C and `u8` and `u32` in Rust for unsigned integers encoded using either 8 or 32 bits. Others hide such details, such as Python (whose integers have no size limit) and JavaScript (whose numbers are all floating point numbers), although such hardware-related considerations may resurface in languages designed to hide them in unexpected ways.[7]

7. This often happens for performance reasons. For example, even though JavaScript does not distinguish between integer and decimal numbers, adding `|0` after a numeric expression in JavaScript can be used to force the JavaScript engine to treat the result as an integer. Before WebAssembly, this was used by asm.js (@6) to run JavaScript code transpiled from low-level languages such as C very efficiently compared to standard JavaScript. Similarly, Lucas Pluvinage (@7) showed how a trick with CPU caching called *value speculation* that was previously demonstrated by Francesco Mazzoli (@8) in C can actually be used in OCaml too, resulting in much faster code in certain cases, even though the technique requires to do pointer arithmetic, a programming technique OCaml is explicitly *not* designed to support.

Similarly, different languages offer different ways to control the flow of the program. Historically, this was implemented using instructions to jump to a particular memory address, either conditionally or unconditionally, a paradigm notably used in machine and assembly languages. The lack of clarity of this approach, famously pointed out by Dijkstra (1968), led to the *structured programming* paradigm, in which languages should offer syntactic structures that describe their intention when they alter the flow of the program (such as *while* and *for* loops). However, others paradigms recommend different approaches, such as favouring recursion over loops, as in *functional programming*, or simply specifying the desired computation without specifying how it should be implemented, as in *logic* and *constraint-based programming*. Overall, while the best kind of structure to offer to the programmer remains a debate, offering ways to control the flow of the program appears to be a key feature in all the computer languages considered to be programming languages that are in use today.

Markup languages

Markup languages are computer languages designed for annotating text documents, such as to alter the style of the text or specify the layout of the document. In these languages, the majority of the code is usually text, in addition to which some characters have a special meaning, e.g., pairs of * and _ to bolden and italicize text in Markdown, and commands can be inserted, e.g., `\includegraphics` to insert an image in LaTeX. Historically, markup languages have been developed to specify how a text document should look like when displayed on a screen or printed on paper, before it could be specified by interacting with the rendered version of the document as in *What You See Is What You Get* (WYSIWYG) editors—a practice which only became popular in the late 1970s and from the 1980s on, following the spread of direct manipulation interfaces (Shneiderman, 1983).

Nowadays, these languages have other uses. They are commonly used to let users of online communities format comments or articles without needing a rich text editor using, e.g., BBCode on various forum boards, Markdown on Reddit, and Wikitext on Wikipedia. They are also used to format text written in other computer languages, such as documentation comments in programming languages that can be interpreted by a program and turned into typeset or interactive documentation. Some markup languages can even be used as programming languages even though they were not designed for that purpose, as demonstrated by the Turing-completeness of TeX and LaTeX (Erdweg and Ostermann, 2011).

Domain-specific languages

Domain-specific languages are computer languages designed for a specific application domain, rather than a generic goal such as writing any kind of computer program or document. Just like some markup languages can also be used as programming languages, some domain-specific languages are also programming or markup languages. For example, the Matlab language is primarily designed for mathematics, in which efficient arrays and matrices are first class citizen of the language; but it can also be used to write any sort of program. Similarly, some markup languages are specifically designed to document code written in other computer languages, such as Javadoc for Java and JSDoc for JavaScript, usually by writing a comment just before the entity to document in the host language they are used in. There are also domain-specific languages that are neither programming nor markup languages. As an example, languages to describe grammars of other languages fit into this category. They generally inherit from common grammar notations, such as the Backus-Naur form (BNF), and exist in different flavours, which can be communicated as is to the public, such as Python's grammar (@9) in the official documentation of the language; be processed by a parser generator, such as Yacc (@10), Bison (@11) and ANTLR (@12); or be typeset in a document, such as with the *naive-ebnf* LaTeX package (@13).

Data formats & protocols

Unlike programming languages, markup languages and domain-specific languages, which are all computer languages, data formats and protocols are not always computer languages, as the "language" of a format or protocol is not systematically used by humans, nor designed for them. While image formats such as JPEG, PNG or GIF and protocols such as TCP and SSL may be considered as languages—in the sense that the former strictly codifies how a data structure is organised, and the latter strictly codifies a form of communication between computers—, they are not computer languages as defined earlier, as none of them is interpreted and expressed by humans. However, some data formats and some protocols have been or become used by humans, therefore making them computer languages, even though they were not designed with that purpose in mind. Examples include XML and JSON which, although initially devised as data storage formats made to be read and written by programs (respectively inspired by tree structures and JavaScript objects), are now commonly used for configuration files that are edited as text, as in the case of XHTML[8] for creating web pages and Vega (Satyanarayan et al., 2016) for specifying data visualisations. In addition, certain protocols may be considered as computer languages in specific cases in which a human writes messages and reads responses in place of a computer, such as for educational or debugging purposes. This is especially adapted to text protocols such as SMTP and HTTP, as demonstrated by the developer tools of web browsers such as Mozilla Firefox and Chromium, which both support manually writing and sending HTTP messages to debug a web server or application.

8. Unlike XHTML, the HTML language is not based on XML but on SGML, which has a more permissive syntax than XML (and is therefore more robust to syntax errors).

2.2 WHAT ARE COMPUTER LANGUAGES USED FOR?

While the first computer languages were created in the 1950s with the goal of simplifying computer programming, they have progressively been developed for other purposes as well, such as querying databases, describing documents, and teaching computer science to children. Nowadays, computer languages are manifold: they are used in various domains, by various people, for various reasons, as illustrated by the numerous examples shown in Figures 2.2, 2.3 and 2.5. This section gives an overview of five major goals computer languages are used for: developing applications, solving scientific and technical problems, writing documents, teaching computer science and creating art and design work. My goal in giving this panorama of computer languages is threefold. First, it further motivates the importance of studying and improving our interaction with computer languages, by showing how crucial and widespread these languages have become in modern societies. Then, it presents different users groups that can be targeted by user-centred design research, an approach I chose to use in the applied work I will present in chapters 6 and 7. Finally, it helps framing what computer languages are in a different way than the theoretic definition given in the previous section, by inferring what the term refers to from a large body of examples.

2.2.1 DEVELOPING USER AND BUSINESS APPLICATIONS

Broadly put, an application is a program designed to help a human perform a particular set of tasks. Applications may be one of the most common type of program, as demonstrated by the global use of some of them, such as word processors like Microsoft Word and web browsers like Google Chrome, and the millions of applications available on online *stores*, such as Apple's AppStore for iPhones and Google's PlayStore for Android phones. As a consequence, the computer languages used to develop them are crucial in the digital ecosystem we now live in.

Applications were first developed and adopted by businesses and public administrations to digitalise a number of tasks that used to be done by hand. Early examples include industries such as banking and finance, in which applications were developed to process worldwide transactions, and transportation and sales, to manage inventories and bookings; as well as governments. Accordingly named,[9] the COBOL language was developed in the late 1950s in order to help writing business applications in a machine-independent, high-level programming language. Other computer languages have been developed for or adopted by businesses in the following decades, such as SQL for retrieving and processing data stored in relational databases, Erlang for creating back-end applications designed to be highly available and easy to maintain, and Java for developing applications that could run on any computer. Today, these industries employ hundreds of thousands of programmers, not only to develop new features, but also to maintain decades-old systems that are still in use. As an example, industrial systems in banking rely on *billions* of lines of COBOL code to power critical services such as transactions and devices such as ATMs (@14), even though COBOL is often considered to be a deprecated language that is not taught in computer science curricula anymore. Similarly, several states in the USA reported issues processing a sudden and massive increase in the number of unemployment applications during the Covid-19 crisis, which overloaded the systems written in COBOL designed to process them that had not been replaced in time (@15).

9. COBOL is an acronym standing for *COmmon Business-Oriented Language.*

Outside of the business world, the increasing demand for applications, the pace at which features are being developed and added, and the scale and diversity of the public using them, also motivated the development of new computer languages. This is particularly visible in web and mobile applications developed by companies such as Google, Microsoft and Meta, which started by developing them using languages such as PHP and JavaScript but eventually switched to custom languages and frameworks they developed to better fit their needs. This includes Hack as a replacement of PHP, Kotlin as a replacement of Java, Reason as a replacement of OCaml, and Flow and TypeScript as replacements of JavaScript, complemented with front-end frameworks such as React and Angular. In addition to building on existing languages, new languages were also developed from scratch along the way, as demonstrated by Meta's GraphQL language for optimising API calls and Google's Dart language for developing cross-platform applications.

2.2.2 Solving engineering and scientific problems

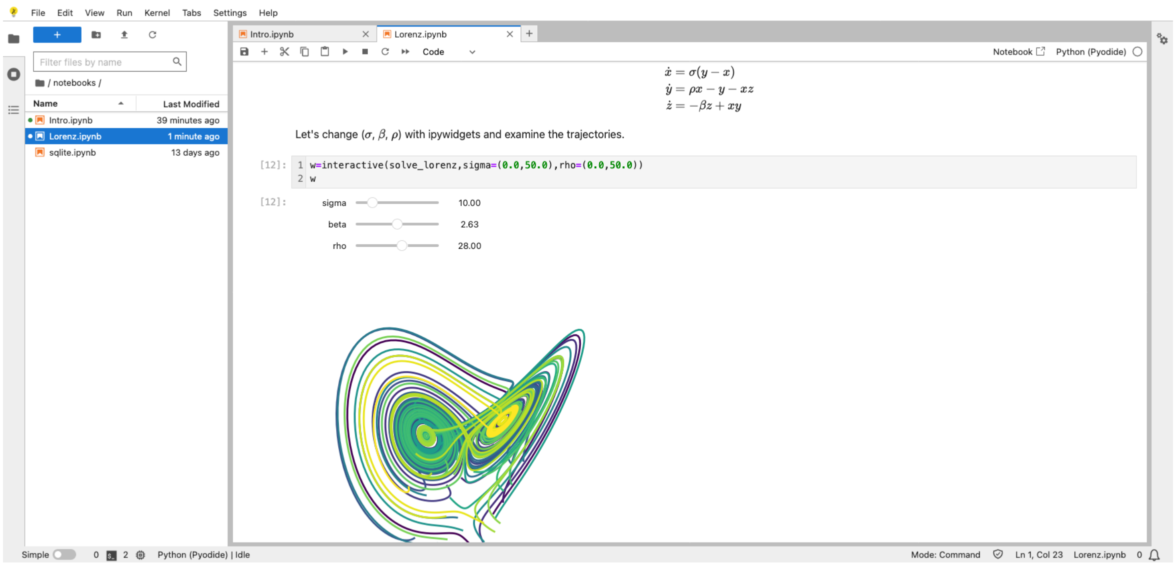
Since computers were initially devised to perform mathematical operations, such as ballistic computations, it is no surprise that computer languages were developed to assist engineers and scientists using them for that very purpose. Backus, who came up with the idea of Fortran in 1954,[10] one of the first programming languages ever created, report that "*one of our goals was to design a language which would make it possible for engineers and scientists to write programs themselves*" (Backus, 1978, p. 168). The language was later use to develop popular libraries for scientific computing, such as BLAS (@20) and LAPACK (@21) for linear algebra, some of which remain in use today despite the language being over 60 year old.

10. Similar to COBOL, Fortran (initially written in full capitals) is an acronym standing for *Mathematical FORmula TRANslating System*, as introduced in the preliminary report on the Fortran language published in 1954 (Backus, 1978, p. 168).

Nowadays, multiple computer languages can be used to perform mathematical operations efficiently, sometimes as a part of a larger computer algebra system (CAS). Languages such as Mathematica, Maple, Matlab and Julia were explicitly designed for that purpose, whereas systems such as Sage (@22) and libraries such as NumPy (@23) rely on Python. Furthermore, in addition to performing numerical and algebraic operations, computers are now also used to help writing mathematical proofs, requiring different sorts of systems and languages. The work of logicians from the 20th century and results such as the Curry-Howard correspondence, which states that constructing a program of a given type is equivalent to proving the logical proposition associated with that type, led to the development of theorem provers, which are systems designed to establish proofs with the help of a computer. Accordingly, special languages with type systems adapted to the task were developed for that purpose, such as Gallina for Coq (Bertot and Castéran, 2004), Isar for Isabelle/HOL (Nipkow et al., 2002), and Lean and F* for eponymous systems (de Moura et al., 2015; Swamy et al., 2016). In addition to resulting in mechanised proofs of major theorems, such as the four colour theorem from graph theory,[11] theorem provers have also been used to verify other programs, such as the Compcert compiler (Leroy et al., 2016), which guarantees that the generated executable code behaves in accordance with the formal semantics of the C language.

11. The initial proof of the theorem was devised by Appel and Haken in the 1970s with the help of a computer (Appel and Haken, 1977), but the program they conceived was not proved to be correct, and verifying the complete proof still required a certain amount of human labour. It is only in 2005 that Gonthier et al. published a fully automatised proof of the four colour theorem using the Coq theorem prover (Gonthier, 2008).

Verified software is of uttermost importance in application domains where safety is critical, such as in trains, planes and nuclear power plants. To satisfy critical constraints of real-time systems such as flight control systems, the operations performed by the program these systems run must not only be correct, but also complete under hard time constraints. The development of programming paradigms fit for this task in the 1980s, such as synchronous programming, was accompanied by the development of dedicated computer languages such as Esterel and Lustre. Moreover, in addition to modelling software, engineers must also model hardware, such as electronic circuitry. This can been achieved with yet another class of computer languages, called hardware description languages (HDLs), which include languages such as Verilog and VHDL. Besides specifying electronics that can be simulated and tested numerically, circuit descriptions written in HDLs are also used as input of programs designed to lay out electronic components and wires that



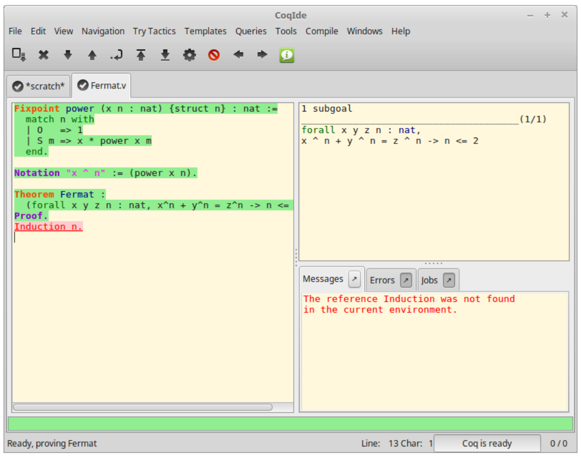
$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Let's change (σ, β, ρ) with ipywidgets and examine the trajectories.

```
[12]: w=interactive(solve_lorenz,sigma=(0.0,50.0),rho=(0.0,50.0))
      w
```

```
[12]:  sigma  10.00
       beta    2.63
       rho    28.00
```

a. JupyterLab.

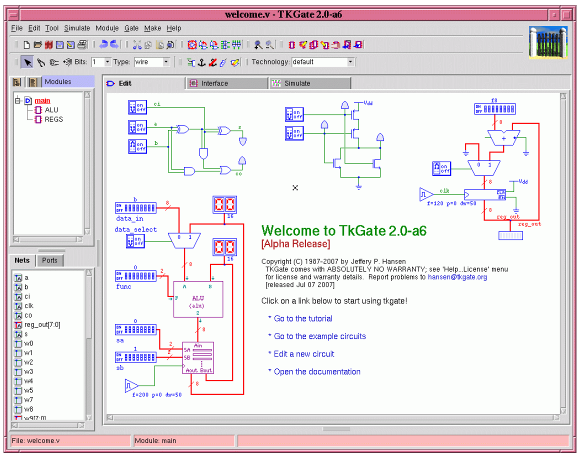


```
Fixpoint power (x n : nat) {struct n} : nat :=
  match n with
  | O   => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n : nat, x^n + y^n = z^n -> n <=
Proof.
Induction n.
```

```
1 subgoal
______________________________________(1/1)
forall x y z n : nat,
x ^ n + y ^ n = z ^ n -> n <= 2
```

```
The reference Induction was not found
in the current environment.
```

b. CoqIDE.



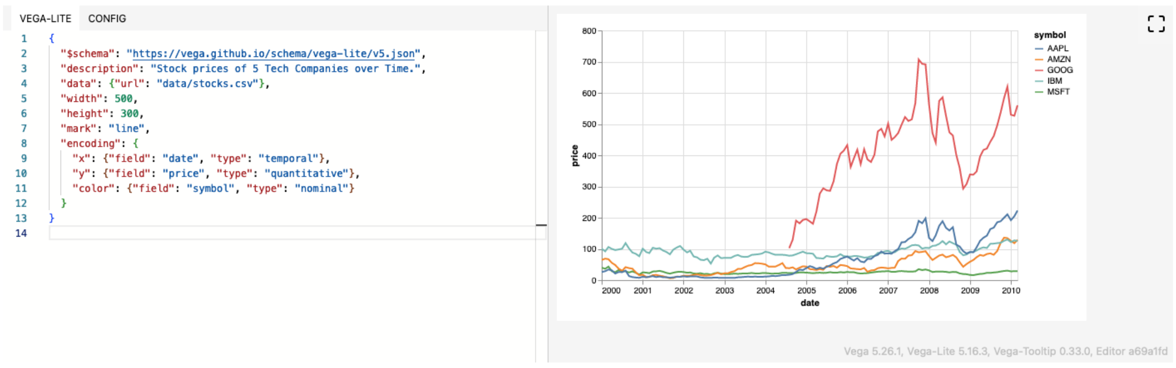
Welcome to TkGate 2.0-a6
[Alpha Release]

Copyright (C) 1987-2007 by Jeffery P. Hansen
TKGate comes with ABSOLUTELY NO WARRANTY; see 'Help...License' menu for license and warranty details. Report problems to hansen@tkgate.org
[released Jul 07 2007]

Click on a link below to start using tkgate!

* Go to the tutorial
* Go to the example circuits
* Edit a new circuit
* Open the documentation

c. TkGate.



VEGA-LITE
CONFIG
price
date
symbol
AAPL
AMZN
GOOG
IBM
MSFT
Vega 5.26.1, Vega-Lite 5.16.3, Vega-Tooltip 0.33.0, Editor a69a1fd

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v5.json",
  "description": "Stock prices of 5 Tech Companies over Time.",
  "data": {"url": "data/stocks.csv"},
  "width": 500,
  "height": 300,
  "mark": "line",
  "encoding": {
    "x": {"field": "date", "type": "temporal"},
    "y": {"field": "price", "type": "quantitative"},
    "color": {"field": "symbol", "type": "nominal"}
  }
}
```

d. Vega Editor.

Figure 2.2. Example uses of computer languages for addressing engineering and scientific problems. (a) User interface of JupyterLab (@16), a notebook environment. Cells containing text and mathematical formulae, written in Markdown and LaTeX, respectively, can be interwoven with cells of code written in languages such as Python or Julia that can be executed to display their output below the cell. (b) User interface of CoqIDE (@17), a system to write and prove mathematical theorems using the Gallina language of the Coq proof assistant. (c) User interface of TkGate (@18), a graphical system to draw and simulate electronic circuits specified according to a subset of the Verilog language, which can also be edited as text. (d) User interface of the Vega Editor (@19), an online code editor to specify and render data visualisations using Vega (Satyanarayan et al., 2016) or Vega Lite (Satyanarayan et al., 2017), two computer languages based on JSON.

connect them on buffers and printed circuit boards—a task that has become unfeasible by humans for complex circuits such as modern microprocessors, which contain billions of transistors.

Furthermore, computers also play a central role in science and engineering to help practitioners work with ever-growing amounts of data. The ability to work with data—from collecting it to analysing and visualising it—has become one of the most demanded skill on the job market in the past decade, and is not expected to plateau anytime soon.[12] To support these needs, various computer languages are being used and conceived. This includes general-purpose languages such as Python, often with the help of dedicated libraries such as pandas (@25) and Matplotlib (@26); specialised-languages such as R, purposely designed for statistical analysis; as well as other languages, such as JSON for visualising data with Vega (Satyanarayan et al., 2016). It also includes query languages, designed to collect and sometimes process data, such as SQL for relational databases, GraphQL for web APIs and SPARQL for semantic web graphs.

12. The Bureau of Labor Statistics of the USA indicates that the number of data scientists employed in the USA is expected to grow by 36% between 2021 and 2031 (@24).

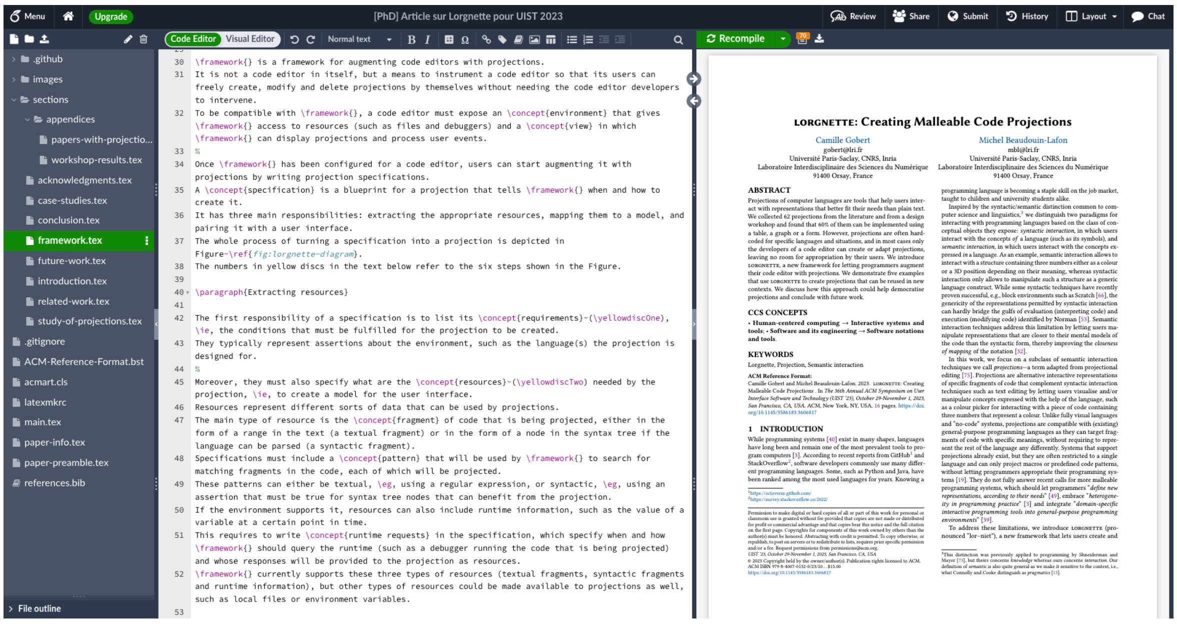
2.2.3 WRITING DOCUMENTS

In addition to processing data, computer languages can also be used to create data. Document description languages, in particular, are computer languages designed for creating digital documents by the means of a specialised language. Initially, these languages were designed to typeset documents to be printed. Early examples of such languages include the markup language used by the RUNOFF typesetting program, published in 1964, which later inspired troff (@32) and groff (@33), as well as Scribe (Reid, 1980), TEX (Knuth, 1984b) and LATEX (Lamport, 1994), which were all developed in the late 1970s and early 1980s. Despite being about forty years old, LATEX remains vastly used today for writing academic and technical documents, as reported by Knauff and Nejasmic (2014) and illustrated by the millions of users reported by Overleaf, an online LATEX editor, in 2021 (Reis et al., 2021).

Other document description languages are primarily designed for digital documents. This includes HTML, a language used to describe the tree structure of webpages, which is often complemented by CSS for styling these documents, as well as languages with less “intrusive” syntax than HTML tags, such as Markdown, reStructuredText (reST) and AsciiDoc, which are commonly used to provide basic formatting to participants of online platforms and messaging applications and programmers writing documentation comments in other languages.[13] Document description languages can also be specifically designed for creating interactive digital documents. For example, Idyll (Conlen and Heer, 2018) extends Markdown’s syntax to let writers insert widgets that depend on variables that can be controlled by the reader (such as buttons and sliders), and HeartDown (Li et al., 2022) and Nota (@36) let users write academic articles that include interactive explanations of symbols and interactive components, without requiring post-hoc annotations using systems such as ScholarPhi (Head et al., 2021) and Chameleon (Masson et al., 2020).

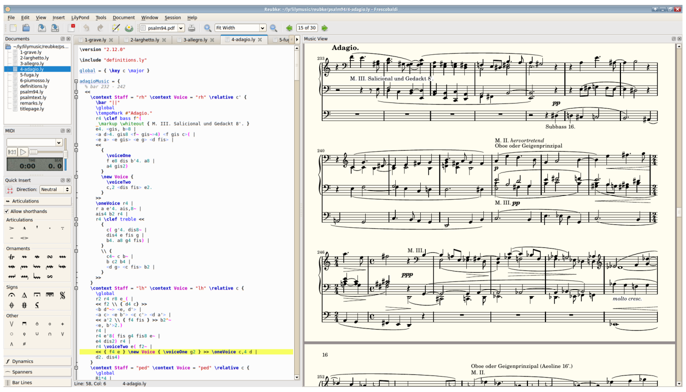
13. For example, PEP 287 (@34) specifies that reST is the default language for documentation comments in Python, and Rust’s official documentation (@35) states that documentation comments are interpreted as Markdown.

Certain computer languages have been designed to describe documents that mix markup and programming languages. The idea of weaving text and

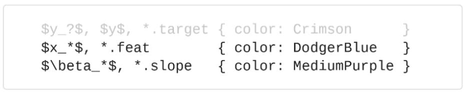


```
\framework{} is a framework for augmenting code editors with projections.
It is not a code editor in itself, but a means to instrument a code editor so that its users can
freely create, modify and delete projections by themselves without needing the code editor developers
to intervene.
To be compatible with \framework{}, a code editor must expose an \concept{environment} that gives
\framework{} access to resources (such as files and debuggers) and a \concept{view} in which
\framework{} can display projections and process user events.
%
Once \framework{} has been configured for a code editor, users can start augmenting it with
projections by writing projection specifications.
A \concept{specification} is a blueprint for a projection that tells \framework{} when and how to
create it.
It has three main responsibilities: extracting the appropriate resources, mapping them to a model, and
pairing it with a user interface.
The whole process of turning a specification into a projection is depicted in
Figure~\ref{fig:lorgnette-diagram}.
The numbers in yellow discs in the text below refer to the six steps shown in the Figure.

\paragraph{Extracting resources}

The first responsibility of a specification is to list its \concept{requirements}~(\yellowdiscOne),
\ie, the conditions that must be fulfilled for the projection to be created.
They typically represent assertions about the environment, such as the language(s) the projection is
designed for.
%
Moreover, they must also specify what are the \concept{resources}~(\yellowdiscTwo) needed by the
projection, \ie, to create a model for the user interface.
Resources represent different sorts of data that can be used by projections.
The main type of resource is the \concept{fragment} of code that is being projected, either in the
form of a range in the text (a textual fragment) or in the form of a node in the syntax tree if the
language can be parsed (a syntactic fragment).
Specifications must include a \concept{pattern} that will be used by \framework{} to search for
matching fragments in the code, each of which will be projected.
These patterns can either be textual, \eg, using a regular expression, or syntactic, \eg, using an
assertion that must be true for syntax tree nodes that can benefit from the projection.
If the environment supports it, resources can also include runtime information, such as the value of a
variable at a certain point in time.
This requires to write \concept{runtime requests} in the specification, which specify when and how
\framework{} should query the runtime (such as a debugger running the code that is being projected)
and whose responses will be provided to the projection as resources.
\framework{} currently supports these three types of resources (textual fragments, syntactic fragments
and runtime information), but other types of resources could be made available to projections as well,
such as local files or environment variables.
```

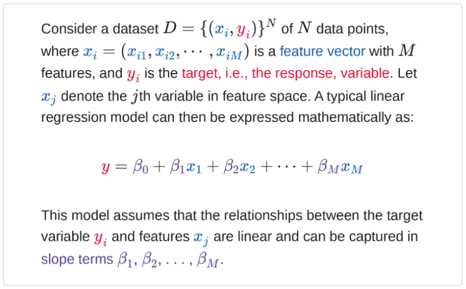
a. Overleaf.



b. Frescobaldi.



```
$y_?$, $y$, *.target { color: Crimson     }
$x_*$,  *.feat       { color: DodgerBlue   }
$\beta_*$, *.slope   { color: MediumPurple }
```

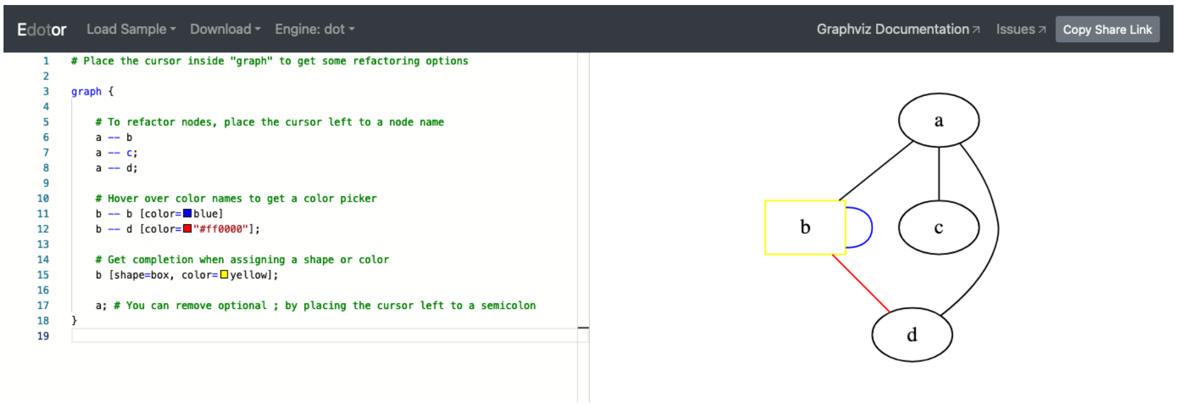


Consider a dataset $D = \{(x_i, y_i)\}^N$ of N data points, where $x_i = (x_{i1}, x_{i2}, \cdots, x_{iM})$ is a feature vector with M features, and y_i is the target, i.e., the response, variable. Let x_j denote the jth variable in feature space. A typical linear regression model can then be expressed mathematically as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_M x_M$$

This model assumes that the relationships between the target variable y_i and features x_j are linear and can be captured in slope terms $\beta_1, \beta_2, \ldots, \beta_M$.

c. FFL.



```
# Place the cursor inside "graph" to get some refactoring options

graph {

    # To refactor nodes, place the cursor left to a node name
    a -- b
    a -- c;
    a -- d;

    # Hover over color names to get a color picker
    b -- b [color=blue]
    b -- d [color="#ff0000"];

    # Get completion when assigning a shape or color
    b [shape=box, color=yellow];

    a; # You can remove optional ; by placing the cursor left to a semicolon
}
```

d. Edotor.

Figure 2.3. Example uses of computer languages for authoring digital documents. (a) User interface of Overleaf (@27), an collaborative online code editor for writing LaTeX documents that displays both the code (left) and the PDF generated by compiling it (right). (b) User interface of Frescobaldi (@28), a code editor for LilyPond (@29), a language for describing music sheets. The music sheet displayed on the right corresponds to the LilyPond code shown on the left. (c) Example use of FFL (Wu et al., 2023), a language for styling mathematical formulae written in LaTeX inspired by CSS. The FFL code shown at the top specifies rules for multiple patterns that, when applied to a document written in Markdown with LaTeX, outputs a document in which matching patterns have been formatted, as in the example shown at the bottom. (d) User interface of Edotor (@30), an online code editor for Dot, a language for describing graphs that shall be rendered as diagrams by Graphviz (@31).

code dates at least back to Knuth's concept of *literate programming* (Knuth, 1984a), an approach to programming in which a programmer alternates between writing code in a programming language and explaining the code using plain text—so that by using the right programs, the document can be processed to generate either code or documentation. This idea is at the root of systems such as Jupyter notebooks (@16), Observables (@37) and Codestrates (Rädle et al., 2017), which let users create documents by mixing cells made of either text, code, or the output of the code's execution, using languages such as Markdown, Python and JavaScript. It is also embodied in Catala (Merigoux et al., 2021), a language designed for writing legislative texts that contain parts that must eventually be transformed into a computer program (such as tax calculations), whose compiler verifies and automates the translation of such parts into executable code.

Furthermore, some document description languages are designed to describe (fragments of) documents that do not represent text but other sort of information, such as vector graphics, diagrams, and music. Some computer languages are specifically designed to describe images, such as SVG and PostScript (@38) for general-purpose vector graphics, Metafont (@39) for describing font glyphs, and TikZ (@40) for drawing images from within LaTeX documents. Similarly, DOT, Mermaid (@41) and Snapdown (Whatley et al., 2021) were designed to describe graphs and diagrams. Regarding music, LilyPond (@29) and Tabdown (@42) are respectively used to describe music scores and guitar tabs, whereas Alda (@43) is rather meant to be interpreted by the computer to play the notes written by its user.

2.2.4 TEACHING COMPUTER SCIENCE

Languages play a dually important role in teaching computer science, as they are both the object that is being taught to students and the communication medium used to explain that object to them. As the interest of the general public for programming, the availability of personal computers and the global spread of the internet grew, more and more initiatives to introduce children to programming were developed. This movement is highlighted by programs such as the Hour of Code (@44), which support parents and educators willing to introduce their children and students to programming, as well as nationwide stands and investments, such as the *Computer Science for All* initiative in the USA in 2016 (@45) and the introduction of computer science and programming lessons in French high-schools in 2019.[14] To achieve this goal, different sorts of computer languages are used. This includes languages specially designed for beginners and education—such as *mini-languages*, which are languages designed to be simple on purpose, and *sub-languages*, which are restricted versions of more complex languages (Gilsing et al., 2022)—as well as general-purpose languages, sometimes included in programming systems specifically designed for education.

14. Legal texts introducing the subject in the penultimate (@46) and the final (@47) years of high-school even specify that Python is the official language that must be taught to students.

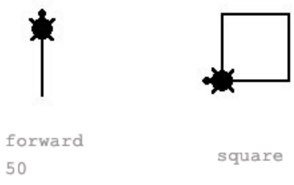
Early languages designed to introduce the general public to programming date back to the 1960s. The idea and first dialect of the BASIC language's family[15] were invented by Kemeny and Kurtz in 1964 as an attempt to make computing—and therefore, at that time, programming—more accessible to non-scientific students (Kurtz, 1978). BASIC eventually became widely

15. The acronym these languages are named after stands for *Beginner's All-purpose Symbolic Instruction Code*.

successful and available on many more systems than the computer and *Dartmouth Time-Sharing System* (DTSS) it was specifically designed for at its inception, becoming a staple of personal computing in the 1970s (@48). While BASIC was designed to help non-scientists use computers, the Logo language, created in 1967, was specifically designed to teach programming to children.[16] It is especially known for popularising the concept of *Turtle graphics* (Papert, 1982), a metaphor to help beginners draw graphics by controlling the movement of a virtual turtle (Figure 2.4). Among more recent examples of educative programming languages, Scratch (Resnick et al., 2009) may be the most successful one. Initially developed in Squeak, it was eventually redesigned as a web application and became one of the main system for teaching programming to children over the years, reaching 50 million users and hundreds of millions of projects in 2022 (@50).

16. The Logo Fundation reports that several researchers introduced Logo to elementary school students in the 1970s (@49).

a. The robot turtle.



b. The virtual turtle.

Figure 2.4. Logo's *turtle* was initially a physical turtle robot controlled by a computer; but it eventually became a virtual metaphor used for drawing. The images are reproduced from the Logo Foundation's website (@51).

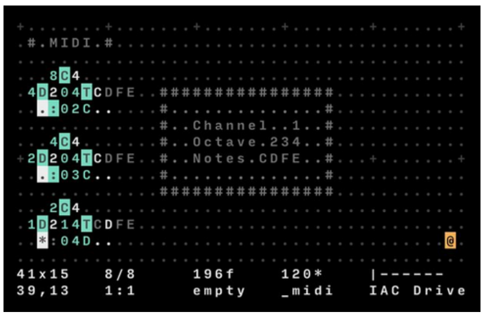
While educative programming languages are undoubtedly successful, they often remain educative tools which do not let their users harness the full power of the skills they learn by using them. For this reason, various initiatives have been taken to help novices learn and use computer languages that were not designed for education. As an example, the Alice (Cooper, 2010) and Greenfoot (Kölling, 2010) systems let their users manipulate entities in virtual worlds just as in Scratch; but unlike the latter, users must directly program in Java and learn some of the language's concepts. This deliberate design choice reflects the different age groups targeted by these systems, as discussed by their authors: "*CS1/pre-CS1 for Alice, 8-16 years old for Scratch, and 14+ years for Greenfoot*" (Utting et al., 2010, §2).

To help users transition from structured programming systems such as Scratch to various programming languages designed to be edited as text, hybrid programming environments that combine block and text editing have been demonstrated in prototypes such as PencilCode (Bau et al., 2015), Pencil.cc (Weintrop and Wilensky, 2017) and Tiled Grace (Homer and Noble, 2017). Accordingly, the Blockly framework (@52), which helps creating block editors for web applications, has been designed to support code generation in various general-purpose languages, such as Python and JavaScript. Another possible direction is to avoid the need of transitioning altogether. Gradual programming languages, such as Hedy (Hermans, 2020), include multiple *levels*, each with their own features and grammar, so that users can decide how much of the language they want to use depending on their expertise. This novel approach prevents learners from being overwhelmed with too many features from the start, while progressively learning a fully-fledged programming language (a subset of Python, in the case of Hedy).

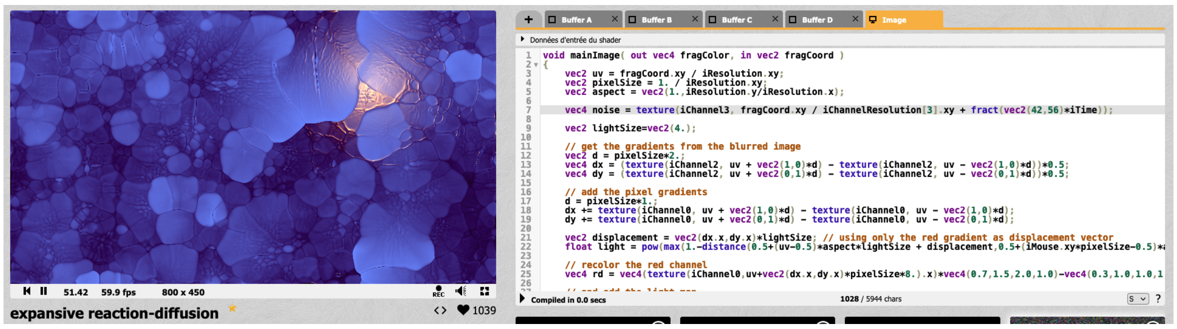
2.2.5 CREATING ART AND DESIGN WORK

Computers have been used for creative purposes since the early 1950s. In 1952, Christopher Strachey created a program at the source of the *Love letters* art piece: computer-generated love letters written by picking words in a database at random and printed on a teleprinter (Dreher, 2020, ch. 3). In the same decade, an unknown IBM engineer is reported to have digitalised and inserted pin-up images in a program ran by a military computer (@57). The 1960s, however, mark the beginning of widespread art creation using computers.

a. Algorave party.

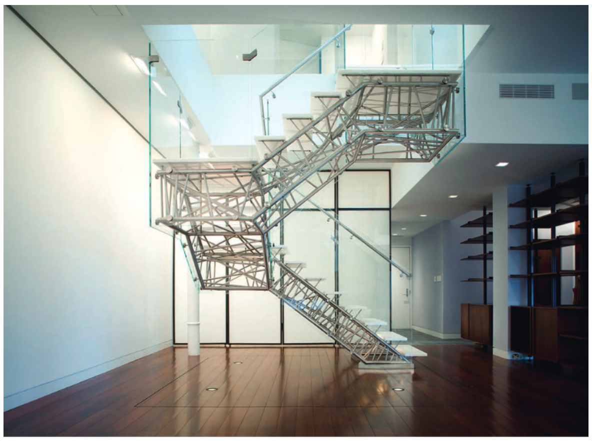


b. Orca.

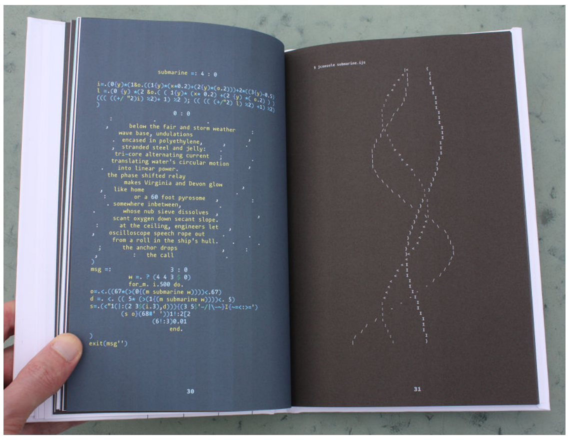


```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord.xy / iResolution.xy;
    vec2 pixelSize = 1. / iResolution.xy;
    vec2 aspect = vec2(1.,iResolution.y/iResolution.x);

    vec4 noise = texture(iChannel3, fragCoord.xy / iChannelResolution[3].xy + fract(vec2(42,56)*iTime));

    vec2 lightSize=vec2(4.);

    // get the gradients from the blurred image
    vec2 d = pixelSize*2.;
    vec4 dx = (texture(iChannel2, uv + vec2(1,0)*d) - texture(iChannel2, uv - vec2(1,0)*d))*0.5;
    vec4 dy = (texture(iChannel2, uv + vec2(0,1)*d) - texture(iChannel2, uv - vec2(0,1)*d))*0.5;

    // add the pixel gradients
    d = pixelSize*1.;
    dx += texture(iChannel0, uv + vec2(1,0)*d) - texture(iChannel0, uv - vec2(1,0)*d);
    dy += texture(iChannel0, uv + vec2(0,1)*d) - texture(iChannel0, uv - vec2(0,1)*d);

    vec2 displacement = vec2(dx.x,dy.x)*lightSize; // using only the red gradient as displacement vector
    float light = pow(max(1.-distance(0.5+(uv-0.5)*aspect*lightSize + displacement,0.5+(iMouse.xy*pixelSize-0.5)*a

    // recolor the red channel
    vec4 rd = vec4(texture(iChannel0,uv+vec2(dx.x,dy.x)*pixelSize*8.).x)*vec4(0.7,1.5,2.0,1.0)-vec4(0.3,1.0,1.0,1
```

c. Shadertoy.



d. The Genetic Stair.



e. Code Poetry.

Figure 2.5. Example uses of computer languages for creating art and design work. (a) An algorave party held in Lorient, France, on October 13th, 2022. The projected code is written using Sardine (Forment and Armitage, 2023), a musical live coding framework for Python. The photography was taken by Guillaume Kerjean and is reproduced from Forment and Armitage (2023, fig. 1). (b) User interface of Orca (@53), a live coding environment featuring a computer language designed for creating sequencers, in which each letter corresponds to a different operation. (c) User interface of Shadertoy (@54), an online community for sharing shaders written in GLSL that can be edited and rendered in real time using a text editor embedded in the webpage. The image shows the *expansive reaction-diffusion* shader, which was published on Shadertoy (@55) by Flexi on January 6th, 2016. (d) A stair in a renovated apartment in New York, USA. The steel pipe structure was created by the architects with the help of several algorithmic techniques—including finite-element analysis and a genetic algorithm—that were implemented using the RhinoScript and VBScript languages (Jabi, 2013, p. 80). The photography was taken by Ty Cole and is reproduced from Jabi (2013, p. 72). (e) The *submarine* poem (@56) from the Code Poetry book. The poem (left) is written in the J language and can be executed to display an animation (whose one frame is printed on the right).

Dreher (2020) and Dietrich (1986) report that the first art exhibition dedicated to computer-related art pieces were organised in the 1960s, including music, pictures and sculptures. At the time, artists were constrained to either use one of the few general-purpose programming languages available at the time, such as ALGOL and Fortran, or languages specialised for, e.g., drawing graphics, which were often specific to a particular computer model, such as the GRASS language, developed in the 1970s to "*bring the immense complexity of a Digital Equipment Corporation PDP-11/45 and a Vector General 3DR Display system within the grasp of artists and educators*" (DeFanti, 1980, p. 96).

Computers remained fruitfully used for artistic creation in the last 50 years, and have now become a standard tool used by various sorts of artists. In addition to the many general-purpose languages that now include modules for drawing on the screen or playing music, special languages have been developed for creative users who were not trained as computer scientists, such as Max/MSP, Pure Data and Processing. Albeit less accessible to non-experts, shader languages such as GLSL are also becoming more and more pregnant in creative communities such as Shadertoy (@54), an online platform on which artist-programmers share animations designed to be ran exclusively on GPUs. Furthermore, the growing interest for live coding is now challenging the role of computer languages. Live coding (Rein et al., 2018) is a type of performance art in which the artist creates the art piece by coding it in real time in front of the audience, such as during *algorave* parties and dance performances (Françoise et al., 2022). Since artists often project their screen to show the program they are writing, in accordance with the TOPLAP manifesto,[17] the computer languages used in the process transition from being hidden tools to being a part of the art pieces they help create. Again, languages used for that purpose can be split into two categories: those based on general-purpose languages, such as Haskell for Tidal (McLean, 2014), JavaScript for Gibber (Roberts and Kuchera-Morin, 2012) and Python for Sardine (Forment and Armitage, 2023), and those created from scratch for the task, such as sclang, the language of the SuperCollider audio synthesis system (McCartney, 2002).

17. TOPLAP (@58) is an organisation centred around live coding founded in 2004. The same year, they published a draft that describes the key components of a live coding performance, known as the TOPLAP manifesto (Blackwell et al., 2022, ch. 2), which includes statements such as "*show us your screen*" and "*code should be seen as well as heard*".

Besides artists, designers now commonly use computer-aided design (CAD) software to craft all sorts of products—from a small object to 3D print to an entire building—by modelling them digitally. Among other things, CAD software enable parametric design, a design practice centred around adjusting a number of parameters that control the aspect of a digital model. As an example, parametric design has been used by architects to explore spaces of configuration—a method sometimes called *combinatorial architecture* (Tiazzoldi, 2016)—both manually and with the help of algorithmic methods, eventually resulting in the creation of architectural artefacts in the real world. In his taxonomy of parameters, Jabi (2013, pp. 196–197) suggests that even though parameters most commonly include geometric and topological quantities, such as the size of an element or the number of times it is duplicated, other sorts of parameters may be of interest, from the properties of the material to environmental factors such as the amount of sunlight or the quality of a Wi-Fi signal. While most of the process can usually be performed by interacting with a graphical user interface, e.g., by manipulating sliders or parametrised elements of a 3D model (Michel and Boubekeur, 2021), several pieces of

Approach	Main conceptual tools	Concerns
Humanities	Semiotic and functional models	Conceptualisation, Communication
Psychology	Cognitive models, notations	Comprehension, Behaviour, Usability
Formal languages	Automata, Grammars	Computability, Complexity
Programming	Grammars, semantics	Implementation, Verification, Efficiency
Artificial intelligence	Probabilistic models, machine learning	Automation, Synthesis

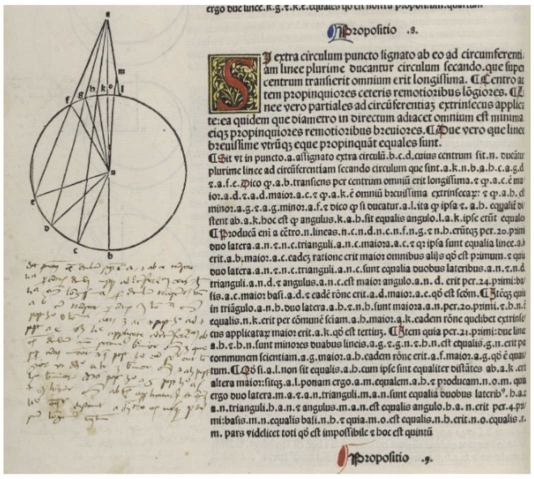
Table 2.2. Summary of the different approaches for studying computer languages presented in this section, including the type of conceptual tools used to represent and work with languages and the main concerns of each approach.

CAD software can also execute scripts written in computer languages, as demonstrated by the Python APIs of Blender (@59) and Rhino (@60), and software-specific languages, such as Autodesk's Maya Embedded Language (MEL) in Maya and MAXScript in 3ds Max.

2.3 HOW TO STUDY COMPUTER LANGUAGES?

Just like natural languages can be studied from various perspectives, ranging from poetry to neuroscience, studies of computer languages can look very different from one field to another. Because of their dual nature, which makes them fit for humans and computers alike, computer languages have mostly been studied from either a human-centric point of view, e.g., in psychology, or from a computer-centric point of view, e.g., in programming. Yet, studying computer languages from a human-computer interaction perspective requires a more holistic approach, able to capture the needs, abilities and limitations of both humans and computers. Unfortunately, no such theory currently exists.

In this section, I present a panel of theories and concepts that have been used to reason about computer languages, with the goal of identifying their strengths and limitations to inspire work on a new theory of computer languages. I first review how computer languages have been studied in five different research fields in the past—humanities, psychology, formal languages, programming and artificial intelligence—whose main conceptual tools and concerns are summarised in Table 2.2. For obvious reasons, this work should not be seen as an exhaustive coverage of the literature on computer languages, but rather as a review of both seminal and recent pieces of work on topics at the intersection between languages and computers, mostly ranging from the early 20th century to nowadays. I then address the particular situation human-computer interaction lies in. More specifically, I show that HCI lacks a theory of interaction for reasoning about and designing our interaction with computer languages beyond well-known but limiting views, such as the classic textual/visual distinction often found in the literature. I conclude by suggesting some directions to work towards a more holistic theory of interaction with computer languages.



ergo due linee.k.g.⁊.k.e.equales qđ eſt noſtru propoſitum.quartum

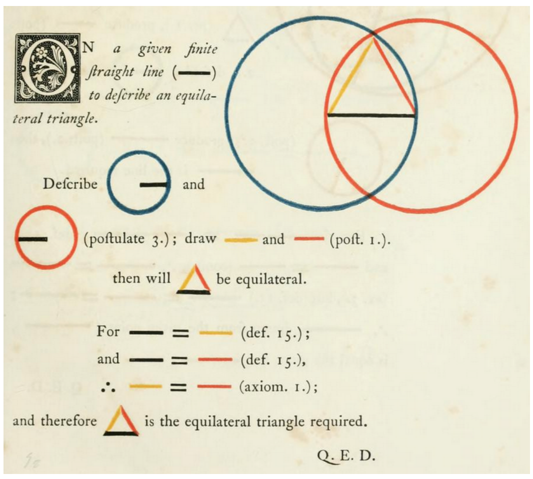
Propoſitio .8.

SI extra circulum puncto ſignato ab eo ad circumferentiam linee plurime ducantur circulum ſecando.que ſuper centrum tranſierit omnium erit longiſſima. Centro autem propinquiores ceteris remotioribus lõgiores. Linee vero partiales ad circũferentiaʒ extrinſecus applicate:ea quidem que diametro in directum adiacet omnium eſt minima eiqʒ propinquiores remotioribus breuiores. Due vero que linee breuiſſime vtrũqʒ eque propinquãt equales ſunt.

Sit vt in puncto.a.aſſignato extra circulũ.b.c.d.cuius centrum ſit.n. ducãtur plurime linee ad circũferentiam ſecando circulum que ſint.a.k.n.b.a.b.c.a.g.d.⁊.a.f.e.Dico ꝙ.a.b.tranſiens per centrum omniũ erit longiſſima.⁊ ꝙ.a.c.ẽ maior.a.d.⁊.a.d.maior.a.c.⁊ ꝙ.a.k.ẽ omniũ breuiſſima extrinſecaꝝ: ⁊ ꝙ.a.b.eſt minor.a.g.⁊.a.g.minor.a.f.⁊ dico ꝙ ſi ducatur.a.l.ita ꝙ ipſa ⁊.a.b. equaliter diſtent ab.a.k.hoc eſt ꝙ angulus.k.a.b.ſit equalis angulo.l.a.k.ipſe erũt equales Producã eni a cẽtro.n.lineas.n.c.n.d.n.e.n.f.n.g.⁊ n.b.erũtqʒ per.20.primi duo latera.a.n.⁊.n.c.trianguli.a.n.c.maiora.a.c.⁊ q2 ipſa ſunt equalia linee.a.b erit.a.b.maior.a.c.eadeʒ ratione erit maior omnibus alijs qđ eſt primum.⁊ quia duo latera.a.n.⁊.n.c.trianguli.a.n.c.ſunt equalia duobus lateribus.a.n.⁊.n.d trianguli.a.n.d.⁊ angulus.a.n.c.eſt maior angulo.a.n.d. erit per.24.primi:baſis.a.c.maior baſi.a.d.⁊ eadẽ rõne erit.a.d.maior.a.e.qđ eſt ſcđm. Itẽqʒ quia in triãgulo.a.n.b.duo latera.a.b.⁊.n.b.ſunt maiora.a.n.per.20.primi.⁊.b.n.ẽ equalis.n.k.erit per cõmunẽ ſciam.a.b.maior.a.k.eadem rõne quelibet extrinſecus applicataꝝ maior erit.a.k.qđ eſt tertiũ. Item quia per.21.primi:due linee a.b.⁊.b.n.ſunt minores duabus lineis.a.g.⁊.g.n.⁊.b.n.eſt equalis.g.n.erit per communem ſcientiam.a.g.maior.a.b.eadem rõne erit.a.f.maior.a.g.qđ ẽ quartum. Qđ ſi.a.l.non ſit equalis.a.b.cum ipſe ſint equaliter diſtãtes ab.a.k.erit altera maior:ſitqʒ.a.l.ponam ergo.a.m.equalem.a.b.⁊ producam.n.o.m. quia ergo duo latera.m.a.⁊.a.n.trianguli.m.a.n.ſunt equalia duobus laterib⁹.b.a.⁊ a.n.trianguli.b.a.n.⁊ angulus.m.a.n.eſt equalis angulo.b.a.n.erit per.4.primi:baſis.m.n.equalis baſi.n.b.⁊ quia.m.o.eſt equalis.n.b.erit.n.o.equalis.n.m.pars videlicet toti qđ eſt impoſſibile ⁊ hoc eſt quintũ

Propoſitio .9.

a. Ratdolt's edition (1482).



ON a given finite ſtraight line (——) to deſcribe an equilateral triangle.

Deſcribe ◯ and ◯ (poſtulate 3.); draw —— and —— (poſt. 1.).

then will △ be equilateral.

For —— = —— (def. 15.);
and —— = —— (def. 15.),
∴ —— = —— (axiom. 1.);

and therefore △ is the equilateral triangle required.

Q. E. D.

b. Byrne's edition (1847).

Figure 2.6. Extracts from two different editions of Euclid's *Elements*: (a) the first edition ever printed (@61), published by Erhardus Ratdolt in 1482, and (b) a more modern and colourful edition, published by Olivier Byrne in 1847. The images are respectively reproduced from the Folger Shakespeare Library (@62) and the Internet Archive (@63).

2.3.1 HUMAN-CENTRIC PERSPECTIVES

Humanities

The distinction between matter and form is a longstanding topic in philosophy. Early distinctions include those of ancient Greek philosophers, such as Plato and Aristotle. In his *theory of Forms*, Plato distinguishes the material world, made of alterable *matter*, from the real world, made of unalterable *forms*—permanent concepts we can only refer to (Rickless, 2020). The former represents the world we experience—imperfect and constantly changing—whereas the latter is a static, perfect world of concepts. Aristotle proposes a similar distinction in his *hylomorphic* view of the world in which, unlike in Plato's view, forms do not exist independently of matter, but only in conjunction with it (Ainsworth, 2020). These views of the world recognise the referential power of language, which gives humans the ability to manipulate and communicate on forms from the physical world we live in. It was used in early mathematical treaties such as Euclid's *Elements*, which describes and manipulates "ideal" mathematical concepts such as points, lines and regular shapes with the help of language (to refer to them) and drawings (to eschew them), as shown in Figure 2.6.

At the end of the 19th century, the development of analytical philosophy by philosophers such as Frege, Russel and Wittgenstein progressively put language and logic at the centre of the philosophical method—a period that was retrospectively described as the *linguistic turn* by Rorty (2007). This increased focus on language as an object of study, rather than merely as a medium, as in philology, led to the foundation of modern theories of signs: the semiotics of Charles Sanders Peirce, issued from his work on logic in the late 19th century (Peirce, 1976), and the linguistics of Ferdinand de Saussure, which he founded in his *Course in General Linguistics* in 1916 (Saussure, 1916). Pierce proposed to model a semiotic system with a triadic model, in which a symbol (the *sign*) only refers to a concept (the *object*) because the interpreter understands that symbol in a peculiar way (the *interpretant sign*):

> *A sign is something, A, which brings something, B, its interpretant sign, determined or created by it, into the same sort of correspondence (or a lower implied sort) with something, C, its object, as that in which itself stands to C.*
>
> — *The New Elements of Mathematics* (Peirce, 1976)

Saussure, on the other hand, proposed a simpler approach with a dyadic model, in which every *sign* is a two-sided object with a symbol (the *signifier*) that refers to a concept (the *signified*) through a relationship that he considers to be arbitrary:

> *We propose to retain the word* sign *to designate the total, and to replace* concept *and* acoustic image *by* signified *and* signifier, *respectively. The latter terms have the benefit of highlighting the opposition that separates them, either from each other, or from the total they form together. As for* sign, *we settle with it because we do not know what to replace it with since the common language does not suggest any other.*
>
> — *Course in General Linguistics* (Saussure, 1916)

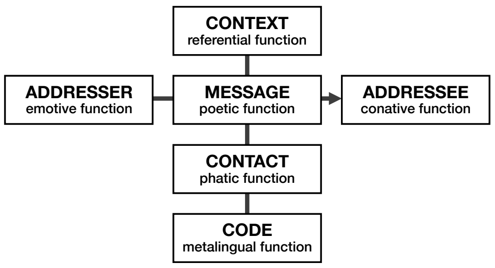
In his *Fundations of the theory of signs*, Morris (1938) proposes a model compatible with this early work. He decomposes semiosis, "*the process in which something functions as a sign*" (§2), into four components: the element that acts as a sign (the *sign vehicle*), what it refers to (the *designatum*), the person who interprets it (the *interpreter*), and the interpretation itself (the *interpretant*). In addition, Morris proposes that every sign system can be studied through three different lenses: the *syntactic* lens, which concerns the relations between the signs themselves; the *semantic* lens, which concerns the relations between the signs and the objects they refer to; and the *pragmatic* lens, which concerns the relations between the signs and those who interpret them.

Overall, these semiotic models are in line with Plato and Aristotle's distinctions between forms and matter.[18] They all highlight that semiotic systems form a bridge between something in the material world (sign, signifier, sign vehicle) and something in the ideal world (object, signified, designatum) through a relationship that is more or less arbitrary (interpretant), although dependent on the person interpreting it (interpreter).[19]

Semiotic theories have previously been used to analyse and design our interaction with computer systems by treating them as sign systems (Nadin, 1988; Andersen, 1992; de Souza, 1993). A number of authors also specifically applied these theories to study computer languages. For example, while only the notions of syntax and semantics have become common knowledge amongst programming language users, Zemanek (1966) and Connolly and Cooke (2004) both argue that all three aspects of semiotics identified by Morris, including pragmatics, apply to programming languages. Similarly, Tanaka-Ishii (2006) explicitly relates Pierce and Saussure's models to object-oriented and functional programming paradigms.

18. Their work also echoes work in other disciplines, such as Freud's work on the psychoanalytic theory at around the same time, in which the role of the psychoanalyst is to decode the unconscious meaning (signifieds) of the patient's language (signifiers).

19. The amount and importance of inter-personal variance in that interpretation process remains an active scientific debate, fueled by theories such as the Sapir-Whorf hypothesis. However, I leave its discussion out of the scope of this thesis for the sake of brevity.



Component	Function	Purpose
Addresser	Emotive	Expressing feelings.
Addressee	Conative	Acting on others.
Message	Poetic	Valuing the form.
Context	Referential	Referring to the world.
Contact	Phatic	Supporting the communication.
Code	Metalingual	Describing the language itself.

Figure 2.7. Scheme of Jakobson's model of language. Each rectangle represents a component of the model, arranged according to Jakobson's original scheme (Jakobson, 1960). The table on the side describes the meaning of each of the six functions.

The applicability of theories and models initially devised for languages other than computer languages extends beyond the form they take. Following the structuralist approach to language initiated in the early 20th century, Jakobson (1960) introduced another model in the 1960s, this time focusing on the functions of language rather than its structures and notations. Jakobson defines six complementary functions of language, each of which is represented by a component of the model (Figure 2.7). Although it was designed for natural languages, Jakobson's model can be applied to computer languages too, though some functions appear to be more prevalent than others. For instance, all programming languages are, by definition, used to act on the computer by making it execute a series of instructions (conative function); and it is common to talk about the meta-programming capabilities of a language (metalingual function), as illustrated by, e.g., Lisp macros and C++ templates (Lilis and Savidis, 2019).

Computer languages can also be used for less obvious functions. Their use in works of art—as in what Sondheim calls a *codework* (Memmott, 2011)—demonstrate their poetic function, as illustrated by the *code poetry* project (@64), which uses programming languages as poetic media in which both the code itself and its execution are part of the art piece. Furthermore, comments in computer languages may be considered to serve an emotive function in situations where programmers use them to, e.g., make jokes or express their frustration. Such comments can be found in online threads of funny comments spotted or written by programmers, as on StackOverflow (@65), as well as in code bases, as in Quake III Arena's popular implementation (@66) of the fast inverse square root function, in which two lines performing hard-to-understand bitwise operations are respectively commented with "*evil floating point bit level hacking*" and "*what the fuck?*".

Psychology

Advances in the field of cognitive psychology during the second half of the 20th century, such as the development of models of short- and long-term memory by Miller (1956), Atkinson and Shiffrin (1968) and Baddeley and Hitch (1974), were eventually used to apply findings in psychology to programming and computer languages, as reported by Blackwell et al. (2019). In his article on the semiotics of programming languages published in 1966, Zemanek states that "*reading, learning and teaching of programming languages are psychological problems on which the success of a programming language*

Dimension		Meaning
Viscosity		How hard is it to make a meaningful change?
Consistency		How much can be inferred from what is already known?
Diffuseness		How many different symbols are needed?
Premature commitment		How constrained is the order in which to do things?
Hidden dependencies		How much does modifying a single entity affects the rest of the system?
Hard mental operations		How demanding is the notation in terms of cognitive resources?
Role-expressiveness		How hard is it to understand the role of an entity in the system?
Abstraction	*	How much redefinition of the ordinary notation is permitted?
Visibility	*	How easily and immediately accessible is the information?
Error-proneness	*	How easy is it to make notational errors?
Secondary notation	*	How much information can be expressed independently from the syntax?
Closeness of mapping	*	How close to the application domain is the notation?
Progressive evaluation	*	How much of incomplete notations can be evaluated preemptively?
Provisionality	**	How much of an exploratory process can be recorded?

Table 2.3. Description of the dimensions of the *cognitive dimensions of notations* framework. The dimensions are sourced from three articles: those initially present in Green's original article from 1989 (Green, 1989) carry no mark; those which appeared in Green and Petre's article from 1996 (Green and Petre, 1996) are marked with an asterisk (*); and those which appeared in Blackwell et al.'s article from 2001 (Blackwell et al., 2001) are marked with two asterisks (**).

may depend much more than on all its technical properties." (Zemanek, 1966, §4). Accordingly, in the following decade, Sime et al. (1973) compared two type of conditionals for a programming language, Brooks (1977) proposed to model programming activities with short- and long-term memory structures complemented with symbolic rules for advancing the model's execution,[20] and Shneiderman and Mayer (1979) highlighted the difference between syntactic and semantic knowledge in programming. This early interest for the psychological aspect of programming eventually led to dedicated groups of interest at the frontier between human-computer interaction and programming, such as the *Psychology of Programming Interest Group* (@67), funded in 1987.

20. Brooks' model is reminiscent of later cognitive models such as ACT-R—both of which were inspired by Newell's work (1972; 1994).

The growing interest for studying programming languages from a psychological perspective at that time led to studies of the impact of their notations. In the 1980s, individual properties such as indentation (Miara et al., 1983) and colours (Rambally, 1986) were shown to impact program comprehension, and so were short but meaningful lines of code—*beacons*, in Wiedenbeck's vocabulary (Wiedenbeck, 1986). Following that line of work,[21] Green (1989) introduced a more holistic framework called the *cognitive dimensions of notations* in 1989, in which he defines several dimensions that each characterise an independent aspect of a *notation*—a term Blackwell et al. later defined as "*marks made on some medium*" (Blackwell et al., 2001, §3). The dimensions of the framework are summarised in Table 2.3.

21. According to Green and Petre (1996, §2), the framework is founded on the study of the activity of programming, from the point of view of psychology and human-computer interaction.

This framework is particularly adapted for "*written-down, symbol-based systems*" (Green, 1989, §3), such as various sorts of computer languages, though Green also mentions that it "*also applies to interactive languages*" (§3) that describe sequences of operations that characterise an interaction.[22] While the dimensions themselves are focused on the notation, the overall framework is more general and posits that every notation is part of a larger *notational system*. Green (1989) defines that system as a combination of a notation and an environment (§2), i.e., the context the notation is used, perceived and modified in—therefore including the device the notation is displayed on and

22. For a more extended analysis of the language metaphor to characterise interaction, see, e.g., Nielsen (1986) and Baudel (1995, ch. 7).

the interaction techniques available to the users of the language. Blackwell et al. (2001) further argue that multiple notations can cohabit within a single system, as illustrated by different graphical programs running in different windows at the same time, each with their own notation.

While Green's framework has been largely adopted to analyse notations of computer languages, Conversy (2014) reports on three alternative. The first alternative dates back to the first half of the 20th century, when Gestalt psychologists proposed several principles, known as *Gestalt principles*, which characterise how humans interpret perceptual cues (Rock and Palmer, 1990). These principles notably describe how separate objects can be related to each other based on their proximity, their similarity, their alignment, etc, to form new entities that are more than the parts that compose them. The second alternative was proposed in the 1960s by Bertin in his *Semiology of Graphics* (Bertin, 1967), in which he isolated seven *visual variables* that describe independent dimensions that the human visual perception system can distinguish: position, shape, size, colour, brightness, texture and orientation. The last alternative comes from Moody (2009b), with the goal of analysing diagrammatic programming languages. In contrast with the rather abstract dimensions of Green's cognitive dimensions, Moody proposed to build on Gestalt principles and Bertin's semiology to rather focus on the visual aspects of notations, leading to the *physics of notations* framework. The framework includes nine vision-oriented principles to design and evaluate notations, such as *perceptual discriminability* (different symbols should be visually distinguishable), *graphics economy* (the visual noise should be minimised), *semantic transparency* (the representation should suggest the meaning) and *dual coding* (text and graphics should be complementary).

2.3.2 COMPUTER-CENTRIC PERSPECTIVES

Formal languages

The 20th century saw the rise of mathematical models of language, which are tightly coupled with the development of computer science. In his seminal work on the theory of computation in the 1930s, Turing introduced the concept of *Turing machines*, theoretical constructs that operate on a finite *alphabet* of *symbols* that can be read and written on an infinite *tape* by a head controlled by an *automaton*. The movement of the head and the action it should take (reading or writing a symbol) is controlled by a graph of *states* and *transitions* between states that are conditioned by the symbol currently under the tape's head. Turing introduced this formalism with the goal of defining a theoretical framework to mechanically process languages, in such a way that each machine encodes an algorithm that decides whether a sequence of symbols written on the tape—a *word*—is part of the machine's language or not. Rather dually, lambda calculus, the other major model in the theory of computation, can itself be considered a formal language. Developed at around the same time by Church, it was later shown to be equally expressive as a Turing machine and used as a foundation for paradigms such as functional programming and languages such as Haskell.

In the 1950s, Chomsky introduced the concept of *formal grammar*, another formalism for describing languages. A formal grammar is made of *terminal symbols*, *non-terminal symbols*, and *production rules* that describe how non-terminal symbols can be rewritten as sequences of symbols. According to this theory, specifying a grammar and deciding on a special non-terminal symbol—called an *axiom*—fully describes a language, whose all sentences can be generated by applying the right sequence of production rules (starting from the axiom). Chomsky further showed that grammars can be categorised by constraining the type of production rules they can include, resulting in a hierarchy of grammars known as the *Chomsky's hierarchy*, in which more general types of grammar include less general types of grammar (Figure 2.8). Incidently, this hierarchy also categorises the languages that can be generated by these different classes of grammars.

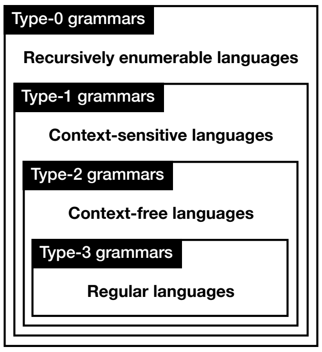


Figure 2.8. Classes of languages that can be generated by the four types of grammars described in Chomsky's hierarchy (Chomsky, 1959). The inclusion of a rectangle into another represents the fact that more general classes of languages also include less general classes.

Automata-based models such as Turing machines and formal grammars are two kinds of formal models of languages that are duals of each other. The former describes a language intensionally, by specifying how to decide whether a given word belongs to that language or not. Conversely, the latter does it extensionally, by specifying how to generate every word that is part of that language. That correspondence between two definitions of formal languages is, in fact, mathematically proved: Chomsky (1959) showed that the set of languages that can be recognised by a Turing machine is exactly the set of language that can generated by a type-0 grammar (recursively enumerable languages). This also holds for more restricted classes of languages, such as context-free languages, which are generated by type-2 grammars and recognised by pushdown automata (in which the tape is only a stack), and regular languages, which are generated by type-3 grammars and recognised by finite-state automata (in which there is no tape).

The idea that real computers, which are no more capable than Turing machines, can only have a limited understanding of the computer languages they manipulate, has consequences on the design of computer languages and tools to work with them. In particular, among all desirable properties for computer languages, some cannot be computed at all; and among all computable properties, some are harder to compute than others.

The first observation is based on the notion of *computability*, i.e., what can be computed by any computer *at all*, enriched by fundamental results of what can and cannot be computed, such as Rice's theorem. As an example, the fact that it is not possible to create a Turing-complete programming language in which all potential runtime errors can be detected before the program is executed,[23] motivates choices such as including an exception mechanism (at the language level) and adding runtime checks (at the library level) to mitigate crashes, as well as the development of appropriate debugging tools.

23. For example, statically determining that a program written in C is free of any out-of-bound array indexing operation is known to be incomputable.

The second observation is based on the notion of *complexity*, which classifies algorithmic problems in complexity classes, akin to Chomsky's four types of grammars, which, informally speaking, determine how much time and/or memory is needed to solve a particular problem. This affects design choices such as the syntax of computer languages, which may or may not be parsed by an efficient parsing algorithm depending on the grammar of the language. For example, the grammars of many programming languages have been designed to be compatible with efficient algorithms that can only parse certain classes of languages, such as LL(n) and LR(n) parsing algorithms, which require that at most n symbols/tokens need to be looked ahead to decide which rule to use. This explains why many languages prefix different sorts of constructs with distinct keywords or characters, as a means to diminish the ambiguity of the grammar, and therefore the parsing time.

Programming

Languages play a key role in programming, as the main way to let a human programmer specify what a computer should do. However, unlike formal models of language that are designed for mathematical reasoning only, programming requires operational models of languages. As programming languages evolved, they progressively became more and more distant from machine code, both in terms of syntactic features (conditions, loops, functions, classes, etc.) and semantic expressivity (richer primitive types, collection types, polymorphic functions, etc.) Accordingly, the major concern of programming regarding computer languages are the design process, which decides what goes in a language and how, and the translation process, which turns code written in that language in the only dialect that can be processed by a computer—machine code. To accommodate this evolution, the translation process performed by compilers and interpreters evolved as well, constrained by theoretical models of computing on one side, and practical engineering considerations on the other side.

The first step of the translation process is to represent the code in a form that is convenient to be processed by a computer. This is usually performed by assembling symbols—or groups of symbols called *tokens* if the language was pre-processed by a lexical analyser, or *lexer*—to form a *syntax tree*.[24] To that end, the syntax of a programming language is often specified in the form of a formal grammar, which is used as a reference to conceive the syntactic analyser, or *parser*, that will generate the syntax tree, either manually or by feeding the grammar to a parser generator.

24. Syntax tree can be further classified as *concrete syntax trees* (CSTs)—also called *parse trees*—or *abstract syntax tree* (ASTs). The former include every node generated by the syntax, while the latter discard certain information and only represent higher-level syntactic structures that are meaningful for the task (Aho et al., 2006, §2.5.1). However, since the difference does not matter in the context of this thesis, I often do not distinguish between the two, and simply refer to the general concept of a syntax tree.

Once the syntax tree is constructed, the next step is to enforce various semantic properties, mostly with the help of a *type system*. A type represents a piece of semantic information about a value. While types were initially used to determine the amount of memory to allocate for a value, their purpose evolved to also help ensure syntactically valid programs are also semantically correct, such as by verifying that a given function is only called with the right type and number of arguments. In order to conceive rules that effectively verify properties of interest while making such constraints expressible in the language, possibly with some degree of inference (to avoid specifying every type by hand), type systems must be carefully designed. Since verifying such properties means *proving* that a program verifies them, programming languages are increasingly formalised as mathematical objects and rules used to specify the semantics of the language before turning them into type inference and/or type checking programs. Furthermore, other theories are sometimes used to provide additional guarantees about the behaviour of a piece of code. For example, abstract interpretation can be used to constrain the set of values that a variable can take, allowing to prevent situations such as integer overflow/underflow and divisions by zero.

While such theories help conceive safe programming languages, they do come at a cost, and are not always compatible with other practical constraints, such as the usability of the language. Memory management is a classic example of such trade-offs: choosing a memory management approach rather than another affects the concepts the language must build on; the primitives that must be made available to its user; and the machine code generated at the end of the process. Incidently, different computer languages make different choices on this matter.

Certain languages, such as C and C++, include primitives to let the user of the language manually allocate and free memory. This approach has been criticised[25] but remains in use today, for it tends to be simpler to implement, as it delegates the responsibility to the language's users; can yield better performance, as the control over memory operations is finer; and may be required for certain tasks, as working on operating systems or embedded electronics often requires to work directly with the computer's memory. In response, various other mechanisms have been developed to alleviate the users from the burden of managing the memory by themselves. They include semi-automatic mechanisms such as *smart pointers*, which must be manually created by users but automatically destroy the object they point to when they become out of scope if they are the last reference to that object, as well as fully-automatic mechanisms such as *garbage collectors*, like those of Python and Java's runtimes.

25. The existence of a null pointer even being qualified as a "billion-dollar mistake" by Tony Hoare (@68), one of its creators, due to the many runtime errors it has caused.

To make the most of both worlds, languages with smart but complex way of automating memory management such as Rust are now being developed.[26] Yet, this added complexity has been shown to make Rust particularly hard to learn (Fulton et al., 2021). As a consequence, Coblenz et al. (2022) suggest that providing a garbage collector library (at the cost of performance) may be a trade-off to help users learn concepts such as ownership and aliasing, even though it precisely defeats the very purpose of Rust's approach to memory management.

26. Rust has a memory management system that tracks when to allocate and free memory at compile time. Unlike other approaches, it does not require the user to manually perform these operations, nor does it induce any runtime overhead. The system works by including the *lifetime* of every variable in their type and by constraining what can be done with a variable depending on which piece of code *owns* and *borrows* it at every computational step, which requires Rust users to adapt their mental models of the code to this singular approach.

Artificial intelligence

Artificial intelligence can be considered as a field of computer science concerned with giving computers capabilities that resemble those we deem as intelligent in animals, and more specifically in humans, such as their cognitive functions. Among all the directions taken by this field, making computers analyse and synthesise languages has been at the centre of artificial intelligence from early on. In his seminal work on artificial intelligence published in the early 1950s, Turing (1950) questioned whether machines can think and devised the *Turing test*—that he called the *Imitation Game*—as a means to test this hypothesis. This test works as follows: a human must discuss with another entity by exchanging messages written in a natural language, thinking the entity is another human, whereas the messages are actually read and written by the computer taking the test. If the human does not notice that they are not talking with another human being, the computer passes the test; otherwise, it fails the test. According to Turing, machine intelligence is therefore highly correlated to its capacity to understand and synthesise sentences written in a natural language.

This vision started a long line of research on natural language processing. The first computer program that is considered by some to have—to some extent—passed the Turing test is the ELIZA program, conceived in 1966 (Weizenbaum, 1966). ELIZA used a set of rules to transform the human's messages by detecting keywords and applying transformation rules to generate an answer. It belongs to a wide category of *rule-based* artificial intelligence models, in which decision rules are pre-programmed by human experts. However, ELIZA's "intelligence" is very questionable, as it does not build an artificial representation of what the user says, and simply rephrases their sentences based on predefined scripts to simulate a very generic human-like conversation. French (2000) reports that the initial optimism towards creating a machine able to pass the Turing test decreased from that time on, even leading Minsky to state that "*The AI problem is one of the hardest ever undertaken by science*" in 1982 (Kolata, 1982, p. 1238).

Starting from the late 1980s though, the development of machine learning models, in which decision rules and features of interest are no longer explicitly designed by the programmer but learnt by the computer, led to several breakthroughs in natural language processing. In 1989, LeCun et al. (1989) showed that convolutional neural networks were significantly more performant than other methods for automatically recognising handwritten ZIP code digits on US mail. In the 2010s, the use of long short-term memory cells in neural networks designed to process natural languages was shown to drastically improve the performance of the models (Sundermeyer et al., 2015) and ended up being rapidly used in commercial products such as Android's speech recognition system (@69). More recently, the development of large transformer models such as GPT-3 allowed to reach unprecedented performance in text synthesis. The potential of the latter became especially visible with the release of OpenAI's *ChatGPT* chatbot at the end of 2022, which is based on the GPT-3.5 model and its hundreds of billions of parameters (Dale, 2021). The impressive capacities of ChatGPT—which can as well write essays, find bugs in code, and pass the entrance exam of law and business schools—, and the risks and issues it raises, were echoed in statements such as the warning letter issued by technology company executives and researchers in the beginning of 2023 (@70), as well as changes in policies, such as the 2023 update of the ACM policy on authorship, which now includes explicit rules regarding "*generative AI tools and technologies, such as ChatGPT*" (@71).

The use of artificial intelligence techniques to help humans work with computer languages has followed a similar track. For several decades, the dominant rule-based approach was used to derive programs (output in some computer language) from specifications, such as logic formulas in theorem provers and constraints in declarative languages such as SQL and Prolog (Flener, 2002). Similarly, hand-crafted heuristics have been used to infer properties from code written in computer languages, such as *code smells*[27] in Stench Blossom (Murphy-Hill and Black, 2010) and ongoing refactorings in BeneFactor (Ge et al., 2012). Dedicated systems and languages, such as Semgrep and CodeQL, have even been conceived specifically for manually describing patterns to search for in code bases.

27. The term *code smell* was coined by Fowler (Fowler, 2019, ch. 3) to qualify patterns of code that are likely to cause issues in the future, and would be better off refactored.

Yet, just as for natural language processing tasks, machine learning techniques have become increasingly used to analyse and synthesise computer languages. The idea to derive programs from input-output pairs rather than specifications can be traced back at least to the work on the *programming by example* (also called *programming by demonstration*) paradigm led by Lieberman in the 1980s (Lieberman, 2001), which was subsequently applied to various domains, such as learning the structure of a piece of text or a sequence of operations. It was applied to computer languages by enabling the conception of example-directed systems for, e.g., synthesising regular expressions (Zhang et al., 2020) and generalising search-and-replace (Ni et al., 2021), as well as to synthesise code to transform data in several languages, as demonstrated in FlashFill (Gulwani, 2011) for inferring formulas in Microsoft Excel, Wrex (Drosos et al., 2020) for cleaning data in Python, and Falx (Wang et al., 2021) for visualising data in R. In addition, models trained from examples have also been used to synthesise computer languages from images to, e.g., turn sketches into imperative code that draws them (Ellis et al., 2018) and turn drawings of quantum circuits into Python code (Arawjo et al., 2022).

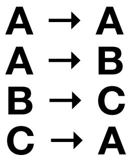
In the past few years, models primarily trained on natural language corpora and designed to solve natural language tasks have been increasingly adapted to work with computer languages too. For example, GitHub's *Copilot* (🔗72), a tool for synthesising code in several programming languages given a prompt written in a natural language such as English, is based on the Codex model (Chen et al., 2021), which was created by fine-tuning GPT-3 with a corpus of code written in various programming languages. Similarly, ChatGPT is now being used to write, explain, debug, optimise and translate code (Perkel, 2023), even though it was not specifically designed nor trained to solve programming-related tasks. At their core, these transformer models are hardly specific to computer languages. Their very large parameter sets encode a probability distribution over vectors that represent tokens (such as words) and is used to predict the next token to synthesise given a window of previous tokens as context; but they are ignorant of language-specific structures, such as syntactic and semantic rules, and have proved to be equally able to synthesise other types of media, such as images (Esser et al., 2021). Yet, they seem to currently outperform former state-of-the-art techniques to synthesise computer languages that used to embed language-specific knowledge, such as syntax-guided synthesis (Alur et al., 2018), at least in terms of generality.

2.3.3 HUMAN-COMPUTER INTERACTION PERSPECTIVE

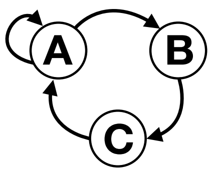
Computer languages have received substantial attention from HCI researchers, as demonstrated by the large body of technical, empirical, and theoretical contributions published in the literature. This includes creating and evaluating a great number of systems and interaction techniques for computer languages and programming, studying communities who use them, and reflecting on the forms they take and possible taxonomies for classifying them. Yet, despite the large amount of work related to computer languages in HCI, there is, to the best of my knowledge, no theory of computer languages fit for the interaction-centric approach of HCI. This is unlike the other fields that I presented above, in which theories of sign systems, human perception, logic or statistics, provide researchers with a glossary of clearly defined concepts. As a result, research in HCI often refers to computer languages as sorts of blurry monoliths, assuming more or less implicitly what is and what is not a computer language, without further questioning what is it we actually interact with when introducing, evaluating or classifying interaction techniques for computer languages.

One example of this limitation lies in the previous approaches to distinguish between so-called *textual* and *visual* programming languages. Myers (1990) proposes to distinguish between textual languages, in which symbols can only be combined along one dimension, and visual languages, in which more than one spatial dimension can be used. Moody (2009a) uses the same definition, to which he adds that the difference between textual and visual languages also lies in the fact that, according to the dual channel theory, textual and visual notations are processed by different systems in our brains. On the contrary, McGuffin and Fuhrman (2020) do not define visual languages according to what they look like, but rather according to how we interact with them: as systems in which instructions are directly manipulated, instead of being written using text input techniques. Yet, as Conversy (2014) previously demonstrated with examples, and as McLean (2011) points out, such distinctions are unsatisfactory:

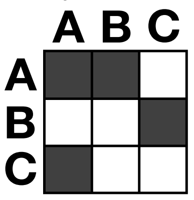
> *Research into visual languages is hampered by a definitional problem. In a well-cited taxonomy of visual programming, Myers (1990, p. 2) defines visual languages as "any system that allows the user to specify a program in a two (or more) dimensional fashion." Myers specifically excludes conventional textual languages, because "compilers or interpreters process them as long, one-dimensional streams." This exclusion is highly problematic however, as at base all a computer can do is process one-dimensional streams. Further, some textual languages such as Haskell and Python do indeed have two dimensional syntax, where vertical alignment as well as horizontal adjacency is significant; however no-one would call either language 'visual'. Worse still, for the majority of visual languages, 2D arrangement has no syntactical significance whatsoever, and is purely secondary notation. Often graphical icons are described as visual, but they too are discrete symbols, in other words textual.*
>
> — McLean (2011, ch. 5)



a. List notation.



b. Diagram notation.



c. Table notation.

Figure 2.9. Three different notations of the same graph: (a) the list of arcs between nodes; (b) nodes connected by arcs; and (c) the adjacency matrix of the graph.

Following Conversy and McLean's visions, I further argue that being rather textual or visual—no matter which of the above definitions is used—is actually a quality of a notation, rather than a quality of a language, and that as a consequence, a *single* language can be noted using *multiple* notations.

Take, for example, a simple graph language inspired by the DOT language. Let us assume that this language is primarily written and encoded as text files, as most computer languages, and formed of three types of symbols—nodes, arcs, and letters—and two composition rules stating that (1) each node is labelled by a sequence of letters and (2) each arrow connects two nodes in an asymmetric fashion. Initially, graphs described using this language would likely be edited as sequences of characters that correspond to the encoding of the graph in the computer's memory, such as using a text editor. Let us then suppose that a new editing environment for that language is released, and allows to edit the same text files using other notations, such as a box-

and-arrow diagram or a table representing the graph's adjacency matrix, as illustrated in Figure 2.9. In this situation, the same file, representing the same graph, written in the same computer language, can now be visualised and modified using different, complementary notations. The fact that they use one or two spatial dimensions, are processed by different parts of the brain, and controlled using keyboard inputs or direct manipulations, does not affect the underlying concepts that form the graph, the specifications of the symbols and rules, nor the way it is encoded in the computer's memory.[28] This thought experiment exemplifies why merely opposing textual and visual languages or notation carries little useful information. In such taxonomies, any language could be one, the other or both depending on the notation(s) provided by the environment it is edited in; leaving unclear what else constitutes a computer language besides its notation.

28. The new editor might of course rely on data structures that are different from the sequence of characters encoded in the computer's memory to create the alternative notations; but they would have to be derived from the encoding of the graph, and mapped back to it, for it to be a new editor for an existing language.

Furthermore, the notion of notation alone seems too weak to capture the vast diversity of user interfaces and interaction techniques that have been developed to interact with computer languages. When used to describe a computer language, the term *notation* usually characterises a visual sign system used to encode the symbols and the rules of the language. Yet, computer languages can also be manipulated through graphical user interfaces that say nothing about the symbols and the rules of the language. For example, using a colour picker to modify a piece of code that represents a colour, as demonstrated in Graphite (Omar et al., 2012) and Livelits (Omar et al., 2021), is hardly a representation of the numeric symbols that are used to encode the colour in Java and Hazel. Further, in paradigms such as output-directed programming (Chugh et al., 2016), in which a computer language can be modified by directly manipulating the output it generates, such as an image or a webpage, the visual aspect of the output—i.e., the notation—cannot even be described in advance, as it entirely depends on what the program generates.

While theories such as the cognitive dimensions of notations can accommodate certain variations through the concept of secondary notation to, e.g., describe visual augmentations of the code (Sulír et al., 2018), they are not appropriate to describe the interaction techniques and paradigms I just mentioned. How, then, conceive a theory of interaction that is specific enough to integrate the specificities of computer languages, yet general enough to yield a design space of interaction techniques that captures existing work and suggests directions to explore?

The recent development of a more holistic theory of programming *systems*, rather than *languages*, partly addresses this question. This theory was introduced by Jakubovic et al. (2023) in the beginning of 2023, who define a *programming system* as follows:

> *A* programming system *is an integrated and complete set of tools sufficient for creating, modifying, and executing programs. These will include notations for structuring programs and data, facilities for running and debugging programs, and interfaces for performing all of these tasks. Facilities for testing, analysis, packaging, or version control may also be present. Notations include programming languages and interfaces include text editors, but are not limited to these.*
>
> — Jakubovic et al. (2023, §1)

While previous work, such as those of Green (1989) and Bret Victor (@73), respectively stated that "*system = notation + environment*" and "*system = language + environment*", they rather focused on the right-hand part of the equation, little developing the whole that glues the summed elements together. Instead, Jakubovic et al. (2023) introduce their work by stating that "*while* programming languages *are a well-established concept, analysed and compared in a common vocabulary, no similar foundation exists for the wider range of* programming systems" and conclude that "*the academic research on programming suffers from this lack of common vocabulary*" (§1.1). They follow by developing a framework for analysing and designing programming systems holistically, in the light of multiple dimensions that they group under seven categories: interaction, notation, conceptual structure, customisability, complexity, errors and adoptability.

Although I believe this approach is promisingly holistic and already operational,[29] I also argue that it is too much centred on programming systems and textual languages to serve as a basis for this thesis. In the framework, the term *language* mainly refers to a traditional textually-encoded programming language as part of a larger system, whereas the term *notation* refers to the user interface through which the system can be programmed, be it textually or not. For example, the authors indicate that one class of systems they are interested in is made of "*software ecosystems built around a text-based programming language*" (§3). They later add that "*to speak of a programming system, we need to consider a language with, at minimum, an editor and a compiler or interpreter*" (§3.1), and that "*efforts to support programming without relying on textual code can only be called "languages" in a metaphorical sense*" (§3.3)—a claim I respectfully disagree with in this thesis. Furthermore, although the authors argue that programming languages already have a theory (§2), I argue that the theory they refer to is only a theory of their specification (as mathematical objects) and implementation (as technical objects), but not a theory of that delimits *how* we can interact with computer languages and *why* is that. As a consequence, I argue that HCI would benefit from a new theory of interaction of computer languages, inspired by the holistic approach taken by Jakubovic et al. (2023), but centred on language and interaction, rather than programming and systems.

29. It has, for example, already been used by McNutt and Chugh (2023) to analyse the Vega Editor.

3

Decomposing computer languages

In the previous chapter, we observed that computer languages are critical objects in the world of computing. They include a diverse range of languages; they are used for many purposes by users with radically different backgrounds and needs; and they are studied from a variety of research fields, each with their own perspectives. As objects primarily living in computers, they also are objects we must interact with—the very topic of human-computer interaction. Yet, there is no interaction-centric theory of computer languages that explains how we can and cannot interact with them in a particular way, and why is that.

This chapter is the first step towards filling this gap by analysing what makes a computer language according to the definition given in section 2.1 and proposing a new holistic model of computer languages. Section 3.1 motivates the need for a new theory by showing that existing theories, frameworks and methods are insufficient to explain, and not only analyse or design, our interaction with computer languages. Section 3.2 progressively derives a glossary of concepts from the definition of a computer language and a single axiom, centred around five aspects that, I argue, every computer language exhibit: conceptualisation, specification, implementation, interaction and contextualisation. Taken together, these aspects form a holistic model of computer language that builds on several theories of languages while addressing limitations discussed in the previous chapter.

3.1 MOTIVATIONS

Taken separately, the existing theories that have been used to study computer languages are, although helpful, insufficient to describe how human-centric and computer-centric considerations of computer languages depend on each other. The former are described by theories of concepts, functions, sign systems, notations and perception, whereas the latter are described by formal models of computation, syntax, semantics and artificial cognition. Previous work already acknowledged that computer-centric theories such as Turing machines (Martin et al., 2023) and formal syntax and semantics (Jakubovic et al., 2023) alone are not appropriate to describe our interaction with programming systems we actually create and use. I further argue that since human-centric theories were mostly created with signs and languages interpreted by humans alone in mind, they fail to capture critical nuances that exist with computer languages, which are meant to be interpreted both by humans and by computers. Research in computer languages from a human-computer interaction point of view suffers from this lack of reconciliation, as researchers are left to either take a view of computer language that reduces our interaction with computer languages to our interaction with natural languages, or ignore the interactive aspect altogether.

To address this issue, I argue that a theory prompt to describe and explain how we can interact with computer languages must be holistic enough to capture what matters for humans and what matters for computers in a single conceptual model. This goal contrasts with previous work that rather addressed how computer languages, programming and software is used and perceived, as shown by studies with students (Moskal et al., 2017) and artists (Li et al., 2021) as well as by diverse stances from researchers (Knuth, 1974; Dijkstra, 1977; Bergström and Blackwell, 2016; Ko, 2016; Martin et al., 2023). It also complements *frameworks*, such as cognitive dimensions of notations (Green, 1989) and technical dimensions of programming systems (Jakubovic et al., 2023), and *methods*, such as natural programming (Myers et al., 2004) and PLIERS (Coblenz et al., 2021), which help analyse and design languages and systems, but do not explain what the very nature of a computer language is nor what it tells us about how we can interact with them.

Given the limitations of existing work I just mentioned, I chose not to frame the notion of computer language according to a specific theory or conceptual framework alone. Instead, I favoured a more constructivist approach by deriving a series of deductions, using only the definition of a computer language given in the previous chapter and a single axiom—humans require concepts to reason—as a premise, from which I derive five complementary aspects that, I argue, *must* belong to the concept of computer language. Taken together, the aspects of this model form a glossary of concepts that I will rely on in the rest of this thesis.

3.2 HOLISTIC MODEL

The gist of the model I propose, which is described in this section, is the following. Computer languages, just like any language, assume the existence of concepts the language refers to, both for making sense of the language itself, e.g., to understand the purpose of a symbol, and for making sense of its intent, e.g., to interpret what an identifier means—a process I call *conceptualisation*. Then, to be turned into a system of shareable codes, we must describe the symbols of the language and the rules for combining and interpreting them in a *specification*, usually with the help of a formal notation. Since a computer can only process data encoded in the only alphabet it speaks, i.e., binary digits, codes that adhere to the specification must further be turned into an encoding that can be read from and written to the computer's memory by a computer program that adheres to the aforementioned specification, as part of its *implementation*. However, since long sequences of bits are hardly appropriate to be manipulated by humans, the encoding must therefore be mapped to some representations that support our *interaction* with computer languages. Finally, as neither of these steps exists in isolation from the world,

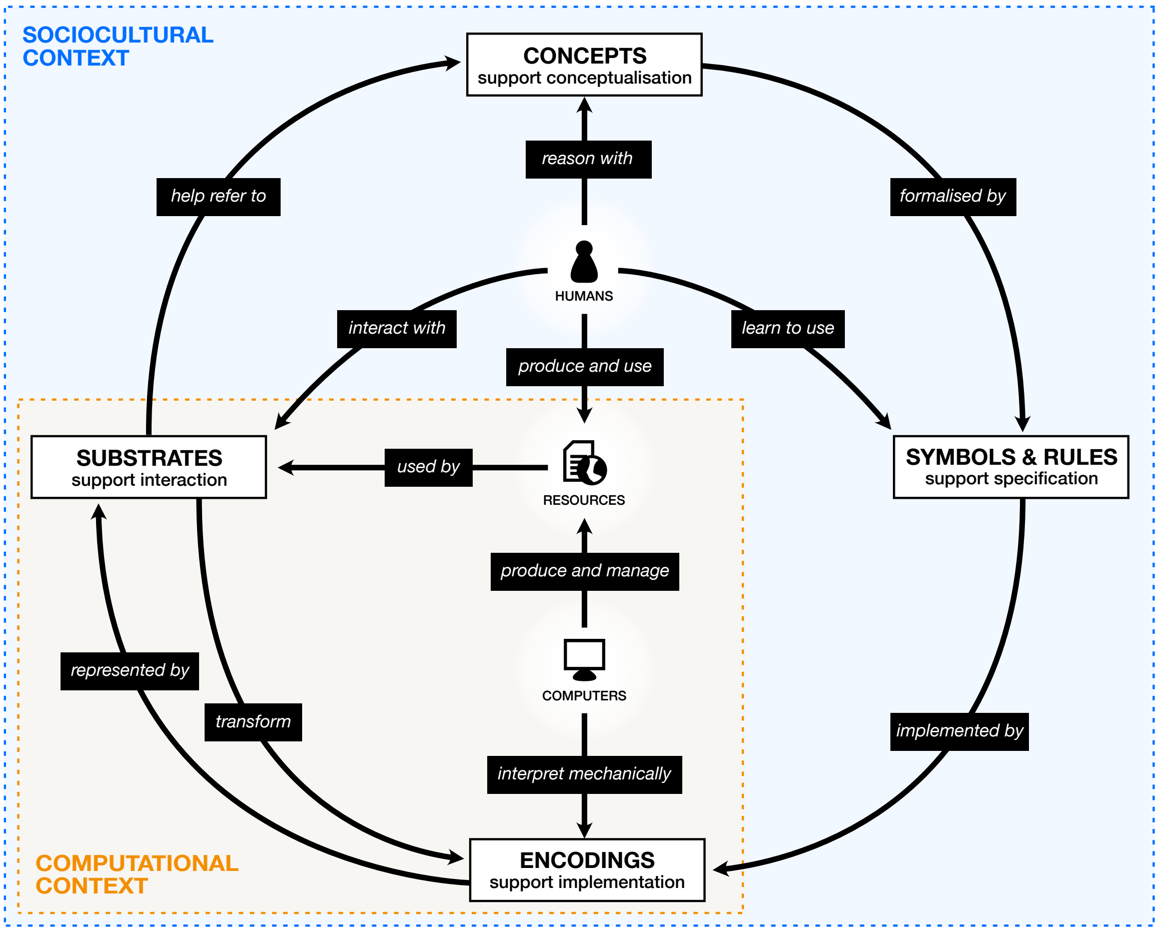


Figure 3.1. Scheme of the holistic model of computer languages introduced in this chapter. It applies to every computer language matching the definition given in the previous chapter. White boxes represent four of the five fundamental aspects of the language; the last one being represented by the two dotted boxes, which represent the contextualisation of the language. Icons represent entities using the language (humans, computers) and data they produce and manage (resources), which are not part of the language itself, but strongly connected to its existence and use. Arrows represent relationships between the different aspects of the language, the actors who use it, and the resources related to it. Labels in black boxes explain the nature of these relationships.

Aspect	Example	Purpose
Concept	Boolean values	Conceptualising the notion of a binary value of truth.
Symbols	$\langle bool \rangle$ ≔ True | False	Formalising the concept as two symbols, True and False.
Encoding	value = 0 ⇔ value is False value ≠ 0 ⇔ value is True	Mapping the symbols to numbers in the computer's memory.
Substrate	`bool show = TRUE;`	Representing the encoding so that users can interact with it.
Context	Program's output	Showing the result of setting the boolean to one value or another.

Table 3.1. Example of an analytic use of the holistic model of computer languages to distinguish between the different aspects of a computer language. The example focuses on a hypothetic computer language inspired by C, in the context where a boolean value is used as a flag to control the visibility of an element in the coded program's output.

studying computer languages calls for *contextualisation* so as to situate these languages in the computational and sociocultural contexts they live in. The resulting model, which encompasses these five aspects of computer languages, is schematised in Figure 3.1 and exemplified in Table 3.1.

3.2.1 CONCEPTUALISATION

As human beings, our intellectual thought process relies on *concepts*, akin to Plato's forms, that we mentally manipulate and relate to physical elements of the material world we live in. More specifically, I define a *concept* as a category of ideas we can distinguish from other categories of ideas, independently of whether and how it is represented. For example, the sentence "*a circle of radius r centred in point P*", the mathematical set

$$\left\{(x, y) \in \mathbb{R}^2 \text{ such that } \sqrt{(P_x - x)^2 + (P_y - y)^2} = r\right\}$$

and the drawing shown in Figure 3.2 are three different ways to refer to the same instance of the concept of *circle*.

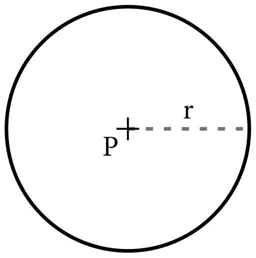


Figure 3.2. Drawing of a circle.

Concepts can be further grouped into *ontologies*, which are more abstract concepts that designate sets of concepts connected to each other in a way we deem meaningful. Ontologies can be used to contextualise data and languages, as illustrated by their use in the semantic web. For example, the ontology of mathematics includes all the mathematical concepts (also called objects, as they are necessarily abstract) and gives a particular meaning to the above descriptions of a circle. The same descriptions may be understood and analysed from a different angle through other ontologies, such as an ontology of language, which may be concerned with the structure of the sentence, or an ontology of drawings, which may be concerned with the look of the circle.

The core of the present theory lies in one fundamental axiom: just like any other language used by humans, computer languages are intellectual tools for manipulating concepts. Note that this axiom does not imply that computers cannot be used without manipulating concepts: they could be used at random, or through a form of *embodied* or *craft* knowledge, free of any thinking. Instead, this axiom suggests that as soon as one wants to reason about computer languages, i.e., using their intellect rather than (just) their emotions, they are faced with the unavoidable necessity to *conceptualise* the language. In the light of this initial assumption, I shall further distinguish between two different sorts of concepts that can be manipulated with the help of computer languages: intrinsic concepts and extrinsic concepts.

Intrinsic concepts

Intrinsic concepts are concepts that are used by the language itself, i.e., the concepts that are required to describe the purpose of its symbols and its rules. They are typically learnt along with the language, or reused from the previous knowledge of another language,[30] and form the atomic concepts we must reason with when we work with a computer language. As a rule of thumb, the intrinsic concepts of a language are the prerequisites one must understand to make sense of the language's documentation.

30. Transfering knowledge between languages is a well-known phenomenon concerning natural languages, whose application to computer languages is being increasingly studied, as reported by Hao and Glassman (2020). I further discuss how it has been applied to help programmers learn new computer languages and avoid certain pitfalls in subsection 3.2.5.

Very diverse intrinsic concepts have been used to conceive features found in computer languages. Early programming languages relied on concepts close to hardware, such as *instructions* available with a particular processor, *registers* that can be read and written, and *jumps* in memory. Higher level languages such as Fortran introduced the possibility to write *expressions* that can be evaluated using static *literals*, dynamic *variables* and mathematical *operators* to combine them. The advent of structured programming brought a fair share of new concepts, including *procedures*, *functions* and *modules* for reusing code, a well as *conditions* and *loops*. Many computer languages also make a number of data structures first-class citizen of the language, requiring to understand concepts such as *tuples*, *arrays*, *lists* and *maps*. Object-oriented programming languages often rely on *classes*, *prototypes* and *inheritance*, whereas functional programming languages may build on *polymorphism* or *algebraic data types*, and sometimes even borrow concepts such as *monads* and *functors* from category theory. Languages that offer primitives for parallel, synchronous or asynchronous operations often require to understand the concepts of *atomics*, *threads*, *locks*, *signals* or *promises*. In addition, languages with meta-programming capabilities may require to understand concepts such as *macros*, *decorators* and *templates*.

Intrinsic concepts can become part of a computer language in multiple ways. Some, like variables and operators, are reified as individual symbols. Others, like expressions and lists, are usually composed of several other symbols. Moreover, many concepts do not appear in the syntax, but rather describe general patterns or dynamic behaviours permitted by the language, as illustrated by, e.g., threads and polymorphism. Furthermore, a single language can also include seemingly different concepts that actually refer to the same parent concept. For instance, Python distinguishes between *named functions* (introduced with the `def` keyword) and *anonymous functions* (introduced with the `lambda` keyword), although both of them build on the same concept.

Extrinsic concepts

Extrinsic concepts are concepts that are expressed using the language and its intrinsic concepts without being part of the language themselves. Some extrinsic concepts group and abstract intrinsic concepts to form new concepts that help using the language itself. This was previously identified at multiples scales, such as design patterns (Gamma et al., 1995) at the program/module level, micro-patterns (Gil and Maman, 2005) at the file/class level, and nano-patterns (Gil et al., 2019) at the function/line level. For example, Gamma et al. (1995) describe the *Observer* pattern as "*defin[ing] a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*", i.e., as a relation between *objects* in which a series of *methods* is *called* when the *state* of the observed object changes.[31] Other extrinsic concepts refer to concepts completely unrelated to the language they are expressed in. For example, a collection of three numeric values in a computer language may be used as a *date*, a *colour* or a *3D position*, three extrinsic concepts that are foreign to most of the computer language they are used in.

31. I emphasised four key intrinsic concepts the extrinsic concept of observer builds onto: two that are explicitly cited (*object*, *state*), and two that are implicitly used (*method*, *call*) in the expression "*notified and updated automatically*".

Libraries are an important source of extrinsic concepts, requiring their users to familiarise to their own concepts before using them, often by reading user guides or relying on external sources of knowledge. Some include concepts commonly used in programming, such as data structures like *buffers*, *trees*, *graphs* and *data frames*, and common operations like *searching*, *sorting*, *filtering*, *mapping* and *reducing*. In addition, libraries let users of computer languages reify many domain-specific concepts, such as *paths*, *files*, *directories*, *permissions*, *addresses*, *pipes* and *sockets* in operating systems; *models*, *views*, *controllers*, *events* and *callbacks* in user interfaces; and *ciphers*, *keys* and *hashes* in cryptography.

Interestingly, some extrinsic concepts eventually become part of the intrinsic concepts of certain languages they are heavily used in. This is similar to the way concepts that are commonly used but cannot be described by a single word eventually become part of the language with the help of a neologism or a foreign word. As an example, the concept of *iterator*, which was introduced as a design pattern by Gamma et al. (1995), eventually made its way into the syntax of Java 5's *enhanced for loops* (noted `for (Element e : iterable)`) in 2005 after being commonly used by programmers (@74). Similarly, the concept of *null coalescing operator* (@75) made its way into JavaScript in 2020 to reify expressions of the form "if A is defined then A else B" as a dedicated binary operator (noted `A ?? B`). The distinction between intrinsic and extrinsic concepts in computer languages is therefore not set in stone, and one must be cautious (and distinguish between several versions of the same computer language) when stating whether a concept is intrinsic or extrinsic.

3.2.2 SPECIFICATION

While the notion of concept captures the ideas we intellectually work with in a form, it says nothing on how we can (1) refer to them in a persistent way and (2) organise and combine them to build more complex ideas. By defining a set of *symbols* and *rules* to combine and interpret them with the help of some meta-language (such as a natural and/or formal language), which themselves rely on a set of concepts forming the intrinsic concepts of the language, a *specification* addresses all these limitations. Moreover, the specification is often complemented by a sign system, which give a physicality to the symbols so that they can be communicated to others, e.g., by speaking or writing them, and taken out of our limited working memory to form larger combinations, e.g., mathematical proofs. However, and unlike natural languages, this sign system is not, strictly speaking, a mandatory part of the computer language we are defining. Instead, it shall only be seen as a bootstrapping tool used to formalise the computer language in the first place, and not to be confused with the notion of *substrate* that will be presented later.

Symbols

Symbols can be defined as the smallest entities that can be distinguished from one another, effectively forming the atomic building blocks of a computer language. Being a symbol requires to be distinguishable; but the actual form of that distinguishability is a matter of the notation, rather than the specification.

For this reason, they differ from the notion of character (such as those of the English alphabet), which are part of a particular notation. For example, a notation may assemble the latin characters `l`, `e` and `t` to form a unique symbol—noted `let`—that corresponds to a keyword for declaring a variable; but another notation may decide to use a different shape or sequence of shapes, without altering the uniqueness of the symbol they both refer to. To some extent, the notion of symbol corresponds to the notion of *token* present both in artificial models of language and lexical analysers.

In practice, computer languages are built around diverse sorts and numbers of symbols. Common categories include *identifiers* (which can often be freely defined by the user of the language), *keywords* and *operators*, but languages such as Boxer (diSessa and Abelson, 1986) and Max/MSP (Puckette, 2002) may rather include symbols called *boxes* and *patches*. Sometimes, symbols can also take a more unexpected form. For example, in Piet (@77), a so-called *esoteric programming language*, symbols are squares of different colours in a picture, as demonstrated in the *Hello, world!* program shown in Figure 3.3.

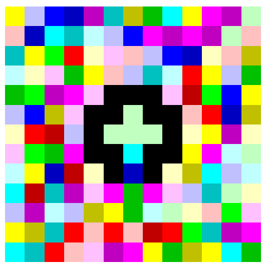


Figure 3.3. A program that prints *Hello, world!* in Piet created by Thomas Schoch (@76).

Different computer languages can feature a very different number of symbols. On one hand, certain languages aim to be minimalistic, as demonstrated by languages such as Brainfuck (@78), whose 8 symbols are sufficient to form a Turing complete programming language by distinguishing a few basic operations on the computer's memory. On the other hand, other languages are instead conceived around very extensive sets of symbols. The APL programming language (Hui and Kromberg, 2020), first implemented in the early 1960s, was initially devised by Iverson as a more concise notation for writing mathematics. To that end, Iverson created a large number of new symbols to represent more or less complex operations using very few characters, resulting in several dozens symbols, often represented using unconventional shapes that are not part of natural and mathematical languages' alphabets (Figure 3.4).

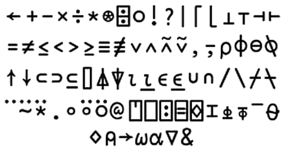


Figure 3.4. Some of the symbols used by the standard notation of the APL language. The image is adapted from the APL Wiki (@79).

Rules

In addition of determining what can be a symbol of the language, a specification must include *rules* to make sense of combinations of symbols. The first type of rules correspond to those of formal grammars and type systems, which specify which combinations can be legally part of the language. While these rules can be implemented as parsers and type checkers, one must keep in mind that this is not a necessity but a contingency of a certain choice of encoding and representation (presented later), which may not guarantee that the code is valid by construction. For example, code encoded and edited as a sequence of characters, as it is common for traditional languages such as C or Java, may include invalid subsequences of symbols, which may only become visible when the user triggers syntactic and semantic analyses of the code. Such mistakes may be prevented by construction if the computer language is manipulated in a way that guarantees the validity of the code, as demonstrated by structured editors, which may interactively reject a change if it does not result in a valid piece of code, and load and save code as a syntax tree, therefore removing the need for parsing possibly invalid sequences of symbols altogether.

The second type of rules correspond to the semantics of the language, which specifies how a human/computer should make sense of/process the structures that are correct according to the first kind of rules, such as to execute the computation they describe. Again, the actual implementations of these rules can take many forms. For example, they may appear in the compiler or the interpreter that decides how code written in a computer languages must be translated into a series of processor instructions. However, they may also be "reversed" (or conceived bidirectionally) in order to infer how to translate interactive changes to the code's output into changes to the code itself, as recently demonstrated in situations in which the input is a programming language and the output is a web page (Mayer et al., 2018), a vector-based image (Hempel et al., 2019) or a data visualisation (Perera et al., 2022).

3.2.3 IMPLEMENTATION

Although equipping a specification with a notation makes the computer language usable by humans, it is not enough to make it usable by computers too. As theorised by Turing, computers are simply machines that manipulate symbols in a predictable and programmable fashion. In practice, Turing's theory can be implemented in a myriad of fashions. The most common implementation, by far, and the one I will focus on in this thesis, is the electronic computer. Electronic computers rely on the presence or absence of voltage in different parts of a circuitry as a means to distinguish between two binary symbols, usually noted zero and one. Yet, more esoteric implementations also exist, such as using the *Magic: the Gathering* card game (Churchill et al., 2019) or a network of carnivorous plants.[32]

32. In a talk on alternative models of computation (@80), David Naccache shows that the fact *Dionaea Muscipula* "bites" only when two of their trigger hairs are touched can be exploited to build a Turing machine out of carnivorous plants and demonstrates a prototype implementation of a XOR gate.

Regardless of how they operate, all these implementations share one commonality: they all require to encode and decode data using the machine's alphabet. This rule applies to computer languages too. To implement the specification of a computer language into a computer, one must decide on an *encoding scheme* in order to write programs able to encode the language into the computer's memory, then decode and process it. Further, to persist in time and be interpreted in the correct way, that encoding must be identified as a separate piece of information by the system managing the memory, which I call a *resource*.

Encoding

Encoding a computer language can be split into two steps: the logical encoding, and the physical encoding. The former describes *what* information is encoded, and the latter describes *how* this information is encoded. Logical encodings can either describe sequences of symbols (to be interpreted in some order), or higher-level syntactic constructs from which eluded symbols can be deduced. In addition, they may also include extra information related to the language or the context it is used in. Physical encodings are usually split into two categories: text encodings, which offer an alphabet of characters that are mapped to numbers, and binary encodings, which are arbitrary mappings from logical data to bits. In the case of a digital computer, which is our focus here, encoding a computer language according to its specification therefore requires to combine a logical and a physical encoding schemes, so that symbols or syntactic constructs can be turned into sequences of bits that fit in the computer's memory, and conversely.

The simplest type of encoding scheme is to encode a sequence of symbols and nothing else. This is the scheme traditionally used by many popular programming languages, such as C, Python or JavaScript, which are encoded by strings of characters, which can be read and written as text as is. Unless each symbol can be mapped to a unique character, this scheme usually requires to tokenise the string to turn it back into a sequence of symbols with the help of a lexer. Moreover, although computer languages have long used restricted physical encoding schemes such as ASCII, which only includes 256 different characters, Unicode-compliant schemes such as UTF-8 are being increasingly used in modern computer languages. As a result, these languages often support identifiers with non-latin characters (and even emojis), as do Python 3, Swift and Julia, as well as less standard mathematical symbols, as do Gallina, Lean and F*.[33]

33. For example, `λ = 0.5` is a valid instruction in Python 3, and `{n : ℕ}` is a valid expression in Lean.

Different textual schemes have different advantages and drawbacks: some include more characters than others at the cost of either an increased memory footprint, when more bits are required to encode each character, or added complexity at runtime, when variable-length encodings (such as UTF-8) impact operations such as computing the length of a string (in terms of characters, not bytes). As an example, Hui and Kromberg (2020) report that before the inclusion of the many uncommon APL characters in the Unicode standard, different implementations of APL on different computers each used their own encoding scheme, making APL code written in one system non-compatible with other systems. Yet, they also underline that "*a variable length encoding makes array operations inefficient*", and therefore, "*many APL systems use multiple internal representations in order to efficiently support character arrays*" (Hui and Kromberg, 2020, §0.4.4).

Another approach is to encode more complex constructs, such as the syntax tree or the graph formed by the relationships between symbols. In this case, the encoding may rely on an intermediate language to describe the underlying structure. For example, the blocks that form the code written in Scratch (Resnick et al., 2009) are stored as a list of objects that refer to each other's identifiers in JSON; the languages created with JetBrains MPS (Voelter and Lisson, 2014) are encoded as syntax trees in XML by default; and the patches and wires of Pure Data's data-flow graphs are encoded with the help of a custom scheme (@81).

Furthermore, there are situations in which the computer language must be encoded along with extra information used by the environment the encoding is decoded in. For example, Pure Data encodes the position of the patches on the canvas in addition to what they are, even though it has no semantic meaning and is encoded for purely aesthetic (or usability) reasons. Similarly, while Python is traditionally encoded as sequences of symbols, systems such as Jupyter notebooks (@82) encode it along with the content of cells that may contain arbitrary data, such as text formatted in Markdown and images that were generated by executing the code. Image-based programming systems such as Smalltalk (Goldberg and Robson, 1983) and Pharo (Bergel et al., 2013) push this idea even further, as the eponymous languages used to program these systems is often stored along with the complete state of the system, including all the data used and produced by the execution of the code, in an image file which is, roughly, a dump of the whole system's memory.

Resources

Any computer system encompasses a coherent body of hardware and software that determines how to mechanically process and transform information encoded in its memory. To be manipulated in meaningful ways, this information must be organised in distinguishable units I call *resources*, which can take many forms.

Operating systems, when they are present in a computer, showcase how raw memory can be turned into different sorts of resources. Some, such as Unix-based operating systems, are conceived to treat everything as *files*, which represent units of readable and/or writable data. Others, such as Smalltalk,[34] take a radically different approach in which everything the system manipulates is an *object*, which is a resource which has a type, a state and a collection of methods, i.e., functions that can perform actions on the object and can only be triggered by sending a message to an object—including oneself.

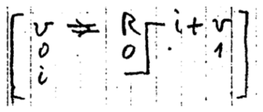
34. Although nowadays, Smalltalk is mostly executed by a virtual machine ran by another operating system, it was initially conceived as an actual operating system for Xerox computers and designed to run on bare metal.

Beyond operating systems, pieces of software with more specific application domains are systems dealing with resources too. In this case, the memory they manage and turn into resources is usually only a subset of the entire computer's memory, such as a file on the disk or a piece of working memory provided by the operating system, and the transformations they can perform are restricted to what the API exposed by the underlying operating system permits. Because these systems are more specific, they can encode, decode and transform more specific types of resources as well. For example, while the aforementioned operating systems can only treat a video as a generic file or object, a media editing system may extract a variety of finer-grained resources out of it—such as audio and video sequences, subtitles, etc.—and therefore provide specialised features that the underlying operating system has no clue about, such as editing subtitles as text or applying a filter onto the video.

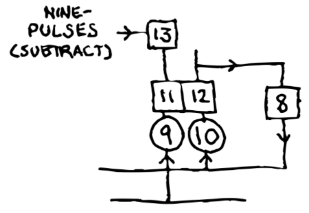
In their encoded form, computer languages are resources within computer systems too: they are nothing but pieces of memory which can be decoded and transformed, just like any other resource. For example, they can be transformed into data structures by lexers and parsers, into another computer language by a transpiler, or into machine code by a compiler, therefore forming new resources derived from the code. Moreover, a resource representing a piece of code can also be combined with other resources, such as other pieces of code and multimedia files, as do web browsers' engines when they combine HTML, CSS and JavaScript code with external resources they refer to (such as images and fonts) so as to present the unified webpage they describe together to the user.

3.2.4 INTERACTION

Although encoding symbols and rules into resources makes computer languages usable by machines, it is still not enough to make them usable in their encoded form by humans too as we never interact with languages as bits in a computer's memory. Even when switches, lights and punchcards were used to read and write computers' memory, they were still physical proxies for the presence or absence of voltage in certain parts of the electronic circuitry. In fact, Arawjo (2020) reports that early computer scientists reasoned on paper by creating notations that fit their needs rather than the machine's, citing work such as Zuse's *Plankalkül* (@83), the first high-level programming notation, and Burks' diagrams describing parts of the ENIAC computer (Burks, 1947), shown in Figure 3.5.



a. Zuse's notation.



b. Burks' notation.

Figure 3.5. Custom notations created by (a) Zuse in 1945 and (b) Burks in 1947 to work with computers and programs written for them. The images are reproduced from Arawjo (2020).

Nowadays, the representations and the techniques that we use to interact with computer languages are very diverse. Often, this interaction is mediated by a user interface displayed on a screen, manipulated with the help of a keyboard and a mouse. Yet, interaction with computer languages can also make use of other modalities. Blind and low-vision software developers have reported listening to screen readers and touching Braille displays to read computer languages (Mealin and Murphy-Hill, 2012). Virtual reality programming environments such as Cubely (Vincur et al., 2017), Ivy (Ens et al., 2017) and FlowMatic (Zhang and Oney, 2020) let users arrange and connect symbols using gestures and handheld controllers. Juxtapose (Hartmann et al., 2008) and Dynamicland (@84) demonstrate how physical objects such as motorised faders and sheets of paper can be used for interaction.[35]

35. Although Dynamicland defends a vision in which people move *away* from the traditional approach to computers and their languages by embedding them in the physical world we live in, it also showcases how objects around us can be used to program and, possibly, manipulate computer languages. Geoffrey Litt shows a glimpse of potential approaches in a blog post on this topic (@85).

A consequence of this diversity is that a holistic model of computer languages should equally capture interaction with flat notations displayed on screen, virtual objects in extended reality spaces, and tangible objects in physical environments. To that end, I build upon Beaudouin-Lafon's theory of *unified principles of interaction* (Beaudouin-Lafon, 2017), which introduces three concepts—substrates, instruments and environments—to establish a universal foundation to interaction with computers, akin to the unification of theories of mathematics under the set theory in the 20th century. Although I depart from the theory, which I deem too specific to a particular implementation requiring, e.g., systematic reactivity and shareability, I borrow and adapt the concepts of *substrates* and *environments* for the purpose of my own work.

Substrates

In order to let humans interact with a computer language in its encoded form, only considering resources and transformations between resources is not enough. Resources alone are not something we can perceive nor act upon: they need to be reified as something fit to our senses and physical abilities. Fortunately, we can equip computer systems with input and output devices that are adapted to our senses and capabilities, such as screens and projectors whose pixels can be controlled by modifying the appropriate resource, and sensors whose signal can be digitalised and written into a resource.

Using those devices, we can communicate with computer systems through a certain sensory space, formed of physical dimensions available to both the computer system and the human user, and limited by what the devices can sense and alter. In the early years of computing, this space was fairly narrow, as interaction with computers mainly occurred via mechanical switches, blinking lights, printed paper and punched cards. As the underlying technology evolved, so did the input and output devices and the span of their configuration spaces. Today, we can use screens with millions of pixels, sometimes filling up entire walls, capable of sensing fingers and pens with great precision, or placed in extended reality headsets we can wear; mice detecting fine changes in position; speakers capable of playing arbitrary sound; and numerous other devices, whose capacities for interaction are commonly researched in HCI.

In order to communicate information on resources, the computer must direct output devices to constrain the physical world in a specific way. For example, a screen must be constrained to light up only certain pixels for us to see particular shapes we interpret as text written in an alphabet we are familiar with, and a speaker must be constrained to vibrate at a certain frequency and amplitude for us to hear a particular sound we interpret as a validation or a failure. In turn, we can adapt our reaction to our interpretation of what these constraints tell us. Conversely, to take an action that may affect a resource, we must constrain the physical world, such as by pressing a key, clicking a mouse button, moving a handheld controller or pronouncing words, enough to trigger a measurable change in the sensors of the input devices.

Since we often reuse representations of digital data with similar characteristics, I propose to reify a set of constraints on input and output devices as a *substrate*, defined as follows by Beaudouin-Lafon:

> *A substrate is a digital computational medium that holds digital information, possibly created by another substrate, applies constraints and transformations to it, reacts to changes in both the information and the substrate, and generates information consumable by other substrates.*
>
> — Beaudouin-Lafon (2017, §2)

I slightly depart from this definition, as I delegate the storage, transformation and reuse of data to the aforementioned notion of resource, and only treat a substrate as a set of constraints linking a resource with an input and/or output device, whereas Beaudouin-Lafon's theory also treats resources as substrates, later distinguished as *data substrates* (Beaudouin-Lafon, 2023, §3.1). In my view, binding data to a substrate entangles a piece of digital information with a configuration of the physical world, making the two co-dependent, so that modifying one can affect the other.

Furthermore, unlike Beaudouin-Lafon, I do not make a distinction between *substrates* and *instruments*, which he defines as a particular kind of substrates "*that can operate on other substrates, even with very little knowledge about their properties and structure*" (Beaudouin-Lafon, 2017, §2). Although I admit that instruments are conceptually useful to foster interaction design (Beaudouin-Lafon, 2000) and study interaction through the lens of the *technical reasoning* theory (Renom et al., 2022, 2023), I believe that substrates are enough to the holistic model of computer languages I am describing. Just like they can reify constraints on an output device only, substrates can also reify constraints on an input device only, regardless of how that input device lets the user "operate" on another substrate. For example, a list or a grid substrate might be used to constrain a mouse by clipping the cursor's position to the indices of the list items or the grid cells, even though the screen displays no list-like or grid-like structure. This is akin to, e.g., speeding up selection with a bubble cursor (Grossman and Balakrishnan, 2005), which always targets the closest selectable item according to a Voronoi diagram that is not shown to the user.

Different substrates can include different types and numbers of constraints, resulting in different degrees of freedom. For example, a list or grid substrate only constrains the horizontal and/or vertical positions of the items it contains and has therefore many more degrees of freedom, leaving the task of constraining the rest of them—such as the content of each item or cell—to other substrates. A textual substrate is already more constrained, as it not only constrains the position of the items it contains, but also their shapes, which must belong to a particular alphabet. Depending on the situation, it may also constrain the dimensions of the characters, e.g., if the font is monospaced, prevent changes in the orientation or the texture of the characters, etc. A button or a menu item is even more constrained: their size is limited and their visual style is often imposed by the system, only leaving their static content, usually textual or iconic, unconstrained. In return, the more a substrate is constrained, the less work is left to the designer of the interactive system, which has fewer choices on how to further constrain (or not) the dimensions of the configuration space left unconstrained by the chosen substrate.

Most computer languages seem canonically associated with a single substrate, to the point where certain substrates have become a way to classify computer languages, leading to the rigid definitions of "textual languages" and "visual languages" I critiqued earlier. For example, we mainly interact with languages such as C, Python and JavaScript through textual substrates, whereas we rather use a tree substrate made of blocks for Scratch and a graph substrate made of patches for Max/MSP. Yet, in practice, we could represent the encoding of any of these languages using any of these substrates without changing the specification or the encoding of these languages. This was exemplified earlier with the four alternative substrates for the toy computer language for describing graphs presented in subsection 2.3.3: a textual substrate representing the encoding as a sequence of characters; a list substrate representing the arcs forming the graph; a graph substrate representing the nodes and arcs on a two-dimensional canvas; and a grid substrate representing the adjacency matrix of the graph. Moreover, and as we shall see later in this thesis, some substrates are only useful to interact with very specific pieces of code and would be meaningless to interact with anything else. For example, a colour picker is helpful to visualise and modify strings or triplets of numbers that represent a colour, but it is inappropriate for interacting with code representing a list of arbitrary objects or a loop.

Environments

Modern interactive computer systems are highly complex and often require to juggle several subsystems at once. This results in systems in which multiple programs and notations cohabit in real time, as noted by Blackwell et al.:

It is possible for several notations to be mixed within a single medium: a computer screen may display multiple windows, each running a different application with its own notation. Even within a window, there may be multiple notations – the main notation of the application, but also generic sub-notations such as menu bars, dialogs, etc.

— Blackwell et al. (2001, §3)

For the perceivable and actuatable spaces that input and output devices give us access to are limited, there must exist some mechanism to let substrates "reserve" or "reset" a portion of the spaces they are designed to constrain, so as to avoid collisions with constraints set by other substrates used within the same computer system.

In addition, systems for working with computer languages may deal with very diverse resources and include many different features. As a result, interacting with these systems often means interacting with different substrates at the same time. For example, a text editor for editing code may be complemented by a tree representing the structure of the coded document, a menu listing various commands the system can execute, and a preview of the output of the code. Such assemblies of related substrates may also be specialised to be adapted to a specific user, situation or device, further enriching the mix of substrates they rely on.

I propose to address these two requirements with the notion of environment. In this model, an *environment* is a concept representing a coherent assembly of substrates that are designed to fairly share the perceivable and actuatable spaces they constrain in parallel, with the goal of facilitating interaction with resources in the context of a particular activity, such as writing a webpage or programming an application. They somehow correspond to the user interface of systems designated by terms such as *code editors*, *integrated development environments* and *programming environments*. Although I explicitly mention the fair use aspect I see in environments, my definition roughly correspond to the one given by Beaudouin-Lafon (2017), who defines environments as "*an unified answer to organizing the digital environment*" that "*can be created for particular tasks and contexts of use, such as project-centric, activity-centric, or data-centric environments*" (§2).

Besides guaranteeing a fair use of input and output devices and letting users interact with multiple resources at the same time, gathering multiple substrates in a single environment also gives users of computer languages the ability to choose the substrate that works best for them for working with a single resource. For example, one user might usually prefer to edit code written in the graph language introduced in subsection 2.3.3 using the graph substrate, but occasionally switch to the textual substrate to copy-paste a piece of graph or fine-tune the position of a node. The complementarity between different substrates offered by an environment is akin to Buxton's concepts of *weak generic tools* and *strong specific tools*, as reported by McGrenere (1998). According to Buxton and McGrenere, there is a trade-off between being generic enough to work in many different situations and being specific enough to offer powerful features that would not make sense in other situations. Weak generic tools can be complemented with collections of strong specific tools (forming a *toolset*), so that users can use strong tools when they are equipped for a particular situation or default to weaker tools when they are not. This very well applies to environments for working with computer languages, which can include "weaker" generic substrates, such as text, and "stronger" specific substrates, such as colour pickers, and let users use one or the other depending on the situation.

3.2.5 CONTEXTUALISATION

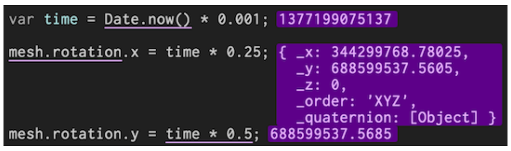
This model of computer languages may appear to be complete with conceptualisation, specification, implementation and interaction, which seem enough to describe all the aspects of computer languages that we must consider when studying how we can interact with them. Yet, none of these four aspects exists in isolation from the rest of the world. Instead, computer languages are necessarily affected by the contexts in which they are conceived, learnt and used, which are crucial to understanding how external factors affect these activities, both positively and negatively. As a consequence, I argue that contextualisation deserves to be an aspect of computer languages on its own in this conceptual model of computer languages. Depending on the nature of the factors that are being considered, I distinguish between two sorts of contexts: the computational context and the sociocultural context.

Computational context

The computational context includes all the factors that concern the computer system, regardless of their (lack of) direct impact on the humans using them. This notably includes all the other resources that are available within a given system, both local and remote, as long as the system has a means to read or write them. This consideration is especially important to account for all the design opportunities that rely on resources other than code that are available within code editing environments. As my goal here is not to cover this particular aspect in depth, I focus on two particular activities that concern computer languages and rely on other aspects of the computational context to exist: embedding information beyond encoded languages in environments, and writing code collaboratively and across multiple computers.

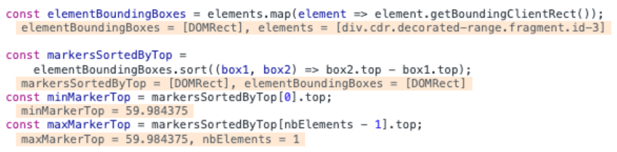
It is common for code editing environments to include information that is not embedded in the encoding itself, but collected from elsewhere in the computational context. For example, debuggers and static analysers can yield information about programs that have been generated from code written in a computer language, even though they may exist as separate resources. For example, the `gdb` debugger (@86) was initially conceived to be used as a separate program that had be controlled via a command-line interface. Yet, as long as it exists in the same computational context as the environment used to edit the code that yields the program being debugged, nothing technically prevents the controls and output of a debugger such as `gdb` from being part of the environment too. This has been demonstrated by the progressive inclusion of systems for compiling, running and debugging code from within code editors, forming so-called integrated development environments (IDEs).

To keep such software modular and favour reuse, various language-agnostic protocols have recently been developed to connect such environments with



```
var time = Date.now() * 0.001; 1377199075137

mesh.rotation.x = time * 0.25; { _x: 344299768.78025,
                                 _y: 688599537.5605,
                                 _z: 0,
                                 _order: 'XYZ',
                                 _quaternion: [Object] }
mesh.rotation.y = time * 0.5; 688599537.5685
```

a. Light Table.



```
const elementBoundingBoxes = elements.map(element => element.getBoundingClientRect());
  elementBoundingBoxes = [DOMRect], elements = [div.cdr.decorated-range.fragment.id-3]

const markersSortedByTop =
    elementBoundingBoxes.sort((box1, box2) => box2.top - box1.top);
  markersSortedByTop = [DOMRect], elementBoundingBoxes = [DOMRect]
const minMarkerTop = markersSortedByTop[0].top;
  minMarkerTop = 59.984375
const maxMarkerTop = markersSortedByTop[nbElements - 1].top;
  maxMarkerTop = 59.984375, nbElements = 1
```

b. Chromium's developer tools.

Figure 3.6. Examples of programming environments that embed runtime information within the code by connecting to an execution environment—such as a debugger—running in the same computational context. (a) In Light Table, so-called *watches* can be manually added by the programmer to continuously display the runtime value of an expression (shown on a purple background). (b) In Chromium, the current values of expressions can be inspected from within the developer tools' panel by pointing at it when a breakpoint is hit. In addition, the value of certain identifiers is automatically displayed next to them in the code (shown on an orange background) .

third-party programs available in the same computational context, even if they have not been designed to be part of a specific code editor. This notably includes the *Language Server Protocol* (LSP, @87), the *Debug Adapter Protocol* (DAP, @88) and the *Chrome DevTools Protocol* (CDP, @89), which help write and debug code while deferring the static analysis and the execution to other programs. As a result, environments for editing code are becoming more and more capable of linking code with its execution, not only through separate user interfaces, such as dedicated debugging panels for, e.g., pausing/resuming the execution and inspecting the stack, but also by augmenting substrates representing the code with this information, as in Light Table (@90) and Chromium's developer tools (Figure 3.6).

Beyond giving access to local data and services, individual computational contexts can be connected together over the network to allow multiple users to collaborate on the same piece of code. This collaboration can either be asynchronous, synchronous, or both. It relies on local resources, such as diffing and compression programs, and protocols for inter-connecting multiple computational contexts, such as TCP and WebSocket. In asynchronous collaboration, each user works on a separate encoding, only synchronising their version with the others from time to time. This is the approach used by versioning systems such as Git and Subversion, which rely on a central repository that everybody can read and write and a history of changes, whose smallest functional unit is the *commit*, which describes changes made to the code. In synchronous collaboration, all the users work on the same encoding, whose modifications must therefore be continuously shared with others using synchronisation mechanisms such as *Operational Transform* (OT) and *Conflict-free Replicated Data Types* (CRDTs). Albeit less common than the asynchronous approach, synchronous collaboration techniques are becoming more and more common in modern code editing environments, as shown by Visual Studio Code's *LiveShare* extension (@91) and JetBrain's *Code With Me* feature (@92), which even supports audio and video calls from within the code editor.

Sociocultural context

Unlike the computational context, the sociocultural context includes all the personal, social and environmental factors, which primarily affect the human beings who use computer languages, rather than the computers that process them. Such factors include, but are not limited to, training and expertise, cultural habits, spoken languages, disabilities, etc. Just like considering the computational context as whole helps understand what computers can and cannot do, considering at least some of the many factors that influence how humans create, understand and use computer languages helps understand their social impact and points to scientific inquiries and design opportunities. Again, I illustrate the impact of the sociocultural context by focusing on two examples: the effect of the proficiency in other languages, and the impact of one's training, work and habits on their expectations regarding computer languages.

Even when they are not used to interact with computers directly, natural languages have been shown to have an effect on computer languages. The widespread use of English in computer languages, in particular, not only impacts the computer languages themselves, e.g., yielding English keywords such as *if*, *while*, etc., but also their users. For example, Guo (2018) reports that even though some computer languages have been developed specifically for beginners, as discussed in subsection 2.2.4, they are often biased in favour of English-speaking users. While some code editing environments can display symbols in different natural languages, as in Scratch (Resnick et al., 2009) and Hedy (Hermans, 2020), Guo suggests that this may not suffice, and reports that non-English learners would further benefit from, e.g., "*instructional materials [. . .] without culturally-specific slang*" and "*code examples that are culturally-agnostic*" (Guo, 2018, p. 2). This bias can be put into perspective with the impact of typewriters developed by western countries during the 20th century, which forced some countries to adapt their alphabets to fulfil constraints that were designed for the English language, i.e., requiring humans to adapt to the machine, rather than the opposite (Arawjo, 2020).

Furthermore, learning and using a computer language is also impacted by the knowledge and use of other computer languages. Studies from Scholtz and Wiedenbeck (2009) and Shrestha et al. (2020) show that while the knowledge of a computer language can be leveraged to facilitate learning another computer language, this knowledge can also be a source of confusion and negatively interfere, e.g., when the same syntax has a different meaning in the two languages. This is reflected in the existence of a number of guides and systems to learn a computer language that are specifically addressed to users of another language, such as ADAPT (Fix and Wiedenbeck, 1996) for C or Pascal users willing to learn Ada and Transfer Tutor (Shrestha et al., 2018) for Python users willing to learn R. Learning a new language in terms of an old one can also improve the understanding of the latter, as reported by a number of students learning Java in C++ terms—an effect called *retroactive facilitation* by the authors of the study (Bower and McIver, 2011).

Besides knowing other languages, being trained or working in a field rather than another is also strongly affecting how computer languages are perceived and conceived. Petricek (2016) and Arawjo (2020) both underline that the fact computer science is rooted in mathematics—both theory-wise, as Turing machines and lambda calculus both originate from research in mathematical logic, and practice-wise[36]—formed a heritage of practices and expectations that lingers on nowadays. For example, Petricek highlights that in academia, "*many novel ideas that defy mathematization are left out because they are too "messy" to be considered through the predominant formal perspective*" (Petricek,

36. Before they were used for business purposes, computers were used to help with computations in domains such as ballistics. The ENIAC, one of the first large-scale digital computers, was commandited by the US army at the end of World War II and programmed by women to "*code into machine language the higher-level mathematics developed by male scientists and engineers*" (Ensmenger, 2010, p. 35) to help calculate artillery firing tables.

2016, §3.1), a claim exemplified by statements such as Leroy's, who argue that "*one of the best things that could happen to software is to be the embodiment of mathematical logic*" (Leroy, 2020, p. 28). As a consequence, many computer languages are still conceived so as to require and adhere to a certain mathematical rigour, even though many of their users are not using them to do mathematics. For this reason, and "*despite their careful design and formal foundations*", they "*address only a modest portion of modern software and only a minority of software developers*" (Shaw, 2022, p. 1).

This mathematically-rooted view of programming has been increasingly challenged in the past decades, leading to more diverse definitions of programming languages, practices and systems (Ko, 2016; Bergström and Blackwell, 2016; Shaw, 2022; Jakubovic et al., 2023). From the 1960s on, languages such as BASIC and Smalltalk were conceived with end-users in mind, rather than traditionally-trained programmers.[37] Unlike other computer languages from that time, which often required users to plan their programs well in advance, deal with hardware-related considerations, and learn about numerous language-specific features,[38] these languages were designed with fewer intrinsic concepts and so as to encourage users to create their programs progressively and interactively. As an example, Smalltalk-80 was explicitly designed so that "*there are very few new programming concepts to learn in order to understand Smalltalk*" and "*every component in the system that is accessible to the user can be presented in a meaningful way for observation and manipulation*" (Goldberg and Robson, 1983, p. viii).

37. In spite of their success, some of these visions have been mocked by certain computer scientists. For example, Dijkstra is reported to have joked that "*It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration*" (Dijkstra, 1982).

38. COBOL, for instance, includes integer, float and complex number types of multiple sizes and more than 300 keywords, resulting in a language that eventually became hard to manage, criticised for its bloated design (Conner, 1984). This can be contrasted with, e.g., the full syntax of Smalltalk being summarised in less than a double page at the beginning of the language's manual (Goldberg and Robson, 1983).

To further contrast with *modern* programming paradigms and languages developed at that time, Noble and Biddle (2002) argue in favour of a *post-modern* view of programming, defined by "*the absence of an overarching grand narrative*" and an "*eclectic tolerance in programming terms*", in which "*programs can exhibit faults in construction*". In the same line, Neff and Stark (2002) report that the many programs that are now living in an ever-changing ecosystem (such as the world wide web) should be acknowledged as somewhat brittle and *permanently beta*. Overall, while programming has long been considered as a craft too, far off from a single and rigid mathematical conception, this vision evolved from being vividly criticised by leading computer scientists such as Dijkstra (1977) to becoming a goal when designing a programming language—an activity that, according to Blackwell (2018), can also be "*craft-oriented and user-centred, rather than computationally-centred*" (§3).

4

Interacting with computer languages

In a well-known analogy, Norman (2002) compares interacting with a computer to bridging two gulfs. To communicate information to the computer, we must plan our actions and bridge the *gulf of execution*. Conversely, to let the computer communicate information to us, we must make sense of our perceptions and bridge the *gulf of interpretation*. According to the model of computer languages described in the last chapter, we only interact with computer languages that have been encoded into a computer through substrates. However, the theory remains agnostic regarding the nature of the gulfs we must bridge to interact with a computer language. What meaning do we give to the substrates we use, and what is the nature of the information we try to interpret and act upon when we interact with a computer language?

This chapter builds onto the holistic model introduced in the previous chapter to work towards a possible answer to these questions. Section 4.1 elaborates on this interactive aspect of computer languages by introducing a new taxonomy of paradigms and techniques for interacting with computer languages, which consists of four levels: graphemic interaction, morphosyntactic interaction, semantic interaction and pragmatic interaction. To define and illustrate each level, I rely on a variety of both classic and state-of-the-art examples from the literature. Section 4.2 pursues by showing that these four levels are in fact often used in combination, as do many established code editors and experimental environments, therefore yielding a design space of hybrid code editing paradigms.

4.1 LEVELS OF INTERACTION

To better understand what constitutes the interactive aspect of computer languages, I propose to consider that the function of substrates is to let us interact with every other aspect we otherwise *cannot* interact with. By giving us access to the graphemes the encodings are made of, they let us interact with the implementation of a computer language. By reconstructing the symbols and the syntactic constructs, they let us interact with the specification of a computer language. By reifying the concepts that the language builds upon and refers to, they let us interact with the conceptualisation of a computer language. Finally, by linking code to other resources that coexist in the same computa-

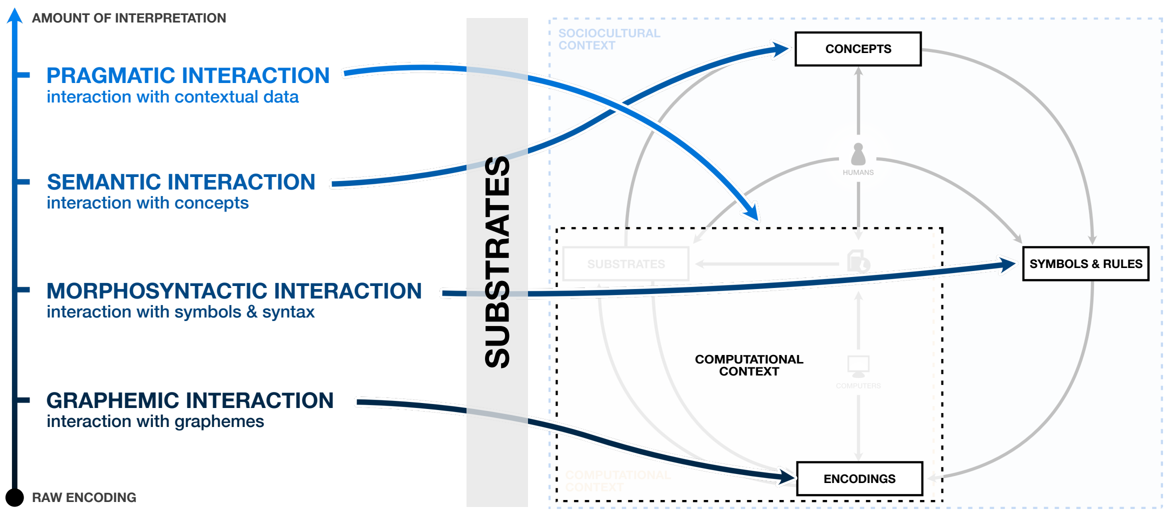


Figure 4.1. Scheme of the four levels of interaction with computer languages (left) induced by the holistic model of computer languages (right). Arrows indicate the objects of interest at each level, whose interaction must be mediated by substrates.

tional context, they let us interact with information that contextualises the use of a computer language. These four options constitute the four levels of interaction with computer languages available to us: graphemic interaction, morphosyntactic interaction, semantic interaction and pragmatic interaction (Figure 4.1).[39] In this section, I define each of these levels and illustrate how they capture the existing literature on interaction with computer languages.

39. The names and the definitions of the four levels are inspired by different hierarchical levels that sign systems and languages can be studied at in semiotics and linguistics, briefly mentioned in subsection 2.3.1.

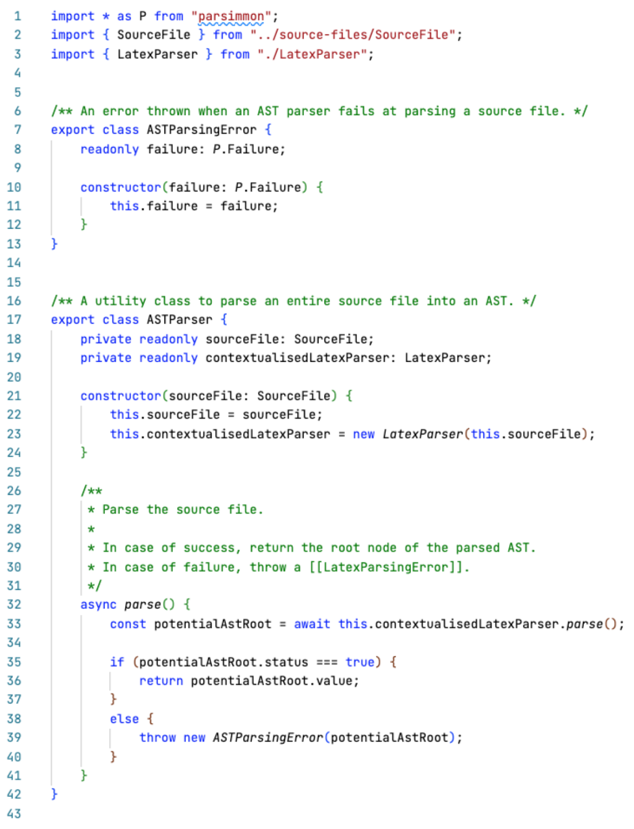
4.1.1 GRAPHEMIC INTERACTION

Graphemic interaction qualifies interaction in which the objects of interest are the *graphemes* that encode the computer language, i.e., the smallest meaningful unit of an encoding, such as characters or bytes. In case of a language encoded graphically, such as Piet (mentioned in subsection 3.2.2), in which code is encoded as a bitmap image, the graphemes may also be considered to be the pixels of the image.[40] Graphemic interaction has been the first style of interaction used to interact with computer languages and remains, by far, the most common one today.

40. In Piet, the coloured squares that form the actual symbols of the language can be larger than one pixel and are purposely named *codels* to distinguish them from the pixels themselves.

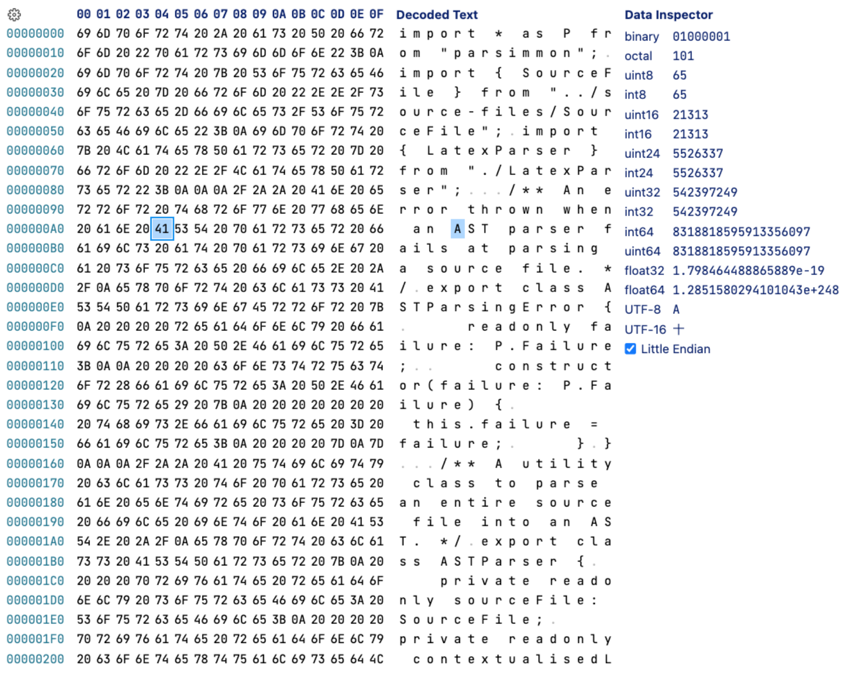
Very often, graphemes used in computer languages can be mapped to characters, i.e., forming what is commonly known as text. This is reflected by the immense and lasting popularity of text editors specialised for editing computer languages, such as Visual Studio Code, Sublime Text, Vim and Emacs, as well as those of more complex environments, such as integrated programming environments (IDEs), as in Eclipse and JetBrains' IDEs. When the graphemes are bytes, however, specialised editors (such as hexadecimal editors) can be used the edit a binary-encoded language. Such editors provide slightly adapted interfaces that visually separate bytes, align them in columns, and often help interpreting them by decoding them using a number of common encoding schemes. As an example, Figure 4.2 shows how the text and hexadecimal editors look like in Visual Studio Code.

Text editors support a wide range of techniques for navigating and manipulating text. In addition to supporting standard text editing techniques, such as copy-paste and search-and-replace, text editors for computer languages



```
import * as P from "parsimmon";
import { SourceFile } from "../source-files/SourceFile";
import { LatexParser } from "./LatexParser";


/** An error thrown when an AST parser fails at parsing a source file. */
export class ASTParsingError {
    readonly failure: P.Failure;

    constructor(failure: P.Failure) {
        this.failure = failure;
    }
}


/** A utility class to parse an entire source file into an AST. */
export class ASTParser {
    private readonly sourceFile: SourceFile;
    private readonly contextualisedLatexParser: LatexParser;

    constructor(sourceFile: SourceFile) {
        this.sourceFile = sourceFile;
        this.contextualisedLatexParser = new LatexParser(this.sourceFile);
    }

    /**
     * Parse the source file.
     *
     * In case of success, return the root node of the parsed AST.
     * In case of failure, throw a [[LatexParsingError]].
     */
    async parse() {
        const potentialAstRoot = await this.contextualisedLatexParser.parse();

        if (potentialAstRoot.status === true) {
            return potentialAstRoot.value;
        }
        else {
            throw new ASTParsingError(potentialAstRoot);
        }
    }
}

```

a. Text editor.



```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text      Data Inspector
00000000  69 6D 70 6F 72 74 20 2A 20 61 73 20 50 20 66 72  import * as P fr  binary  01000001
00000010  6F 6D 20 22 70 61 72 73 69 6D 6D 6F 6E 22 3B 0A  om "parsimmon";.  octal   101
00000020  69 6D 70 6F 72 74 20 7B 20 53 6F 75 72 63 65 46  import { SourceF  uint8   65
00000030  69 6C 65 20 7D 20 66 72 6F 6D 20 22 2E 2E 2F 73  ile } from "../s  int8    65
00000040  6F 75 72 63 65 2D 66 69 6C 65 73 2F 53 6F 75 72  ource-files/Sour  uint16  21313
00000050  63 65 46 69 6C 65 22 3B 0A 69 6D 70 6F 72 74 20  ceFile";.import   int16   21313
00000060  7B 20 4C 61 74 65 78 50 61 72 73 65 72 20 7D 20  { LatexParser }   uint24  5526337
00000070  66 72 6F 6D 20 22 2E 2F 4C 61 74 65 78 50 61 72  from "./LatexPar  int24   5526337
00000080  73 65 72 22 3B 0A 0A 0A 2F 2A 2A 20 41 6E 20 65  ser";.../** An e  uint32  542397249
00000090  72 72 6F 72 20 74 68 72 6F 77 6E 20 77 68 65 6E  rror thrown when  int32   542397249
000000A0  20 61 6E 20 41 53 54 20 70 61 72 73 65 72 20 66   an AST parser f  int64   8318818595913356097
000000B0  61 69 6C 73 20 61 74 20 70 61 72 73 69 6E 67 20  ails at parsing   uint64  8318818595913356097
000000C0  61 20 73 6F 75 72 63 65 20 66 69 6C 65 2E 20 2A  a source file. *  float32 1.798464488865889e-19
000000D0  2F 0A 65 78 70 6F 72 74 20 63 6C 61 73 73 20 41  /.export class A  float64 1.2851580294101043e+248
000000E0  53 54 50 61 72 73 69 6E 67 45 72 72 6F 72 20 7B  STParsingError {  UTF-8   A
000000F0  0A 20 20 20 20 72 65 61 64 6F 6E 6C 79 20 66 61  .    readonly fa  UTF-16  丁
00000100  69 6C 75 72 65 3A 20 50 2E 46 61 69 6C 75 72 65  ilure: P.Failure  ✓ Little Endian
00000110  3B 0A 0A 20 20 20 20 63 6F 6E 73 74 72 75 63 74  ;..    construct
00000120  6F 72 28 66 61 69 6C 75 72 65 3A 20 50 2E 46 61  or(failure: P.Fa
00000130  69 6C 75 72 65 29 20 7B 0A 20 20 20 20 20 20 20  ilure) {.
00000140  20 74 68 69 73 2E 66 61 69 6C 75 72 65 20 3D 20   this.failure =
00000150  66 61 69 6C 75 72 65 3B 0A 20 20 20 20 7D 0A 7D  failure;.    }.}
00000160  0A 0A 0A 2F 2A 2A 20 41 20 75 74 69 6C 69 74 79  .../** A utility
00000170  20 63 6C 61 73 73 20 74 6F 20 70 61 72 73 65 20   class to parse
00000180  61 6E 20 65 6E 74 69 72 65 20 73 6F 75 72 63 65  an entire source
00000190  20 66 69 6C 65 20 69 6E 74 6F 20 61 6E 20 41 53   file into an AS
000001A0  54 2E 20 2A 2F 0A 65 78 70 6F 72 74 20 63 6C 61  T. */.export cla
000001B0  73 73 20 41 53 54 50 61 72 73 65 72 20 7B 0A 20  ss ASTParser {.
000001C0  20 20 20 70 72 69 76 61 74 65 20 72 65 61 64 6F     private reado
000001D0  6E 6C 79 20 73 6F 75 72 63 65 46 69 6C 65 3A 20  nly sourceFile:
000001E0  53 6F 75 72 63 65 46 69 6C 65 3B 0A 20 20 20 20  SourceFile;.
000001F0  70 72 69 76 61 74 65 20 72 65 61 64 6F 6E 6C 79  private readonly
00000200  20 63 6F 6E 74 65 78 74 75 61 6C 69 73 65 64 4C   contextualisedL
```

b. Hexadecimal editor.

Figure 4.2. Example of two common types of editors that use substrates designed for graphemic interaction. Both editor show the same textually-encoded JavaScript file, and both are available in the Visual Studio Code editor. (a) The text editor is the default editor of Visual Studio Code. It provides many text navigation and editing features for textually-encoded computer languages, whose bytes are automatically converted to characters. (b) Microsoft's hexadecimal editor is an alternative editor that can be added to Visual Studio Code as a plugin. It displays uninterpreted bytes (shown as hexadecimal numbers), which can be individually inspected and decoded using a variety of encoding schemes by selecting them.

often include techniques that have been specifically designed for computer languages, such as keyboard shortcuts to jump to the last or the next delimiter of a pair of brackets. Code thumbnails (DeLine et al., 2006) replace traditional scrollbars with a thumbnail of the code showing the region currently shown in the text editor, which can be directly manipulated to move to reach a different part of the time. Code Bubbles (Bragdon et al., 2010) let users open multiple editors and arrange them on a canvas, which can display arbitrarily small or large portions of text files to help users read and compare multiple fragments of code in parallel. Code portals (Breckel and Tichy, 2016) achieve a similar goal by allowing users to transclude text next to the code they are editing, such as code from the same file or from a different file.

Besides parallelising views, several solutions to parallelise edits have also been proposed. Multi-cursors (Miller and Myers, 2001) let users perform the same sequence of edit operations at different locations in parallel, such as for renaming several identifiers in a limited area too narrow to use search-and-replace. Similarly, linked editing (Toomim et al., 2004) lets users declare linked blocks of code, so that when the cursor is placed in one of the blocks, other cursors are automatically added to the same position in the other blocks. CReN (Jablonski and Hou, 2007) goes one step further as it automatically detects identifiers within copy-pasted fragments of code by analysing the syntax tree and propagates changes across copies when one of them is modified.

To complement manual parallel editing, several tools have been devised to help users automate some of that work by working with patterns. As an example, regular expressions can sometimes be used in search-and-replace forms, and various systems have been created to help users write and understand them, such as RegViz (Beck et al., 2014), Regulex (@93) and Regexr (@94), or even synthesise them from positive and negative examples (Zhang et al., 2020). In addition, more advanced systems have recently been devised to let users go beyond the restricted class of patterns supported by regular expressions,. This includes Sporq (Naik et al., 2021), which infers syntactic and semantic code patterns from user-provided examples to search a code base, and reCode (Ni et al., 2021), which rather infers how to generalise a manual edit performed on one match to other matches in the context of search-and-replace.

Since all the main textual encoding schemes are widely supported everywhere, several tools that were designed for text, rather than textually-encoded computer languages, can also be used to edit code in a graphemic fashion. For example, several Unix utilities can be used for that purpose: *grep* can search patterns in the code; *diff* can compute the difference between two versions of the code; and *sed* and *awk* can search and replace textual patterns. Moreover, by writing scripts that use such utilities, users can automate common types of operations on textually-encoded languages. This idea has been reified into the concept of *patch*—a description of a transformation that must be applied to a file by some tool—which can either be written and applied manually, e.g., using *diff* and *git*, or generated automatically (Long and Rinard, 2016).

While the direct mapping of certain encodings to text may be the reason why graphemic interaction is so common, as editing plain text requires almost no preprocessing and can be done with any text editor, it does not always hold. In code editors created with JetBrain's MPS (Voelter and Lisson, 2014) and Barista (Ko and Myers, 2006), the code is encoded and stored as a syntax tree augmented with the appropriate metadata, so that each node can be displayed as an arbitrary graphical user interface. Yet, even though editing a language as text in such an environment is therefore not the default, they both attempt to mimic the experience of regular text editors. The documentation of MPS (@95) states that "*The editor gives you an illusion of editing text, which, however, has its limits. So you are slightly limited in where you can place your cursor and what you can type on that position.*" Similarly, Barista is described as a tool that "*provides designers of editors with a standard text-editing interaction technique that closely mimics that of conventional text editors, overcoming a central usability issue of previous structured code editors*" (Ko and Myers, 2006). Other possible reasons include the cultural habit of treating code as text, which originates from the fact textual notations initially stemmed from the massive use of typewriters at the time when the first programming languages were conceived (Arawjo, 2020), as well as the global use of keyboards to interact with computers, which can be very efficient when one masters them.

4.1.2 MORPHOSYNTACTIC INTERACTION

Morphosyntactic interaction qualifies interaction in which the objects of interest are the *symbols* of the computer language and the *structures* they can form when they are combined, such as two identifiers and a binary operand (three symbols) forming an expression (one structure). Accordingly, code editors that focus on symbols and structures, rather than graphemes, are often said to be *structured editors*. They often feature tree, grid or graph substrates, two of which are depicted in Figure 4.3. While morphemic and syntactic interaction could, in theory, be studied as two separate levels, the two of them are so closely related, and the former is so rarely separate from the latter, that there would be little to say about substrates and techniques for morphemic interaction alone. I therefore decided to unite them together, forming a common level of interaction.

The first tree substrates for computer languages were developed over 50 years ago to support the development of the first syntax-directed editors—a specific kind of structured editors. Once configured for a particular language's syntax, a syntax-directed editor lets programmers build a program by iteratively selecting a non-terminal symbol, which acts as a "hole" in the program, and replacing it by one of the production rules it can be replaced by according to the grammar.[41] The EMILY system (Hansen, 1972), introduced in 1970, is one of the first interactive tree substrate for computer languages. Editing code in EMILY is always a two-step process: first, a user must select a node in the tree; then, they must navigate a menu and choose either to expand it (according to one of the grammar's rules) or to replace it by a terminal symbol. It was followed by a variety of systems for creating such editors, including MENTOR (Donzeau-Gouge et al., 1980), the Cornell Program Synthesizer (Teitelbaum and Reps, 1981), GNOME and MacGnome (Miller et al., 1994), which introduced evolutions such as selecting nodes with the mouse and editing the content of a node as text (which is parsed upon validation).

41. While *holes* have a purely syntactic value in this definition, they can also be given a semantic role, as in Hazelnut (Omar et al., 2017), therefore giving a semantics to operations in structured editors and allowing partial evaluation of programs with holes, instead of merely displaying a syntax error.

Block editors constitute a modern alternative to old-fashioned *syntax-directed editors*. In a block editor, users directly manipulate and intricate *blocks* with holes, sometimes shaped so as to indicate which type of block can be used in each hole—hence their *jigsaw pieces* nickname (Figure 4.3a). Block editors were initially designed to prevent novices from making syntax errors by editing code as text in teaching environments such as Alice (Cooper, 2010) and Scratch (Resnick et al., 2009). Yet, Bau et al. (2017) report that they have been used for many other purposes that education in the past decade, ranging from writing proofs for Coq in HenBlocks (Boey and Adams, 2022), to creating 3D models in Madeup (Johnson and Bui, 2015), to specifying the protocol of user studies in TouchStone2 (Eiselmayer et al., 2019).

Just like the first structured editors, block editors evolved from being direct manipulation-based only to being hybrids between blocks and text, yielding a continuum of substrates in between text and blocks. GP (Monig et al., 2015) explores different alternatives, such as allowing users to type the name of a block at the cursor's position to display a menu suggesting matching blocks that can be inserted at that position. Pencil Code (Bau et al., 2015) displays the code as it would be displayed in a textual substrate within the blocks, and lets users hide them altogether to switch to a fully textual substrate instead. Stride (Kölling et al., 2017) takes a different path by only treating certain syntactic constructs as blocks, such as methods and conditions, and letting users edit their content as text—an approach named *frame-based editing*.

Alternative paradigms rather suggest to only interact with tree structures in a more local and on-demand fashion, often as a complement to a textual substrate, instead of making it the default interaction technique. Structural code selection (Hempel et al., 2018) highlights ranges corresponding to meaningful syntactic elements in a textual substrate to help users apply syntax-directed transformations using direct manipulation. Tiny structured editing (Hempel and Chugh, 2020) brings structured editing to custom data structures by inferring their structure from the way they are printed as text. Gradual structured editing (Moon et al., 2023) lets users manipulate the code as text while displaying the *obligations* resulting from the temporary syntax errors introduced by the current editing operation, such as missing terms, and helping the user return to a syntactically-valid piece of code.

More generally, various techniques have been proposed for addressing the challenge of interacting with tree structures. While the EMILY system only allowed to fold/unfold subtrees, select what to replace a non-terminal symbol with using menus, and type identifiers using the keyboard, later systems progressively introduced more flexibility. For example, the Cornell Program Synthesizer lets users directly type "*expressions and assignment statements*" (Teitelbaum and Reps, 1981, §1), and MacGnome lets them edit arbitrary subtrees as text. To further decrease the need for navigating the syntax tree when editing the code as text, Sandblocks (Beckmann et al., 2023a) supports text edits that cross the subtree that is being edited. For example, inserting a + in a node representing a number literal in Sandblocks will automatically suggest inserting the + sum operator (or the ++ increment operator) in the surrounding expression, instead of requiring the user to move the cursor up the tree beforehand. To parallelise edits in multiple subtrees, Forest (Voinov et al., 2022) introduces multi-cursor editing for tree substrates, along with a number of operations on syntactic cursors to navigate and transform the syntax tree.

Grid substrates are an alternative to tree substrates, in which code is arranged in a one- or two-dimensional grid, although the cells are usually represented and edited as text. The most famous application of grid substrates is the spreadsheet interface, in which code written in the underlying language is arranged in a two-dimensional grid interface. Although they are rarely envisioned from a computer language point of view, all the general-purpose office spreadsheet applications such as Microsoft Excel, Apple Numbers, LibreOffice Calc and Google Sheets are in fact built on top of computer languages similar to dataflow-oriented programming languages, in which the user expresses constraints (the formulas) between values (the cells) and leaves the responsibility of resolving them to the computer. Besides such applications, grid substrates have also been used to describe relationships between data in more tailored use cases, such as customising the look of a webpage (Litt and Jackson, 2020) or creating a simple web application (Chang and Myers, 2014), as well as to help low-vision and blind users perceive and navigate structures such as nested loops and conditions (Ehtesham-Ul-Haque et al., 2022).

While the grid interface brings new interaction opportunities (such as manipulating ranges, rows or columns all at once), they do not affect the expressivity of the underlying language. Several directions have been proposed to let users extend the underlying language with custom functions, though they mostly rely on graphemic interaction.[42] Conversely, some techniques that were initially designed for textual substrates have recently been adapted to work with grid substrates, including linked editing (Joharizadeh et al., 2020) and portals (Williams and Gordon, 2021).

42. This includes defining the function using a separate language, such as XML or Visual Basic, which must be edited as text; as well as defining the function by combining functions built into the language, e.g., using `LAMBDA` in Excel (Sarkar et al., 2022).

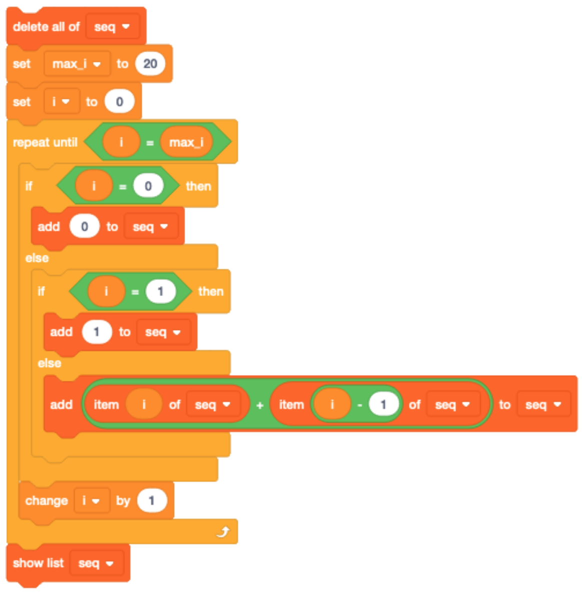
In cases in which the structure to represent does not map to a tree substrate, which reifies hierarchies, or a grid substrate, which reifies alignments, an arbitrary graph substrate can still be used to materialise more general relationships between symbols and structures. Graph substrates are often used to manipulate computer languages that describe streams, i.e., flows of values with operators to transform them, by representing sources, sinks and operators as nodes (also called patches) and connections between them as paths between nodes, similar to notations used in electronics. Just like many tree substrates, graph substrates often remove the possibility of making syntax errors altogether, making them appropriate for users who are not seasoned programmers, such as artists. As a consequence, they appear in a number of creative systems, such as Blender's node editor (@96) for configuring materials, Unreal Engine's Blueprint system (@97) for scripting games, audio-processing oriented languages such as Max/MSP and Pure Data (Figure 4.3a), and Dynamic Brushes' brush configuration system (Jacobs et al., 2018). Graph substrates have also been combined with textual substrates to form yet another sort of hybrid editors, as demonstrated by programming environments such as Nodes (@98) and Enso (@99), in which the structure of the code is represented by a graph, but the content of each node can be edited as text.

Despite their many theoretical advantages, such as preventing syntax errors and helping users visualise structures and relationships that connect symbols together, structured editors have not been very successful beyond educational use cases, as previously noted by Minör (1992) more than 30 years ago. Although some of the future work envisioned by Minör at the time, such as direct manipulation-based editors (akin to the block editors presented above), was indeed completed, the benefits of pushing expert users to switch from text editors to structured editors remain unclear. This is best exemplified by a recent article from Beckmann et al. (2023a), who observe that despite the work put into improving the usability of the Sandblocks editor, "*compared to conventional text editors*", participants of their user study "*only took on average 21% (JS), 34% (Clojure), and 95% (RegExp) longer*" to perform simple changes in short pieces of code.

4.1.3 SEMANTIC INTERACTION

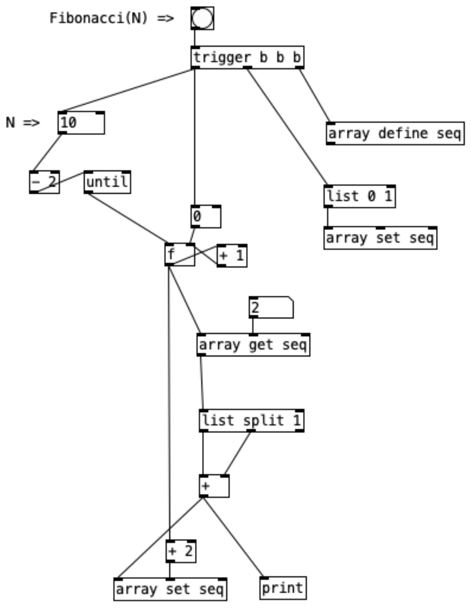
Semantic interaction qualifies interaction in which the objects of interest are the *concepts* that the code written in a computer language refers to. These concepts can be intrinsic concepts, such as when the language is used for meta-programming, or extrinsic concepts, in which case their link with the code is entirely subjective. While the look and feel of certain substrates used for semantic interaction can resemble those used for morphosyntactic interaction, the intent differs. Substrates for morphosyntactic interaction reify the *structure* of the language itself, such as the syntax tree of a piece of code, whereas those for semantic interaction reify the *meaning* we give to a piece of code. For example, consider a structure containing three numbers in a programming language. From a morphosyntactic point of view, this structure is just an assembly of three numeric values; whereas from a semantic point of view, it might represent a concept such as a RGB colour, a 3D position or a date in a calendar, depending on how we interpret it. As a consequence, there is no clear taxonomy of semantic substrates, as they can be as diverse as the notations we may use to represent and interact with the concepts we refer to using computer languages.

Common candidates for semantic substrates are standard data structures, such as lists, trees and tables—not as representations of the language's syntax, as in morphosyntactic substrates, but as opinionated interpretation of objects that do not have this specific meaning according to the language's specification. They have been demonstrated in a number of environments. INCENSE (Myers, 1983) and heterogeneous languages (Erwig and Meyer, 1995) constitute some of the earliest examples of programming systems with semantic substrates I am aware of, using box-and-arrow diagrams to represent records and pointers between them, state machines and AVL trees within a textual substrate. Similar representations are used in Vital (Hanna, 2002), an environment for editing Haskell code, which displays data structures such as lists, arrays and trees as labeled rectangles connected by wires that can be directly manipulated to transform the code. To address the lack of built-in support for representing code describing state machines in environments for editing Java code, the SwingStates library (Appert and Beaudouin-Lafon, 2006) includes a static method to visualise the machine described in the code as a means to help users analyse and debug their code, although it cannot be modified. A more interactive state machine can be found in Causette (Martin et al., 2022), a system for editing Smala code in which the user can draw a path across nodes to specify the causal order in which they should be displayed. JetBrains MPS (Voelter and Lisson, 2014) includes substrates for concepts



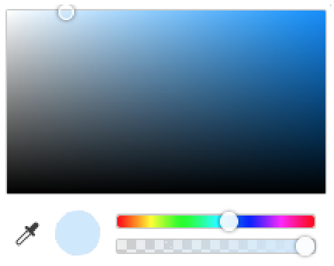
```
delete all of seq
set max_i to 20
set i to 0
repeat until i = max_i
    if i = 0 then
        add 0 to seq
    else
        if i = 1 then
            add 1 to seq
        else
            add item i of seq + item i - 1 of seq to seq
    change i by 1
show list seq
```

a. Scratch.

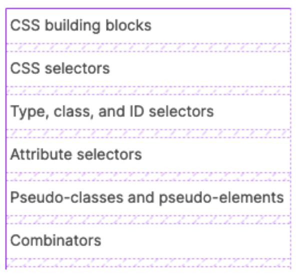


b. Pure Data.

Figure 4.3. Two implementations of the Fibonacci sequence using morphosyntactic interaction. (a) A program written in Scratch that was built using a tree substrate. Each block is drag and dropped from a panel and configured using form elements such as dropdown menus to select variables and text fields to input numbers. (b) A program written in Pure Data that was built using a graph substrate. Each patch is inserted on the canvas with a keyboard shortcut. The command it executes is written using the keyboard, and its inlets and outlets are connected to other patches using the mouse.

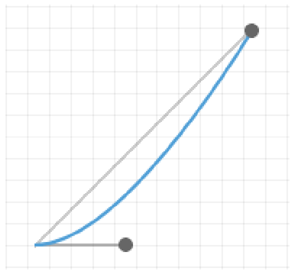


a. Colour.

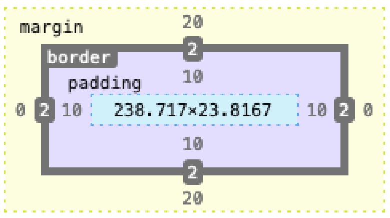


CSS building blocks
CSS selectors
Type, class, and ID selectors
Attribute selectors
Pseudo-classes and pseudo-elements
Combinators

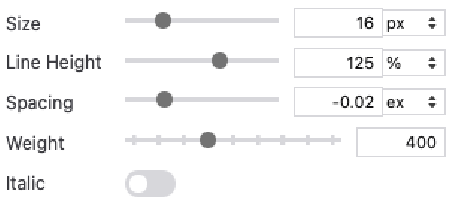
b. Grid layout.



c. Animation timing.



d. Box model.



Size 16 px
Line Height 125 %
Spacing -0.02 ex
Weight 400
Italic

e. Font.

Figure 4.4. Five semantic substrates for concepts used in CSS available in Mozilla Firefox's developer tools. (a) A colour picker representing a colour value. (b) A wireframe structure representing a grid layout within the webpage itself. (c) A curve representing the timing function of a CSS animation. (d) A set of nested rectangles representing the dimensions of the margin, border, padding and content boxes of an element. (e) A form to visualise and configure several properties of a font.

such as decision trees and state machines, either in the form of a grid or in the form of a box-and-arrow diagram, which can easily be used in code editors created with MPS, such as mbeddr for the C language (Voelter et al., 2019). Alectryon (Pit-Claudel, 2020) turns transient data structures into persistent and visual substrates, e.g., by representing a red-black tree as a coloured tree diagram, just as visual syntax for Racket (Andersen et al., 2020). mage (Kery et al., 2020), Livelits (Omar et al., 2021) and The Gamma (Petricek, 2020) let users visualise data tables using grid substrates, some of which support transforming the underlying structure, either by directly manipulating the grid or by using menus. Relationships between data points can also be turned into a substrate, typically using a 2D visualisation such as a bar plot or a scatter plot, as demonstrated in mage (Kery et al., 2020) and B2 (Wu et al., 2020).

Another common type of concept that have benefited from semantic substrates are visual properties, such as colours and dimensions. Colour pickers, in particular, are one of the most widespread semantic substrates for manipulating the concept of colour, common to many computer languages. They are available in several code editors, such as Visual Studio Code for CSS colours and IntelliJ IDEA and Eclipse with Graphite (Omar et al., 2012) for Java's `Color` objects, in which the user can invoke a colour picker when creating or modifying a piece of code representing a colour. In some cases, colour pickers are complemented by other substrates for visualising and manipulating other style properties, as in webpage inspectors, such as Poirot (Tanner et al., 2019) and those of web browsers such as Firefox and Chromium, as well as in Codelets (Oney and Brandt, 2012). For example, in Firefox's inspector, box models of DOM elements can be represented as nested rectangles representing the content, padding, border and margin boxes, with text fields allowing to edit their dimensions. Certain layouts, such as flexbox and grid layouts, can be represented by a wireframe representation of their structure.[43] Animation timing functions can be represented by curves that can be manipulated with the help of handles, and font properties can be modified with the help of a dedicated form interface. The related substrates are shown in Figure 4.4. Other examples include reification of visual properties as standard form elements. Moreover, some of these concepts can also be reified as standard form elements, such as colour and blur effects as visual switches in mage (Kery et al., 2020) and brightness and contrast as sliders in Livelits (Omar et al., 2021).

43. Such structures can also be printed on top of the webpage elements it contains, rather than in the developer tools' panel, in which case the substrate also supports a form of pragmatic interaction (introduced below).

Various other concepts that are harder to categorise have been turned into semantic substrates as well. Form descriptions can be represented as forms themselves, helping users create, modify and delete fields interactively (Andersen et al., 2020). Components forming electronic circuits described in Python are visualised as box connected by wires in IntelliJ with the help of a plugin (Lin et al., 2021). Drawings of quantum circuits are used in place of Python code to write programs for quantum computers in Notate (Arawjo et al., 2022). Value dependencies between nested loop iterations are represented by diagrams in Clint (Zinenko et al., 2015), which can be manipulated to transform the loops in order to parallelise their execution. Ownership and borrowing of Rust variables at different points in the code is represented as a diagram in RustViz (Almeida et al., 2022), which maps a number of possible scenarios to visual marks and annotations. Similarly, static call graphs of Java programs are represented as diagrams in Reacher (LaToza and Myers, 2011), which can be navigated and searched, and whose paths can be interactively expanded to show more details when desired. Game boards whose state is described using a computer language can be visually depicted, as demonstrated for, e.g., Tsuro (Andersen et al., 2020) and Conway's Game of Life (Pit-Claudel, 2020). Finally, collections of simple geometric shapes created with p5.js (@100) in JavaScript, such as rectangles and triangles, can be directly drawn and manipulated on a canvas in a modified version of the p5 editor (Mcnutt et al., 2023).

4.1.4 PRAGMATIC INTERACTION

Pragmatic interaction qualifies interaction in which the objects of interest are other artifacts living in the same computational context that are related to the computer language. This includes the runtime state of the program generated from the code, e.g., the value of a variable; the data it takes as input, e.g., a CSV file referenced in the code; or the output it produces, e.g., the rendered version of a HTML file. While some of the substrates used for semantic interaction already exploit such information, the difference lies in the fact that they do it as a technical necessity—for example, to display a colour picker when the colour is defined using variables in addition to literal numbers. In contrast, substrates used in pragmatic interaction aim to let users see and manipulate the runtime or output data for what it is. When these substrates are updated in real time, semantic interaction can further be qualified as being *live*, though this term may refer to different levels of liveness, as discussed by Tanimoto (1990, 2013). Examples of environments supporting some form of pragmatic interaction are shown in Figure 4.5.

Using a document or a scene described using a computer language has long been used in What You See Is What You Get (WYSIWYG) environments, in which users interact with a substrate representing the document as it should be viewed or printed. For example, user interface builders such as Visual Studio's XAML Designer (@101), Android Studio's Layout editor (@102) and XCode's Canvas (@103) let users create a user interface by directly manipulating standard components that compose them, such as buttons and text fields, whose invisible properties can often be edited using a complementary inspector panel containing appropriate semantic substrates. Similarly, webpage editors such as Adobe Dreamweaver (@104) allow to modify the code by interacting with the rendered webpage, without having to switch to a textual substrate. The same approach has been demonstrated for other kinds of text-centric documents, such as LaTeX documents edited with Compositor (@105).

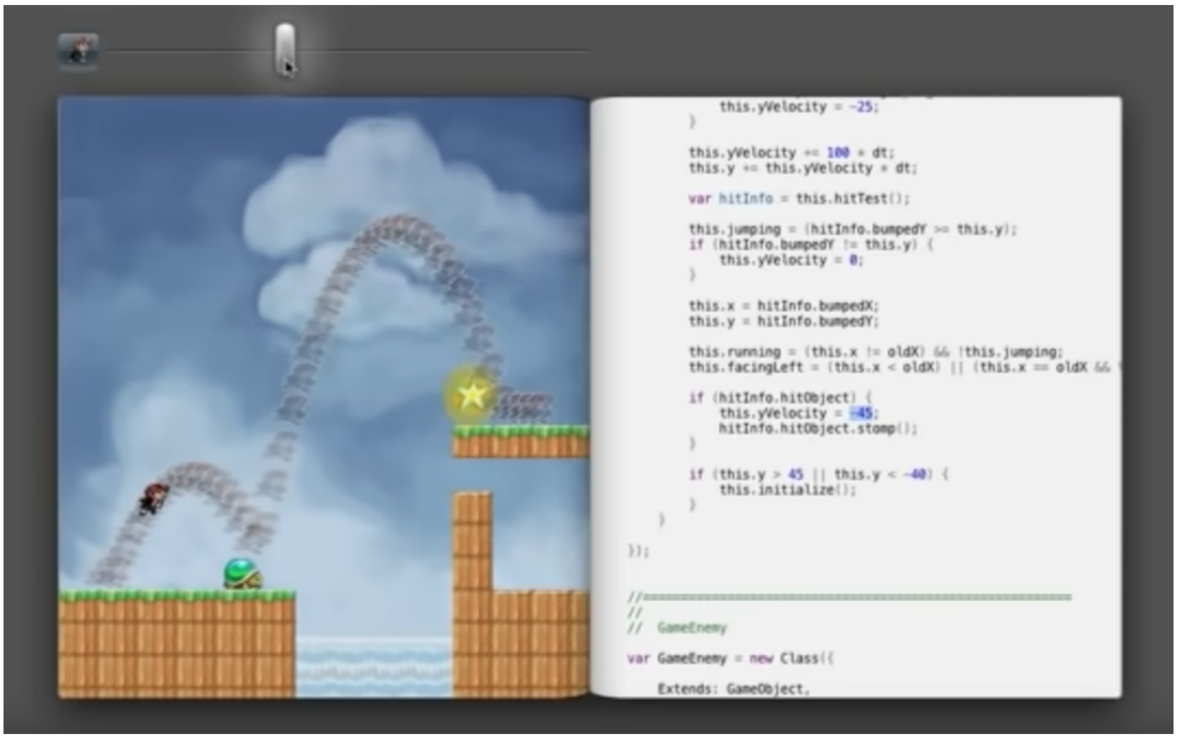
A variant of this approach consists in interacting with a document that is not described within the code itself, but the result of its execution, possibly parametrised by other input data. This kind of environment is less common, as mapping changes in a programmable output back to changes in the code that generated it can be challenging. In his *Inventing on Principle* (@106) talk, Bret Victor showed some possible applications of this idea. In particular, he demonstrated two environments in which the output of JavaScript code is instrumented to be interactive: one that draws a tree, and one that defines a platform game. In the first environment, each pixel of the drawing can be individually pointed at to highlight the line of code containing the instruction that drew it, and conversely. In the second environment, sequences of interaction with the game's character (such as key presses to make it move and jump) can be saved, previewed and replayed, in such a way that modifying the code immediately updates the preview in the game, therefore showing, e.g., how changing a parameter affects the path of the character, how far it jumps, where it lands, etc. However, although these outputs are indeed the results of a program, they are still domain-specific: each of Victor's environment is specifically crafted for a particular situation and unlikely to work in other contexts, even if the computer language stays the same.

This interaction paradigm was later theorised by Chugh as *output-directed programming* (Chugh, 2016).[44] According to Chugh, systems implementing this paradigm should have three properties: the user should be able to directly manipulate the output of the code (*live synchronisation*); such manipulation should be automatically reified in the code with the help of program synthesis (*synthesis*); and the user should remain able to edit the code directly to perform changes that cannot be initiated from the output alone, temporarily breaking the synchronisation with the output if need be (*ad hoc synchronisation*). Output-directed programming has been successfully demonstrated in the Sketch-n-Sketch system, both for HTML code rendered as a webpage (Mayer et al., 2018) and SVG code rendered as an image (Hempel et al., 2019).

44. It was initially introduced under the term *prodirect manipulation*, but later switched to *output-directed programming*. I only use the latter in this thesis.

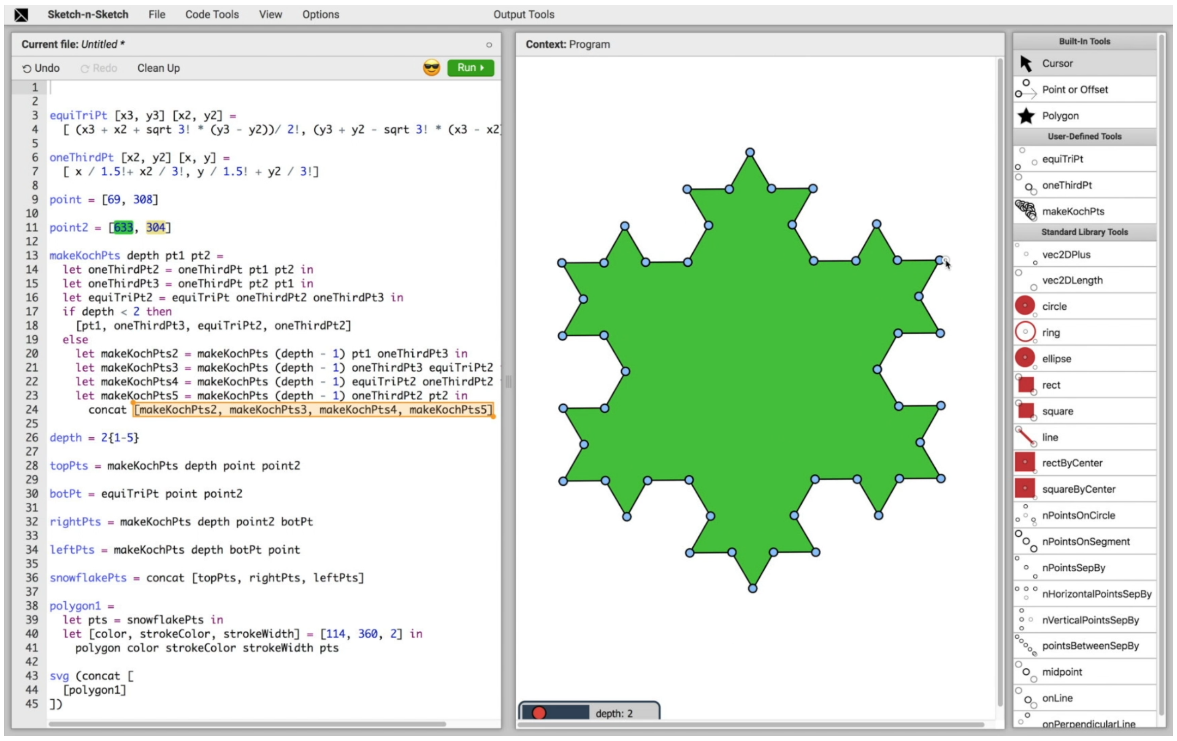
A similar approach was taken in Transmorphic (Schreiber et al., 2017), in which the code describing user interface elements can be modified by directly manipulating the latter—this time with the help of bidirectional lenses (Foster et al., 2007), which explicitly describe how to backpropagate a change in the runtime state. More generally, implementing output-directed editing systems requires to retain *provenance* information: given a piece of output (a pixel, an element in a webpage, etc.), there must be a way to determine which pieces of code generated it, and, if need be, how that generation occurred. The concept of provenance, which was initially introduced for databases (Cheney et al., 2009) before being applied to computer languages—such as general-purpose programming languages (Acar et al., 2013)—has also found applications in the field of data visualisation, in which provenance has been described as a means for creating interactive data visualisations (Psallidas and Wu, 2018) and used to help implement brushing and linking implicitly (Perera et al., 2022).

In addition to linking the code with an output that is already visible, pragmatic interaction can also reify the internal state of the program resulting from the interpretation of a computer language (or the execution of a program compiled from the code) according to the language's specification, as a means to communicate that information—which is normally hidden—to the user of the computer language. This type of pragmatic interaction is particularly common in experimental debugging environments. The Online Python Tutor (Guo, 2013) helps analyse what happens in the computer's memory during the execution of a Python program, step-by-step, by displaying the heap and the stack as diagrams. Omnicode (Kang and Guo, 2017) goes one step further by continuously running Python code and updating



```
        this.yVelocity = -25;
    }

    this.yVelocity += 100 * dt;
    this.y += this.yVelocity * dt;

    var hitInfo = this.hitTest();

    this.jumping = (hitInfo.bumpedY >= this.y);
    if (hitInfo.bumpedY != this.y) {
        this.yVelocity = 0;
    }

    this.x = hitInfo.bumpedX;
    this.y = hitInfo.bumpedY;

    this.running = (this.x != oldX) && !this.jumping;
    this.facingLeft = (this.x < oldX) || (this.x == oldX &&

    if (hitInfo.hitObject) {
        this.yVelocity = -45;
        hitInfo.hitObject.stomp();
    }

    if (this.y > 45 || this.y < -40) {
        this.initialize();
    }
  }

});


//==========================================================
//
//  GameEnemy

var GameEnemy = new Class({

    Extends: GameObject,
```

a. Platform game editor.



```

equiTriPt [x3, y3] [x2, y2] =
  [ (x3 + x2 + sqrt 3! * (y3 - y2))/ 2!, (y3 + y2 - sqrt 3! * (x3 - x2

oneThirdPt [x2, y2] [x, y] =
  [ x / 1.5!+ x2 / 3!, y / 1.5! + y2 / 3!]

point = [69, 308]

point2 = [633, 304]

makeKochPts depth pt1 pt2 =
  let oneThirdPt2 = oneThirdPt pt1 pt2 in
  let oneThirdPt3 = oneThirdPt pt2 pt1 in
  let equiTriPt2 = equiTriPt oneThirdPt2 oneThirdPt3 in
  if depth < 2 then
    [pt1, oneThirdPt3, equiTriPt2, oneThirdPt2]
  else
    let makeKochPts2 = makeKochPts (depth - 1) pt1 oneThirdPt3 in
    let makeKochPts3 = makeKochPts (depth - 1) oneThirdPt3 equiTriPt2
    let makeKochPts4 = makeKochPts (depth - 1) equiTriPt2 oneThirdPt2
    let makeKochPts5 = makeKochPts (depth - 1) oneThirdPt2 pt2 in
      concat [makeKochPts2, makeKochPts3, makeKochPts4, makeKochPts5]

depth = 2{1-5}

topPts = makeKochPts depth point point2

botPt = equiTriPt point point2

rightPts = makeKochPts depth point2 botPt

leftPts = makeKochPts depth botPt point

snowflakePts = concat [topPts, rightPts, leftPts]

polygon1 =
  let pts = snowflakePts in
  let [color, strokeColor, strokeWidth] = [114, 360, 2] in
    polygon color strokeColor strokeWidth pts

svg (concat [
  [polygon1]
])
```

b. Sketch-n-Sketch.

Figure 4.5. Two live programming environments supporting a form of pragmatic interaction with their respective computer languages. (a) A code editing environment presented by Victor in his *Inventing on Principles* talk. The user can record the execution, then pause and rewind it using the slider displayed at the top. Furthermore, by clicking the button at the left of the slider, the trail of positions taken by the character is shown and updated in real time if some code that affects it is modified. (b) A bidirectional programming environment called Sketch-n-Sketch (Hempel et al., 2019). The user can modify the SVG picture displayed on the right by editing the code as text on the left, as well as modify the code by interacting with the picture on the right using direct manipulation and tools, as in traditional vector graphic editing software.

user-defined plots showing relationships between any two variables across execution steps. Light Table's probes, previously shown in Figure 3.6, support inspecting the runtime value of expressions as text directly within a textual substrate. Poker (Descheemaeker et al., 2021) demonstrates how probes can be made configurable and displayed differently in a stream-oriented language by letting users decide what stream they would like to observe, and how they would like to display the values, e.g., by showing a gauge instead of a raw number. Projection boxes (Lerner, 2020b) work at a different scale, by displaying the evolution of the values taken by variables in the local context next to the code, e.g., in the body of a Python function, an approach named local live programming. It was later extended with loop seeds (Lerner, 2020a), which let users specify example values for variables and watch how each iteration of a loop transforms them. Log-it (Jiang et al., 2023) turns log messages into streams of events which can be read per-stream (instead of in a single aggregated list, as in terminals) and lets users visualise them graphically, e.g., using a bar plot to compare successive values, and in context, e.g., by attaching a stream to an element on a webpage. CrossCode (Hayatpur et al., 2023) lets users interactively explore the execution of JavaScript code by replacing abstract identifiers by concrete values and drawing the path taken at runtime in conditions and loops while highlighting how each step affects data structures such as lists, e.g., by showing how elements are reorganised when the list is permuted. Skyline (Yu et al., 2020) and Glinda (DeLine, 2021) show live information about the training phase of machine learning models, mainly in the form of charts; some of which the user can interact with to modify parameters in the code.

Besides being made visible to the user, runtime information can also be used to write and transform code, similar to what output-directed programming offers compared to regular live programming, in which the output of the code is updated in real time but remains static. One way to implement this approach is to directly use runtime data as building blocks for writing code. This idea was demonstrated by Subtext (Edwards, 2005), a programming environment and a language in which the code and the runtime make one, as all the data the code manipulates must exist in the environment, as in Smalltalk, Pharo, or most spreadsheet applications. As such, writing code in Subtext is equivalent to creating new data by directly manipulating and transforming existing pieces of data. In a similar fashion, Maniposynth (Hempel and Chugh, 2022) lets users write OCaml code by directly manipulating functions and runtime values, which can be dragged and dropped onto one-another to create more complex expressions. These expressions are then reified as `let` expressions in the code, which appear in an adjacent textual substrate that is updated in real time (and can also be edited as text, as in traditional OCaml programming).

A slightly different approach is to let the user help the system derive code from runtime information in a more interactive manner, by letting users exploit runtime values to provide positive or negative examples to the system. This is the approach taken by programming-by-example systems, which have been developed since the 1980s (Lieberman, 2001). Among recent work on this topic, SnipPy (Ferdowsifard et al., 2020) and LooPy (Ferdowsifard et al., 2021) build on top of the aforementioned projection boxes to implement *small-step live programming-by-example*, an approach in which the user can not only preview the runtime value of variables that have been specified in the code or that can be computed, but also input examples of values for variables that still have to be defined and let the system synthesise their definition to match user-provided examples. For example, if the system displays that the runtime value of the variable `countries` in the current scope is the list `["France", "Spain", "Italy"]` and the user tells the system the variable `abbr_countries` should contain the list `["FR", "SP", "IT"]`, the system may suggest assigning an expression to `abbr_countries` which, e.g., maps `countries` by extracting the first two characters of each item and making them uppercase.

The rapid rise of conversational agents in code editing environments, supported by systems such as GitHub Copilot and ChatGPT, may eventually result in the development of increasingly collaborative code synthesis techniques. For example, in a system that continuously executes the code, yielding a live output, the user may write most of the code by telling the system which high-level changes they would like to see, according to what they observe and what their goal is; and the system may attempt to synthesise code performing the requested changes, possibly by asking extra information to the user if need be. Currently, code editors only seem to, at best, integrate chatbots able to exploit knowledge about the entire code base, as do GitHub's Copilot Chat (@107) and Cursor (@108). Yet, recent experiments with ChatGPT by Geoffrey Litt (@109) and Philip Guo (@110), who respectively created a web application and a browser extension by asking ChatGPT to write code, trying out the code manually, and reporting on successes or issues, demonstrate the potential of exploiting not just static data, such as the local code base, but dynamic data too, such as runtime errors that must be fixed.

4.2 CROSS-LEVEL INTERACTION

As demonstrated in the previous section, we can interact with computer languages at different levels of interaction, each level reflecting a different aspect of the language we want to interact with. Yet, in practice, different levels are mixed together in a single code editing environment, sometimes qualified as *hybrid* in the literature. This section presents different ways in which levels of interaction can be combined to interact with a single computer language by classing them into two broad categories: combining levels within one substrate and combining levels across multiple substrates.

4.2.1 MULTIPLE LEVELS WITHIN A SINGLE SUBSTRATE

A first approach is to extend a substrate primarily intended for a certain level of interaction in a way that conveys information about or supports interaction with objects of interest that correspond to another level of interaction. One the most widespread application of this approach is to augment textual substrates to support interaction beyond the graphemic level of interaction they are mostly used for. It often relies on visual channels left unconstrained by the shape and spatial organisation of the characters, such as the colour of the text, corresponding to the cognitive dimension Green (1989) calls *secondary* *notation* and that the literature and the documentation of text editors often call *decorations* or *augmentations*.[45]

45. The documentation of code editor such as Visual Studio Code (@111) and CodeMirror (@112) use the term *decorations*, whereas Sulír et al. (2018) use the term *augmentations*.

In code editing environments, textual substrates meant to represent code are very often extended so as to convey morphosyntactic information, by helping the user distinguish tokens of different syntactic nature and identifiers of different types—two techniques respectively called *syntactic highlighting* and *semantic highlighting*. It has also been used to connect graphemic and pragmatic interaction—such as by blurring the text or animating the colour of its background—to convey information about the active value in a sequence of numbers in a live coding environment (Roberts et al., 2015). Other forms of extensions have also been used, including adding text and icons in the margin (Lieber et al., 2014) and small-scale data visualisations in between characters (Hoffswell et al., 2018) to convey information related to the execution of the code. In addition, a few user studies suggest that this type of visual augmentation can help users understand code (Rambally, 1986; Asenov et al., 2016), in line with arguments made by Conversy (2014) on exploiting visual channels in code editing environments so as to avoid redundancy.

Other substrates than text are also concerned by this approach, as already suggested in subsection 4.1.2, in which I report that several environments that make use of a tree, grid or graph substrate to let users interact with code in a morphosyntactic manner also rely on textual substrates and graphemic interaction to some extent. For example, MacGnome (Goldenson et al., 1992), GP (Monig et al., 2015) and Stride (Kölling et al., 2017) all use tree substrates to present code to the user while letting them type code located within nodes as text. Hybridisation can also concern other levels, such as morphosyntactic interaction and semantic interaction, as suggested by Vasek (2012), who proposes to convey semantic information—such as types representing very specific concepts—by introducing new shapes for blocks and holes in order to visually convey this information to the user.

This approach, however, is limited by the number of information channels that can be exploited, in accordance with the constraints imposed by the chosen substrate. For example, the shapes of the characters and the positions of the nodes can hardly be used for other purposes than the one they were intended for in textual and tree substrates, as they respectively convey information about an order and a hierarchy, properties that are essential for users to readily use these substrates in ways they are familiar with. Moreover, even when a visual channel is free, it might be used in idiosyncratic ways that the system is unable to understand. For this reason, the position of nodes in graph substrates is rarely given a semantic purpose in the computer language it represents, even though some users may decide to put nodes at particular positions in order to store information they—and only they—know how to encode and decode. For this reason, relying on a single substrate is sometimes not enough to combine two or more levels of interaction in a meaningful, usable way.

4.2.2 MULTIPLE LEVELS ACROSS MULTIPLE SUBSTRATES

In response to the limitation I just mentioned, a second approach consists in multiplexing levels of interaction across several substrates that can be used in parallel by the user, who is free to switch between them. Unlike extending a single substrate, this approach has the benefit of presenting information and processing input in complementary ways, though it also comes with the risk of spreading information across a larger perceptual space and requiring users to switch their attention focus and learn how to use more diverse representations.

In some code editing environments, code can be edited in a text editor *and* in a tree or graph of nodes, located in two separate panels, i.e., without combining the two into a single substrate. For example, in an attempt to help students learn how to program switch from fully morphosyntactic view of the code, represented as a tree of blocks similar to Scratch (Resnick et al., 2009), the Alice system (Cooper, 2010) also let users display the code as text, in a read-only "*Java-like*" mode in which students "*would not be typing code or [have] to deal with open and close brackets, and semicolons*", but would "*at least be seeing code containing these syntactic structures*" (Cooper, 2010, §2.1). Other systems, such as Enso (@99), Nodes (@98) and natto (@113), go one step beyond by offering their users a graph substrate to work with the high-level organisation of the code, which is split into smaller chunks or modules that are meant to be edited as text individually through a dedicated text editing interface.

Another common combination can be found in environments that display code on one side of the screen, and the output generated by interpreting the code on the other side, with more or less links between the two. The simpler approach consists in not making any link and letting the user write and read code at the graphemic or morphosyntactic level only, and observe the effects of its execution at the pragmatic level only. It is often used in complement to a text editor for document description languages such as HTML, Markdown or LaTeX and live coding systems such as Hydra (@114), in which the output document or the visual animation generated by interpreting the code is displayed in a separate panel or window. A slightly more evolved variant consists in linking parts of the text with parts of the output, as do SyncTeX (Laurens, 2008) for LaTeX, which links the text to the generated PDF, and the tree view of the DOM in the developer tools of all the major web browsers today, which links tree nodes to regions of the rendered webpage. Finally, the most advanced variants not only relate regions of the code with regions of the output they generate, but also let users modify the code by interacting with its output in an output-directed programming fashion. Besides in Sketch-n-Sketch (Mayer et al., 2018; Hempel et al., 2019), this approach has also been demonstrated in CAD software, in which the code describing a 3D object can be edited both in a text editor and by directly manipulating parts of the rendered object in a separate 3D view (Mathur et al., 2020; Cascaval et al., 2022; Gonzalez et al., 2023).

Whether they are supported by a single substrate or by a collection of substrates, combinations of multiple levels of interaction are becoming more and more common, beyond traditional techniques such as syntax highlighting and academic prototypes, as shown by hybrid environments emerging on the professional market such as Enso (@99) and JetBrains MPS (Voelter and

Lisson, 2014). Yet, although this chapter complements the holistic model of computer languages, which explains *why* we must interact with computer languages through one or more substrates, by explaining *what* kind of information these substrates help us perceive and manipulate, it still does not explain *how* interactive systems let us interact with encodings and other resources through interaction, a matter I cover in the next chapter.

5

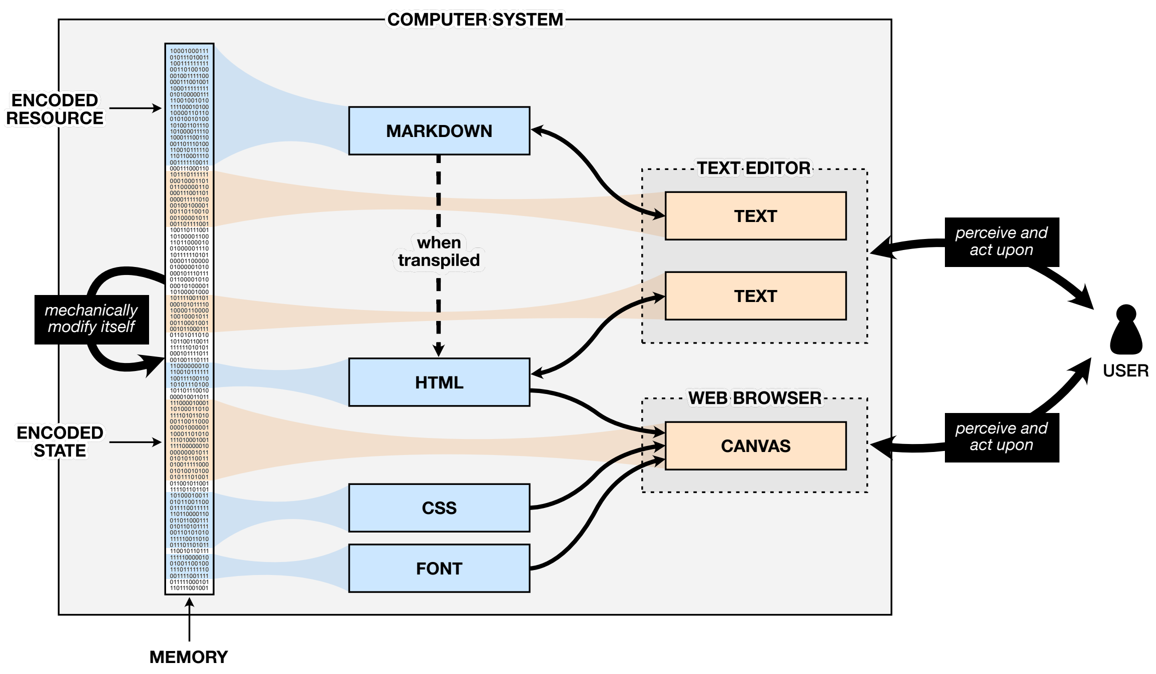
Projecting computer languages

By decomposing computer languages into five different aspects, chapter 3 highlighted the distinction between their existence as resources in a computer's memory, which the machine can process but the human cannot even perceive, and the reification of such resources as substrates, which are opinionated views of these resources that humans can interact with. Then, by identifying the different levels of interaction with computer languages substrates give us access to, chapter 4 showed that we can interact with a piece of code in different ways, using either a single substrate or multiple substrates. However, I have yet to discuss the nature and the implementation of the links that entangle resources and substrates together, as they are fundamental to understand what has been done so far and what is yet to be explored in terms of how we implement techniques for interacting with computer languages.

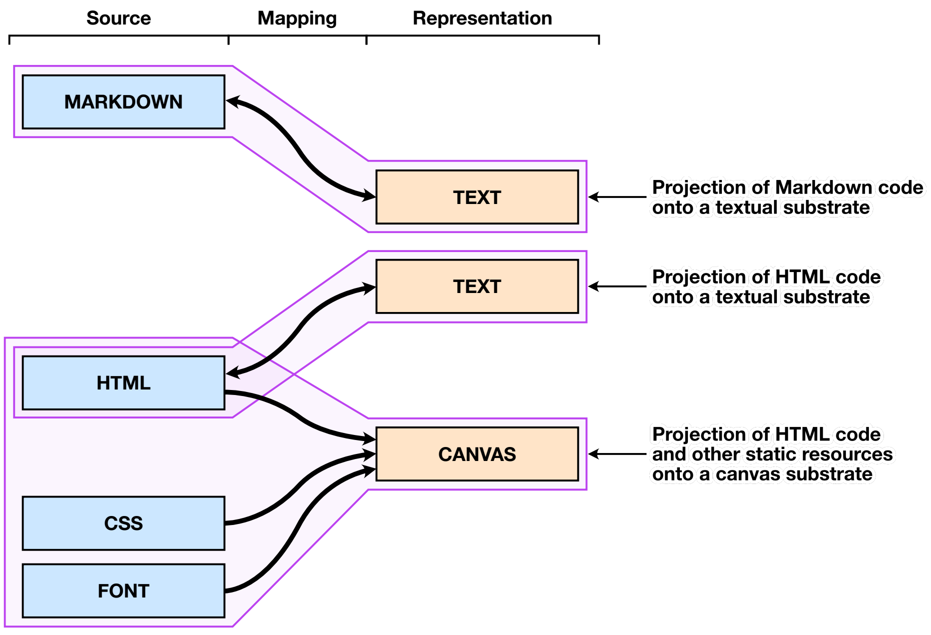
This chapter bridges this gap by focusing on the concept of *projection*, which is the action of mapping resources onto a substrate. As such, it completes chapters 3 and 4, which respectively explain *why* we must interact with computer languages through substrates and *what* these substrates let us do, by explaining *how* they work. Section 5.1 defines a series of concepts I use to describe computer systems—their resources, their dependencies and the dependency graph they form—and projections for making them interactive—their source, their representation and their mapping. Section 5.2 presents seven properties of projections, which help progress towards a design space of projections to classify existing ones and explore novel ideas in a systematic fashion. Section 5.3 reviews the evolution of implementation strategies for projecting computer languages over the last seven decades, distinguishing between two major approaches, uniform projection and protean projection. Section 5.4 concludes by presenting the subspace of the design space of projections I focus on in the rest of this work and explaining how it fits in the current research agenda on projecting computer languages.

5.1 DEFINITIONS

This section defines the concept of *projection*, to which I give a central role in our interaction with computer systems in general, and in our interaction with computer languages in particular. To that end, I first define a computer system



a. Scheme of the entire interactive system.


Source
Mapping
Representation
MARKDOWN
TEXT
Projection of Markdown code onto a textual substrate
TEXT
Projection of HTML code onto a textual substrate
HTML
CANVAS
Projection of HTML code and other static resources onto a canvas substrate
CSS
FONT

b. Focus on the system's projections.

Figure 5.1. Analysis of a hypothetical system for authoring webpages using the glossary defined in this section. Pages can be written in Markdown and transpiled into HTML code, which can both be edited as text in a text editor. The resulting HTML code can be combined with other resources, such as CSS code and font files, resulting in a webpage that can be displayed in a web browser. (a) Scheme of the entire system. The graph shown in the middle of the scheme is the dependency graph of the system. Nodes in blue are resources; nodes in orange are substrates; and dotted areas are environments that group substrates together. Plain arrows represent dependencies that are automatically enforced by the system, whereas dotted arrows represent dependencies that are enforced only when requested. Thick arrows represent the only two ways the system's memory can be modified: either by the system itself, according to instructions stored in its memory (assuming the system is implemented using a Von Neumann architecture), such as when transpiling Markdown code into HTML code; or by interacting with the system through substrates, such as when pressing a key on a keyboard in a text editing environment. (b) Focus on three projections available in the system. Each projection, shown as a purple area (a subgraph of the dependency graph), is composed of a *source* (the projected resources, in blue), a representation (the substrate they are projected onto, in orange) and a *mapping* (the dependencies between the source and the representation, represented by black arrows).

in terms of three concepts that are fundamental to this theory: resources, dependencies and dependency graphs. I then apply these concepts to the particular case of interactive computer systems, leading me to define the notion of projection by defining the three key parts that compose them: the source, the representation and the mapping. Figure 5.1 applies this glossary to describe to a hypothetical system for authoring webpages, with a focus on three of the system's projections.

5.1.1 COMPUTER SYSTEM

To properly define the process by which we interact with computer systems, I must start by defining the components that, I argue, are fundamental constituents of all computer systems, interactive or not. In the definition I choose to present here, the two fundamental constituents of a computer system are its *resources* and the *dependencies* between them which, put together, form a *dependency graph* that functionally defines a computer system.

Resource

> A *resource* is a unit of information managed by a computer system that can be encoded in and decoded from the computer's memory according to some encoding scheme.

Resources correspond to meaningful pieces of information for a given system that can be read from and/or written in the memory of a computer. They correspond to the definition I previously gave in subsection 3.2.3, which further illustrates the concept with examples, such as *files* and *objects* in operating systems and *clips* and *subtitles* in video editing systems.

Dependency

> A *dependency* is a constraint between resources located in the same computational context that must be enforced by the computer system.

Dependencies are fundamental to any computer system, in that they materialise logical constraints between memory regions that are enforced by sequences of computations performed by the computer, in accordance with the atomic instructions that can be executed by its processing unit. As such, every dependency must correspond to a computable function—in the sense of the Church-Turing thesis (Copeland, 2023)—in order to be implemented within a computer system.

Depending on the nature of the resources, dependencies can range from low-level mathematical relationships to high-level descriptive mappings. For example, a dependency of a colour encoded in an hexadecimal format, e.g., the string `#ff0000`, on a colour encoded in RGB, e.g., the numeric triplet $(255, 0, 0)$, is rather low-level, as it can be expressed in a sequence of fundamental operations, such as converting a number from a base to another and concatenating strings together. Conversely, in a system in which code is continuously transformed into another resource representing a document, such as HTML and CSS code that is continuously rendered as a webpage in a web browser, the dependency of the document (the output) on the code (the input) is high-level, as it requires a lot of complex operations to turn the latter's encoding into the former's encoding.

Dependency graph

> The *dependency graph* of a system is the directed graph (R, D) in which the set of vertices R includes all of the system's resources and the set of arcs D includes all of the system's dependencies.

The dependency graph of a computer system is a way to specify and implement what the computer must automate in the system, akin to, e.g., operationalising the system's behaviour using a state machine, as in SwingStates (Appert and Beaudouin-Lafon, 2006), or using a reactive graph, as in Stratify (Beaudouin-Lafon, 2023). It summarises the data and behaviour of a given computer system in a single conceptual object.

5.1.2 PROJECTION

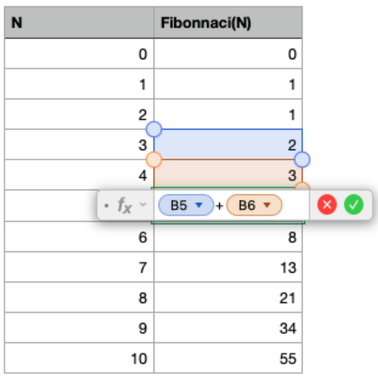
Computer systems as described above are enough to automate transformations between resources, but not enough to let us interact with them. Doing so requires to interact with input and output devices through substrates, as previously explained in subsection 3.2.4. To be interactive, a computer system must therefore map resources onto a substrate and vice-versa, in a process I call *projection*, in which resources are *projected* onto a substrate. The term is inspired by the concept of *projectional editor*, in which the code is encoded as a syntax tree and represented in an arbitrary manner, as introduced by Simonyi (1995). Given the dependency graph of a system, a projection corresponds to a subgraph formed by resources that are being projected, the *source*; a resource corresponding to the state of the substrate we can interact with, the *representation*; and the set of transformations that enforce the dependency of the latter onto the former, the *mapping*.

Source

> The *source* of a projection is the resource, or the set of resources, that contains the data the projection depends on.

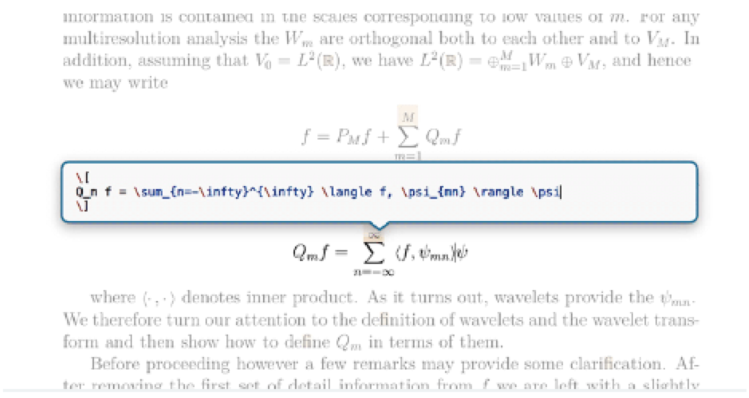
Regarding computer languages, the most obvious data at the source of a projection is code written in a computer language, either in its raw, encoded form, or further transformed into a data structure, such as a syntax tree. The source may also include the runtime state and the output of the program generated by the interpretation of the code, such as the value taken by a variable at runtime and the result of a computation. It may also depend on other data, such as an external file that contains data read by the code or online resources referenced in the code, which may be specified as command-line arguments or as paths and URLs in the code.

The resources directly connected to the substrate by a transformation constitute the *direct dependencies* of the projection. Moreover, although they are not part of the projection per se, the resources that direct dependencies depend on are also of interest when studying a projection. They constitute



N	Fibonnaci(N)
0	0
1	1
2	1
3	2
4	3
6	8
7	13
8	21
9	34
10	55

a. Apple Numbers.



information is contained in the scales corresponding to low values of m. For any multiresolution analysis the W_m are orthogonal both to each other and to V_M. In addition, assuming that $V_0 = L^2(\mathbb{R})$, we have $L^2(\mathbb{R}) = \oplus_{m=1}^{M} W_m \oplus V_M$, and hence we may write

$$f = P_M f + \sum_{m=1}^{M} Q_m f$$

```
\[
Q_n f = \sum_{n=-\infty}^{\infty} \langle f, \psi_{mn} \rangle \psi
\]
```

$$Q_m f = \sum_{n=-\infty}^{\infty} \langle f, \psi_{mn} \rangle \psi$$

where $\langle \cdot, \cdot \rangle$ denotes inner product. As it turns out, wavelets provide the ψ_{mn}. We therefore turn our attention to the definition of wavelets and the wavelet transform and then show how to define Q_m in terms of them.

Before proceeding however a few remarks may provide some clarification. After removing the first set of detail information from f we are left with a slightly

b. Compositor.

Figure 5.2. Two examples in which a representation supporting a lower level of interaction is used to complement a representation supporting a higher level of interaction. (a) In Apple Numbers, double-clicking a cell of the grid displays a projection of the formula that produces the cell's content, which can be edited as text. (b) In Compositor, in addition to directly editing the output document, users can also display and edit the piece of LaTeX code that generated a part of the output as text within the output itself.

the *indirect dependencies* of the projection. In addition, if a projection depends on a resource it does not read but only writes, this resource is called a *reverse dependency*.

Reverse dependencies are useful for describing situations in which the environment includes substrates that help modify a resource without representing it. For example, in a text editor, menu items that trigger commands for applying generic transformations (converting a text encoding into another, changing the font case, etc.) are substrates with reverse dependencies only: their representation is not constrained by any resource, but the text file open in the editor depends on them. They also appear in projections that project a resource R_1 on a substrate S, which modifies a resource R_2 when the user interacts with it. This is the case in output-directed editors, in which interacting with a representation of the output does not modify the output itself, but the code that generated it in the first place.

Representation

The *representation* of a projection is the virtual or physical substrate the user interacts with to perceive and act upon the source of the projection.

Most code editing environments include projections whose representations permit graphemic or morphosyntactic interaction, usually through a fairly generic substrate such as a textual, tree or graph substrate representing the entire code. Conversely, since representations designed to support semantic and pragmatic interaction are usually more concrete and specific, they are rarely standalone, and rather used within or next to a representation that supports graphemic or morphosyntactic interaction, such as by appending a colour picker or a plot to a text editor. As such, representations are often used to switch from a lower level to a higher level of interaction in terms of amount of interpretation, as per the hierarchy introduced in the previous chapter.

However, switching from a higher level to a lower level of interaction is also possible, as demonstrated in spreadsheets and output-directed editors (Figure 5.2). In spreadsheet applications, the default interface projects the code as a grid (morphosyntactic level) and displays the output of the computation described using the spreadsheet language as text in its cells (pragmatic level). It is only by clicking or double-clicking a cell that a user reveals a textual substrate that lets them interact with the underlying computer language in a graphemic fashion, therefore going down the layers of interpretation (Figure 5.2a).

Similarly, in output-directed code editors, in which the user directly manipulates the output to modify the code (pragmatic level), other representations may be included to help the user perform certain changes in the code. For example, a textual substrate may be displayed from within the output to let the user read and modify the piece of code responsible for a specific part of the output as text (graphemic level), as in Compositor (Figure 5.2b). This also includes representations of concepts used at intermediate steps of the computation yielding the output (semantic level), as in Sketch-n-Sketch (Hempel et al., 2019), in which the user can interact with all the lists that contain a shape shown in the output (Figure 5.3).

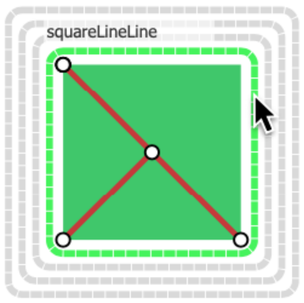


Figure 5.3. Sketch-n-Sketch reifies lists that contain a shape as dotted boxes called *list widgets*, shown around the shape in the output. The image is reproduced from Hempel et al. (2019).

Mapping

> The *mapping* of a projection is the set of transformations between the primary dependencies and the representation of the projection.

Mappings can include transformations going in different directions. If data from the source is projected onto a substrate, but changes in this substrates are not back-propagated so as to update the source, the transformation is said to be *prodirectional*. Conversely, if a representation does not depend on the data of its source but the user can modify it by modifying the representation (a case of reverse dependency), the transformation is said to be *retrodirectional*. If a transformation is both prodirectional and retrodirectional, it is said to be *bidirectional*; otherwise, it is said to be *unidirectional*. The *forward mapping* of a projection refers to the set of all prodirectional transformations it includes; whereas the *backward mapping* of a projection refers to the set of all retrodirectional transformations.

5.2 PROPERTIES

Besides being decomposed into a source, a representation and a mapping, projections can also be studied for the various properties they can exhibit. These properties can be used to describe and categorise existing projections, as well as to create new projections by exploring the design space one can construct out of these properties. This section describes seven such properties, as well as the values they can take, summarised in Table 5.1.

Some of the notions these properties refer to are reported or adapted from previous work. For example persistence, bidirectionality, liveness, compositionality and decentralized extensibility were previously mentioned by Omar et al. (2021), though most of the concepts they refer to were introduced even before that.[46] This thesis complements them with locality, contiguity, location,

46. For example, the term *liveness* is often attributed to Tanimoto (1990), who later revisited them (Tanimoto, 2013). Since then, it has been extensively used by various communities (such as HCI and creative coding) to qualify practices such as *live programming* and *live coding* (Rein et al., 2018).

Property	Values	Description
Locality		The projected data originates from…
	Global	…the entire source of the projection.
	Local	…a subset of the source of the projection.
Location		The projection is displayed…
	Inline	…in place of another projection of the code.
	Floating	…next to another projection of the code.
	Embedded	…within a projection of data related to the code.
	Standalone	…in an unrelated location of the user interface.
Persistence		After it has been used…
	Transient	…the projection disappears.
	Persistent	…the projection remains.
Compositionality		Code shown in the representation of the projection…
	Compositional	…can itself be projected.
	Not compositional	…cannot be further projected.
Liveness		When its source is modified, the projection…
	Live	…is updated in real time.
	Not live	…takes time to update.
Malleability		The projection can be created or modified by…
	Customisable	…changing predefined settings.
	Recomposable	…recomposing predefined building blocks.
	Extendable	…adding a third-party plug-in to the code editor.
	Scriptable	…using a custom script language.
	Reprogrammable	…editing the source code of the code editor.
Language agnosticism		The projection is compatible with…
	Language-specific	…a single (set of) computer language(s).
	Language-agnostic	…any computer language using the appropriate encoding.

Table 5.1. Summary of the seven properties of projections defined in this section. Depending on the situation, a property may qualify something bigger, e.g., the entire system, or smaller, e.g., a part of the representation, than the projection itself. For example, a projection may have a source that includes a first resource used globally, and a second resource used locally; and a projection can be qualified with more than one level of malleability at the same time. For further details and examples, please refer to the description of the properties in section 5.2.

malleability and language agnosticism, which I either created or adapted from work that did not use such terms to qualify projections of computer languages.

5.2.1 Locality

Locality characterises how much and what part(s) of the source is concerned by a projection. Each dependency on a resource can either be *global*, if the projection concerns the entire piece of data contained in the resource, or *local*, if the projection only concerns a subset. If the dependency is local, is can further be *contiguous* if it depends on a single region of the source data, or *non-contiguous* if it is formed of several non-overlapping regions. For example, the dependency of a text editor on a file containing code it displays is global, as the whole file can be edited as text, whereas the dependency of a colour picker on the same file is likely to be local, as it only depends on specific fragments of code representing colours. In the second case, the dependency is contiguous if each projection only projects a single fragment of code representing a colour; but it is non-contiguous if it also projects a subset of the runtime state containing the current value of a variable used as a colour component.

5.2.2 Location

Location characterises the relative position of a projection within the environment it is part of. If a projection appears within a substrate representing the code, as if it were part of this substrate, such as by taking up space reserved to characters in a textual substrate or to nodes in a tree or graph substrate, it is said to be *inline* in that substrate. If it appears close to another representation of the piece of code it represents, but not inline, e.g., next to a piece of text or to a node in a tree, it is said to be *floating* in that substrate. If it appears in a substrate representing something else than the code, such as a canvas displaying the output of the code, it is said to be *embedded* in that substrate. Otherwise, if it appears in an unrelated part of the environment, such as in a panel that the user can freely reposition, it is said to be *standalone*.

5.2.3 Persistence

Persistence characterises the reusability of a projection over time. If the projection remains available after having been used, it is said to be *persistent*. On the contrary, if it can only be used at a specific point in time and not anymore afterwards, it is said to be *transient*. For example, a colour picker is transient if it can only be used to insert a colour code and disappears once the colour has been chosen, as in Graphite (Omar et al., 2012); whereas it is persistent if the user can reinvoke it to modify the colour later on, as in Livelits (Omar et al., 2021).

5.2.4 Compositionality

Compositionality characterises the use of a projection within another: if at least some part of a representation is further projected onto another substrate, the resulting projection is said to be *compositional*. Put differently, a projection P_1 is compositional if (1) its source includes the state of another projection P_2 and (2) its representation replaces a subset of the perceptual space constrained by the representation of P_2. For example, a projection that arranges text within a grid substrate (such as for editing the code of a HTML table) is compositional if the content of each cell of the grid can in turn be projected onto other representations, such as a slider for manipulating text representing a numeric value, as demonstrated in Livelits (Omar et al., 2021), shown in Figure 5.5f below.

5.2.5 LIVENESS

Liveness characterises the time it takes for the system to enforce the direct *and* indirect dependencies of the projection depends when one of the resources it depends on is modified. If the transformations are fast enough so that the propagation of changes appears to be done in real time, the projection is said to be *live*. Critically, lack of liveness is a transitive burden: if any transformation in a sequence of dependencies from a resource R to a substrate S is not rapid enough, then the projection as a whole cannot be live, as it may not acquire information sourced in R fast enough for that. For example, if a projection of a piece of code onto a textual substrate also depends on runtime information to, e.g., display the current value of each variable in the text, and if interpreting changes in the code is too slow to update the running program in real time, then the projection cannot be live.

5.2.6 MALLEABILITY

Malleability characterises the freedom of the users of an interactive system to modify the projections by themselves. If a projection can be created, modified or deleted by the end-users without requiring an intervention from the developers of the underlying system, then it is said to be *malleable*. The notion of malleability echoes the notion of *tailorability*, as presented by Grønbæk et al. (2023), who identify five levels of tailorability, ranging from customising a projection by editing settings to reprogramming the system to modify its projection. These levels help explaining in which way a projection is malleable, rather than simply stating that it is. For example, most text editors can only be tailored by modifying settings (customisation) or adding plug-ins (extension), whereas visual syntax for DrRacket (Andersen et al., 2020) and Livelits (Omar et al., 2021) let the user write custom code to specify the look and behaviour of the projections themselves, as per the API provided by the system (scripting). Malleability is also similar to the notion of *decentralized extensibility* introduced by Omar et al. (2021) to "*distinguish [decentralized extensibility] from systems that are not extensible or that can only be extended via editor extensions or editor generation*".

5.2.7 LANGUAGE AGNOSTICISM

Language agnosticism characterises the independence of a projection from the computer language used to write the code it projects. If the projection can be used to interact with code regardless of the computer language it is written in, it is said to be *language-agnostic*. Conversely, if the projection only works with a single language, or set of languages, it is said to be *language-specific*. For example, a projection that can represent any substring that matches the pattern of an hexadecimal colour code is language-agnostic: it can be used to interact with any piece of code, no matter the language it is written in. However, if the projection relies on information specific to a language, such as decorated Java classes in Graphite (Omar et al., 2012) and typed holes in Livelits (Omar et al., 2021), it is language-specific. Furthermore, the specificity can be extended to entire classes of languages, rather than individual languages. For example, a projection may be compatible with the class of all languages that are encoded as plain text and can be compiled into a certain bytecode that the projection depends on, such as Java, Kotlin and Groovy, which all compile to Java bytecode.

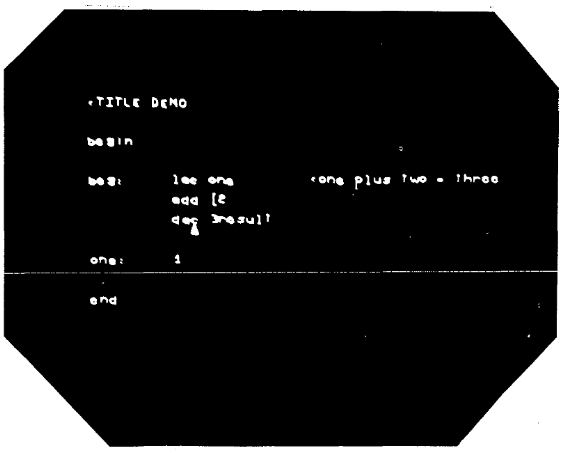
5.3 IMPLEMENTATION STRATEGIES

Over time, projections of computer languages have been implemented in a myriad of fashions, ranging from very simple textual projections printed on paper in the early 1960s, to grid and graph substrates containing complex content in the 1980s, to projectional editors that can represent each node of the syntax tree in a different way in the 2000s. These approaches evolved in accordance to both the evolution of the technology and the needs of the humans using it. This section traces this evolution and presents some milestones in the history of systems that project computer languages as a means to understand what has already been done, what has worked and what has failed. It introduces the knowledge required to properly frame the direction I took to pursue this line of work, which will be presented in the next section. To that end, it distinguishes between two major strategies for projecting computer languages: uniform projection and protean projection.

5.3.1 UNIFORM PROJECTION

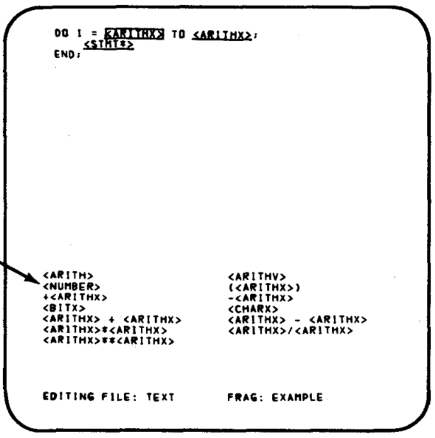
The first strategy to let users interact with a computer language is to project code onto a single substrate. I call this strategy uniform projection, for it leaves no choice to the user, who can only perceive and modify code through a single representation. This strategy is the oldest of the two, and was initially chosen as a default because of the limits of the technology available at the time. Yet, this approach persisted even when more versatile devices—such as modern screens—became widely available, suggesting that it should be understood as a choice of the designer of a system, rather than merely a technological constraint. Figure 5.4 shows six examples of code editing environments built with a uniform projection strategy.

The very first projections of computer languages were tightly coupled to their encoding. As teletypes and teleprinters served as front-end interfaces for interacting with computers, they long were the most common type of input and output devices, making text a candidate of choice for representing what was encoded in the computer's memory. This coincides with the development of English-like computer languages in the 1950s and 1960s, which



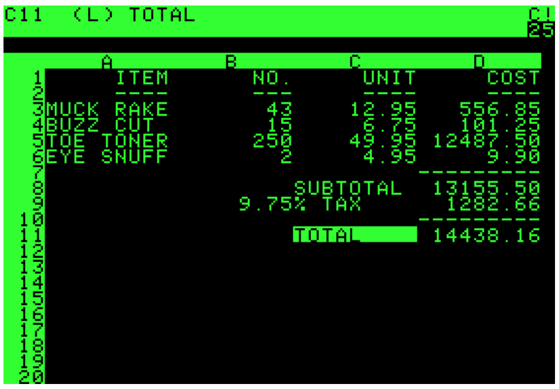
```
<TITLE DEMO

begin

beg:      lee one          <one plus two = three
          add [2
          dep ]result

one:      1

end
```

a. TVEDIT (1965).



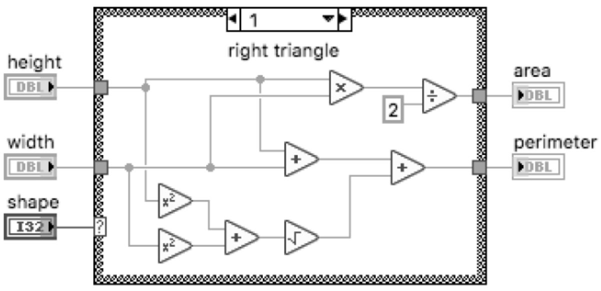
```
DO I = <ARITHX> TO <ARITHX>;
    <STMT*>
END;

<ARITH>                        <ARITHV>
<NUMBER>                       (<ARITHX>)
+<ARITHX>                      -<ARITHX>
<BITX>                         <CHARX>
<ARITHX> + <ARITHX>            <ARITHX> - <ARITHX>
<ARITHX>*<ARITHX>              <ARITHX>/<ARITHX>
<ARITHX>**<ARITHX>

EDITING FILE: TEXT             FRAG: EXAMPLE
```

b. EMILY (1971).

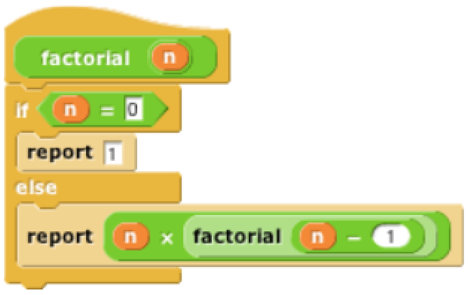


```
C11   (L) TOTAL                                  C!
                                                 25
         A         B          C          D
 1      ITEM        NO.       UNIT       COST
 2      ----        ---       ----       ----
 3MUCK  RAKE         43      12.95     556.85
 4BUZZ  CUT          15       6.75     101.25
 5TOE   TONER       250      49.95   12487.50
 6EYE   SNUFF         2       4.95       9.90
 7                                   --------
 8                        SUBTOTAL   13155.50
 9                      9.75% TAX     1282.66
10                                   --------
11                           TOTAL   14438.16
12
13
14
15
16
17
18
19
20
```

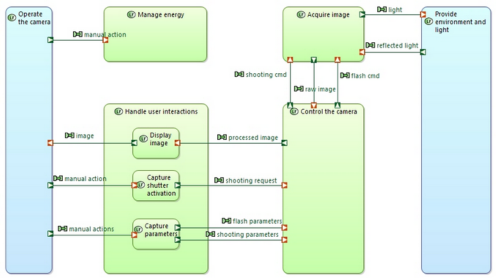
c. VisiCalc (1979).



d. LabView (1986).



e. Snap! (2011).



f. Eclipse Capella (2015).

Figure 5.4. Examples of uniform projection in six environments that let users interact with a computer language. (a) In TVEDIT, one of the first visual text editors, code is shown as a sequence of monochrome characters complemented with a cursor showing the current editing position. The image is reproduced from McCarthy et al. (1967). (b) In EMILY, code is projected onto a tree substrate according to its syntax and can only be modified by locally transforming the tree. The image is reproduced from Hansen (1972). (c) In VisiCalc, the code of the formula of the currently selected cell can only be edited through the textual projection shown at the top of the user interface. The result of evaluating the formulae is shown as text displayed within a grid substrate. (d) In LabView, code is projected onto a graph substrate made of text and iconic elements. The image is reproduced from Kodosky (2020). (e) In Snap!, code is projected onto a tree substrate made of blocks that be directly manipulated. The image is reproduced from the Snap! reference manual (@115). (f) In Eclipse Capella, code is represented by a series of diagrams specified in accordance with the UML standard, which can be created and edited using appropriate drawing tools and direct manipulation. The image is reproduced from the Eclipse Capella (@116) website.

were readily adapted to be typed and printed using such devices, as well as the development of some of the first text editing programs such as the Colossal Typewriter (McCarthy and Silver, 1960) and the Expensive Typewriter (Piner, 1972). As some computers started to be equipped with screens in the late 1960s, digital text editors were developed to virtualise text, yielding the first textual substrates as we known them. This includes line editors, which can only edit one line of text at a time, such as ed (@117), and their visual counterparts, which would display the text being edited and show the changes live, such as TVEDIT (Figure 5.4a) and vi (@118).

The idea of representing and interacting with computer languages independently from how they are encoded became increasingly viable with the development of the first graphical user interfaces for computers in the late 1960s, pioneered by the famous demonstration of the NLS system in 1968 (Engelbart and English, 1968). Most notably, some of the first syntax-directed editors, such as EMILY (Hansen, 1972), shown in Figure 5.4b, demonstrated that computer languages could also be projected onto trees according to their syntax, therefore offering new interaction opportunities. Yet, as the name suggests, syntax-directed editing is circumscribed to projecting the code so as to let users manipulate the syntactic structures it forms, therefore only bridging the gap from graphemic to morphosyntactic interaction, leaving semantic and pragmatic interaction out of the question at the time.

During the 1970s, the growth of graphical user interfaces and the development of more and more capable computers led to a variety of systems in which the user interacts with a graphical projection of an underlying model, regardless of how it is encoded. Fundamentally, models are not so different from computer languages: they too are formed of a number of primitives, which can be combined together to form more complex structures. Yet, unlike most computer languages available at that time, such models were not meant to be edited using representations of *how* they are encoded. Instead, each model could be edited using a domain-specific projection that was more fit for representing *what* the model describes, making it prompt for semantic interaction.[47] This idea was applied to tasks such as writing documents with word processors, creating images using image editors, and developing software with graphical software engineering tools (Kuhn, 1989; Fuggetta, 1993). It enabled the development of new interaction techniques and paradigms now taken for granted, such as direct manipulation of domain objects and WYSIWYG interfaces, effectively forming a split between "old textual systems" and "new visual systems" (Shneiderman, 1983).

47. For example, LaTeX or Markdown have been designed to write text documents using generic text editors, whereas documents encoded in formats such as RTF and DOC are designed to be read by word processors, which are pieces of software dedicated to authoring certain types of models—namely, structured text document encoded using dedicated schemes.

This new approach paved the way for the so-called *visual languages* (and, more specifically, visual *programming* languages), which usually corresponded to projecting code—often specific to a particular domain—onto a graph substrates, therefore making use of two or more dimensions, in line with the definition proposed by Myers (1990). Early examples of such languages include LabView (Kodosky, 2020), shown in Figure 5.4d, which was designed to let scientists and engineers manipulate virtual instruments, and Max (Puckette, 2002), which was designed for musical creation.[48] The growing availabilities of personal computers with graphical user interfaces seem to have catalysed the growth of visual languages in the 1980s. Kodosky, the main author of LabView, reports that their potential for the programming language he wanted to

48. Puckette (2002) reports that Max was later declined into several variants, including the proprietary Max/MSP and the open-source Pure Data languages, presented earlier.

create became immediately obvious to him the day he tried a graphical user interface for the first time:

> *Then came the summer of 1984. My brother-in-law sat me down in front of his Macintosh computer, showed me how to use the mouse and opened MacPaint. It was a revelation. I immediately went out and bought a Mac [...] I was convinced the Mac represented the future of human-computer interfacing. The Mac's graphics provided an obvious way for software instruments to reflect the physical instruments we worked with. The virtual controls on the computer screen could be just like the controls on an instrument, except more flexible. Now it was clear that a virtual instrument should have a graphical interface on the computer screen.*
>
> — Kodosky (2020, §2.3)

Similarly, Puckette, the author of Max, reports that "*many other graphical patch languages—both for music and for other applications—had appeared by 1987 when I started writing the Max "patching" GUI*" (Puckette, 2002, p. 33), suggesting that projecting computer languages onto graph substrates was already common in the mid-1980s.

This period also coincided with the development of the first spreadsheet systems, although these are usually not qualified as visual programming languages. While visual programming environments pioneered the projection of computer languages onto graph substrates, spreadsheet applications pioneered the projection of the code *and* its output onto the same grid substrate, effectively introducing one of the earliest form of pragmatic interaction with computer languages. Examples include VisiCalc (Figure 5.4c), released in 1979, which was the first publicly released spreadsheet application; soon followed by competitors such as Lotus 1-2-3, released in 1983.

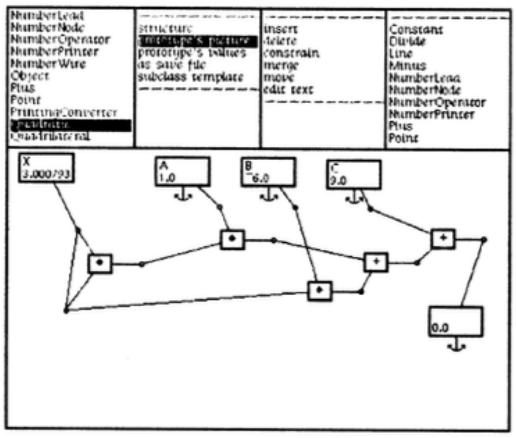
This uniform approach to interaction is further reflected in the desire to consolidate languages and notations for describing models in the 1990s. One of the popular outcomes of that movement is the Unified Modeling Language (UML), a standard for modelling any computer system. Although it was initially designed to be descriptive, it was later adapted to be made executable,[49] paving the way for model-driven engineering systems in the 2000s (Mellor and Balcer, 2002). Since then, such standards have been implemented by software such as Eclipse Sirius (@120), a system for describing models and projecting them onto visual diagrams that originate from the UML standard, as in Eclipse Capella (@121), shown in Figure 5.4f.

49. For example, Foundational UML (fUML, @119) is a subset of the UML standard that provides semantics for translating models specified in this subset of UML into executable programs given a compatible target architecture.

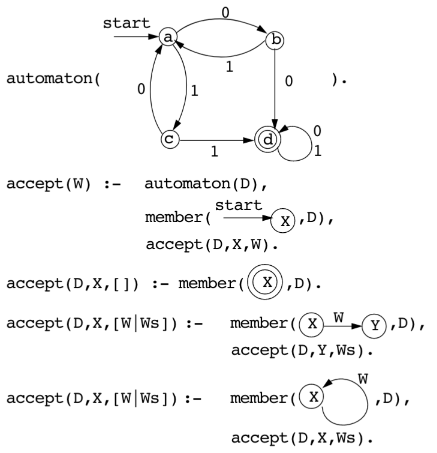
5.3.2 PROTEAN PROJECTION

The other strategy to let users interact with a computer language is to project code onto more than one substrate. I call this strategy protean projection, for the user can switch between different representations to perceive and modify code. Figure 5.5 shows six examples of code editing environments built with a protean projection strategy.

Although protean projection was mostly developed in the last two decades, a number of older systems were designed with this strategy in mind. For example, the ThingLab system (Figure 5.5a), developed on top of Smalltalk in the late 1970s, already lets users draw and manipulate constrained elements

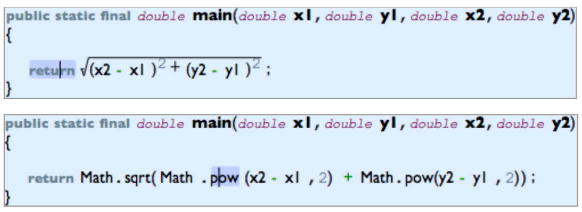


a. ThingLab (1979).



```
automaton(                    ).

accept(W) :-  automaton(D),
              member(      ,D),
              accept(D,X,W).

accept(D,X,[]) :- member(   ,D).

accept(D,X,[W|Ws]) :-   member(       ,D),
                        accept(D,Y,Ws).

accept(D,X,[W|Ws]) :-   member(       ,D),

                        accept(D,X,Ws).
```

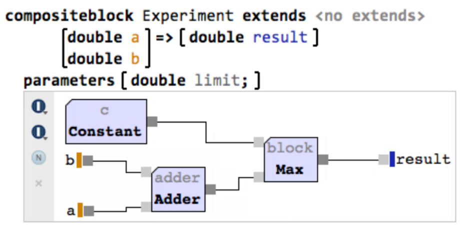
b. Heterogeneous languages (1995).



```
public static final double main(double x1, double y1, double x2, double y2)
{
    return √(x2 - x1 )² + (y2 - y1 )² ;
}
```

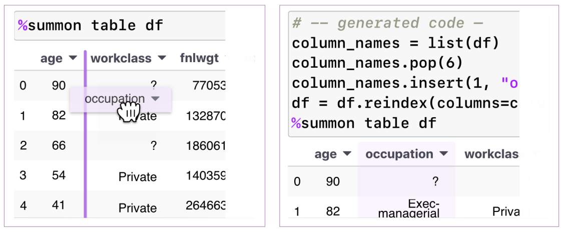
```
public static final double main(double x1, double y1, double x2, double y2)
{
    return Math.sqrt( Math .pow (x2 - x1 , 2) + Math.pow(y2 - y1 , 2));
}
```

c. Barista (2006).



```
compositeblock Experiment extends <no extends>
    [double a ] => [ double result ]
    [double b ]
  parameters [ double limit; ]
```

d. JetBrains MPS (2010s).



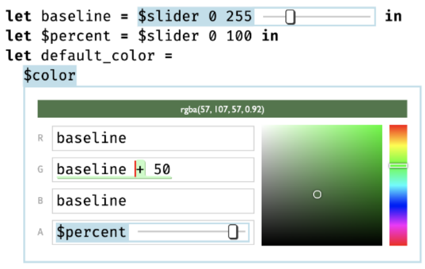
```
%summon table df
```

	age	workclass	fnlwgt
0	90	?	77053
1	82	Private	132870
2	66	?	186061
3	54	Private	140359
4	41	Private	264663

```
# -- generated code —
column_names = list(df)
column_names.pop(6)
column_names.insert(1, "o
df = df.reindex(columns=c
%summon table df
```

	age	occupation	workclas
0	90	?	
1	82	Exec-managerial	Priva

e. mage (2020).



```
let baseline = $slider 0 255       in
let $percent = $slider 0 100 in
let default_color =
  $color
```

```
baseline
baseline + 50
baseline
$percent
```

f. Livelits (2021).

Figure 5.5. Examples of protean projection in six code editing environments. (a) In ThingLab, the user can edit code by inserting and manipulating visual elements on a canvas. This creates and modifies objects, which can also be edited as text in a different part of the interface (not shown). The image is reproduced from Borning (1979). (b) In this heterogeneous declination of the Prolog language, primitive shapes such as circles and arrows can be mixed with text. The system is able to interpret them according to some shape grammar, which is used to turn them into Prolog code when need be. The image is reproduced from Erwig and Meyer (1995). (c) In a Java editor created with Barista, an expression can be projected onto a typeset mathematics substrate when browsing the code, and projected as text when the user wishes to edit it. The image is reproduced from Ko and Myers (2006). (d) In an editor created with JetBrains MPS, text can be laid out in a non-sequential fashion, or even replaced by a different projection, such as a box-and-wire diagram. The image is reproduced from Voelter and Lisson (2014). (e) In Jupyter notebooks augmented with mage, the user can insert special commands within a textual substrate to display interactive output cells. These cells can project both static and dynamic data onto arbitrary substrates, such as the content of a dataframe onto a grid substrate, which can further modify the code when the user interacts with them. The image is reproduced from Kery et al. (2020). (f) In Hazel augmented with Livelits, the user can insert visual macros (such as `$color`) within a variant of Elm projected as text, which can each be projected onto arbitrary substrates, such as a colour picker. The image is reproduced from Omar et al. (2021).

displayed on a canvas whose class and instances could also be edited via a textual projection (Borning, 1979). Starting from the late 1980s, more and more systems in which a computer language could be edited via multiple projections were developed. Examples includes FormsVBT (Avrahami et al., 1989), heterogeneous visual languages (Erwig and Meyer, 1995), shown in Figure 5.5b, and Adobe Dreamweaver (@104).

At about the same time UML was first standardised, Simonyi (1995) published his seminal work on *intentional programming*, predicting "*the death of computer languages*" as we know them. In this work, Simonyi argues that software developers should not be imposed a particular syntax or compilation target, nor be restricted to using a single language just because interconnecting code written in different languages is too difficult. As an alternative, he proposes a switch to a different vision of programming, made of *intents*. Intents could refer to each other so as to form a graph of intents, although it is mostly envisioned as a tree—later called an *intentional tree* (Simonyi et al., 2006). The graph (or tree) of intents could then be transformed into machine code as need be, and could be edited by projecting its nodes so as to match the preferences and needs of the language's users, which could be, e.g., text, formulas with sub- and super-scripts, or box-and-wire diagrams. Although the gist of a graph of intents that can be translated to match a particular computer architecture is reminiscent of UML, intentional programming is, to the best of my knowledge, the first theory to have envisioned that a single piece of code could be represented in different ways, in fact introducing the use of the term *projection* for that purpose.

The idea of decoupling computer languages from a unique representation of the code further took off in the 2000s. For example, in his vision of extensible programming environments for the 21st century, Wilson (2004) imagines that computer languages will be encoded as XML and systematically edited through domain-specific projections, just like domain-specific models have been for decades already:

> *We believe that next-generation programming systems will most likely store source code as XML, rather than as flat text. Programmers will not see or edit XML tags; instead, their editors will render these models to create human-friendly views, just like Web browsers and other WYSIWYG editors.*
>
> — Wilson (2004, p. 54)

Wilson's vision was indeed implemented by systems for creating so-called projectional editors for computer language, in which each node of the syntax tree can be displayed as an arbitrary user interface, as in Barista (Ko and Myers, 2006), JetBrains MPS (Voelter and Lisson, 2014) and Envision (Asenov, 2017). It also shows up in model-driven engineering systems, which progressively departed from relying on the strict, generic and unified UML notation towards offering users a variety of domain-specific notations. The goal was to let experts in different domains work with models using notations they are familiar with without requiring any particular software engineering knowledge, as in the Whole platform (Solmi, 2005), Gentleman (Lafontant, 2022) and Eclipse Sirius (@120).

The vision in which users must define and manipulate abstract entities (objects) that can have more than one representation (views) is also embodied within the Glamorous Toolkit (@122), a programming system built on top of Pharo. In the Glamorous Toolkit, a single class of object can define more than one view, so that when the object is inspected—a notion similar to showing the content of a file in a Unix system—, the user can freely switch between multiple projections for interacting with that object, each shown in a different tab. Feenk (@123), the company that develops the Glamorous Toolkit, describes it as a system designed for *moldable development*, an approach in which specialised tools—and therefore specialised projections—can be built and adapted to match the specifics of each use case. The concept of moldable development is in line with recent work on the notion of *malleable software*, i.e., software designed to let end-users adapt their systems to their needs, as theorised by Tchernavskij (2019) and demonstrated by systems such as Codestrates (Rädle et al., 2017) and Mirrorverse (Grønbæk et al., 2023).

Yet, in spite of the theoretical advantages claimed by Simonyi, Wilson and others, and apart from a few industrial applications such as mbeddr for C (Voelter et al., 2019), implementations of protean projection in which computer languages that were primarily designed to be encoded and edited as text are encoded as trees or graphs of entities that must be projected using a dedicated editor have hardly been adopted in the wild. To create a projectional editor using this approach, one must learn the specifics of the underlying software—such as the meta-language it relies on—and implement both the specification of the language (the concepts, the syntax, the type system, etc.) *and* the projections available to edit it, resulting in a significant development effort. In addition, since the language is encoded in a format specific to the resulting editor, it results in a niche ecosystem in which all users willing to use the language are forced to use this specific editor, with hardly no alternative. While this may be appropriate in certain situations, such as when the code editor will be used by domain experts to write code for critical systems or by students in a controlled educational setting, it may otherwise be too high a cost, both for the developers of the language/editor and the end-users, resulting in overly specific products and underused languages.

In the last decade, augmenting a text editor with additional projections has been increasingly reported in the literature as an alternative approach to implement protean projection. Examples from the early 2010s includes Graphite (Omar et al., 2012) and Codelets (Oney and Brandt, 2012), which both feature user interfaces for configuring code snippets when they are inserted in a text editor. Although they are not part of a code editor, online code generators[50] can also be considered to belong to this trend as they offer specialised user interfaces for synthesising pieces of code serving specific purposes. Unfortunately, these systems are not exactly designed to project code, but only a model that specifies what the code means: once it has been transformed into code, the model—and therefore the user interface—is not updated when the code is modified, such as by editing it as text.

50. Example of such code generators include Tables Generator (@124) for creating and editing tables in different document description languages and CSS Gradient (@125) for configuring gradients in CSS.

This limitation was progressively addressed, resulting in systems in which the code is primarily projected as text but can also be projected differently, as demonstrated by Moonchild (@126), visual syntax (Andersen et al., 2020), as well as mage (Kery et al., 2020) and Livelits (Omar et al., 2021), shown in

Figure 5.5e and Figure 5.5f. In addition to addressing the lack of bidirectionality of the aforementioned systems, mage and Livelits also support a form of pragmatic interaction by exploiting information sourced from the execution of the code they project. To this day, these two systems represent the state-of-the-art in combining a textual substrate with a mix of other substrates, as recently noted by Horowitz and Heer (2023). As such, they constitute one of the main inspirations of my work.

5.4 FOCUS OF THIS THESIS

Despite the work described above, I want to underline how prevalent text-encoded computer languages and graphemic interaction using text editors are in the 2020s. Although describing this trend with exact figures is difficult, some indicators help get a glimpse at how big it is. Regarding computer languages, the Octoverse 2022 (@127), a study of the activity on GitHub, reveals that among the hundreds millions repositories it hosts, the five most popular languages are JavaScript, Python, Java, TypeScript and C#. Regarding the tools we use to interact with them, a survey carried out by StackOverflow (@128) in Spring 2023 reports that Visual Studio Code, Visual Studio, IntelliJ IDEA, Notepad++ and Vim—i.e., five text editors—are ranked as the five most popular code editors by over 86,000 respondents. Furthermore, recent research on structured editors has focussed on making them compatible with textually encoded languages, as well as on carefully emulating the experience of regular text editor (Hempel et al., 2018; Beckmann et al., 2023a).

Whether the motivation is to be readily compatible with many established languages or to accommodate the programmers' habit and desire to interact with text, augmenting text editors with additional projections is a safe option to create tools compatible with the ecosystem that most computer languages currently live in. While this may appear to be an engineering constraint rather than a research agenda, I believe that making novel projections that help interacting with computer languages used by millions compatible with the real-world workflows that they already use is a hard problem which no satisfying solution, therefore making it an excellent candidate for research. This position aligns with Arawjo's invitation to "*build on the lessons of the past by embracing, rather than avoiding, heterogeneity in programming practice*" (Arawjo, 2020, p. 9), as well as Kell's conclusion regarding how we can recycle great ideas that vanished along with Smalltalk's failure by integrating them into a Unix-dominated world:

> *Smalltalk's modernist narrative holds that unification entails implementing one "unified" system—a Smalltalk runtime. The directions outlined in the previous section are motivated by a postmodern goal: to accept the complex reality of existing ("found") software, developed in ignorance of our system, and to shift our system's role to constructing views, including Smalltalk-like ones, of this diverse reality.*
>
> — Kell (2013, §7)

The work I did during this Ph.D. is highly influenced by such postmodern visions. Instead of designing systems at odds with established practices, I rather chose to design *for* them, with the explicit goal of making my work compatible with existing computer languages and code editors, rather than creating new languages and editors unlikely to ever be adopted.

I started by investigating how to apply the concept of protean projection to LaTeX, an established computer language that is widely used in academia and often criticised for its many flaws. Informed by the outcomes of this work, I then shifted my focus to the question of the *appropriation* of projections by their users, following Dix's observation that "*people do not 'play to the rules': they adapt and adopt the technology around them in ways the designers never envisaged*" (Dix, 2007, §2.1). These two lines of work complement the theoretical contributions I made in the current and past two chapters by applying them to real situations with the help of a user-centred design methodology. They yield the main empirical and technical contributions of my work, which I present in the next two chapters.

6

Transitional representations for LaTeX

Digital document preparation systems can be roughly divided into two categories: those in which the user directly interacts with the document, such as when writing a letter using Microsoft Word; and those in which the user interacts with a description or a specification of the document, such as when editing HTML as text to create a webpage. Systems that explicitly rely on a document description language usually fall into the second category. In most of these systems, the user must first code the document using a special computer language, and then use a different program to generate the final document, as do web browsers to display webpage composed of HTML, CSS and other resources.

In this first line of my applied work, I decided to focus on LaTeX, an advanced document description language that was created in the early 1980s and remains in use today, primarily in academic and technical circles. LaTeX is a prime example of a long-standing computer language that is almost always edited via a textual projection in spite of encoding structures that have long benefited from other representations, such as tables and images, making it a good candidate to investigate whether and how protean projection could help LaTeX users write code. To answer this question, I used a user-centred design methodology by grounding my research using a formative user study and evaluating the solution I propose using two evaluative user studies. The results were published in two articles: a preliminary article published in the national IHM'21 conference (Gobert and Beaudouin-Lafon, 2021), which was awarded the best paper award of the conference and a honourable mention for the demonstration, followed by an article in the international ACM CHI'22 conference (Gobert and Beaudouin-Lafon, 2022).

This chapter presents my work on this project as an extended version of the second publication. Section 6.1 situates this work amongst previous work on interacting with the LaTeX language and authoring digital documents, which inspired the directions I chose to follow. Section 6.2 presents a formative interview study with 11 LaTeX users I carried out to better understand the way they use LaTeX and the problems and needs they face in this process. Section 6.3 introduces the concept of *transitional representation*, a specific type of projection that I developed in accordance with the recommendations for design identified when analysing the interviews. Section 6.4 presents *i*-LaTeX, a prototypal LaTeX editor equipped with four alternative projections that complement the standard textual substrate used to author LaTeX documents. Sections 6.5 and 6.6 report on a controlled experiment with 16 participants and a longitudinal study with 6 participants that I conducted to evaluate the effects of transitional representations for LaTeX. Section 6.7 concludes on the outcomes of this work, highlighting the demonstrated success of transitional representations while identifying limitations for spreading their use, which steered my research interest towards a different research question that I report on in the next chapter.

6.1 BACKGROUND

Since its inception, various concepts and tools have been developed around LaTeX, forming a rich ecosystem of practices that influences how users write documents in LaTeX. Moreover, there is a rich body of work on authoring and interacting with digital documents, both within and beyond the LaTeX ecosystem, whose ideas and findings are highly relevant to investigate the effect of new interaction techniques for document description languages. This section presents previous work relevant to each of these topics, which both relates to and differs from the work I will present in the rest of this chapter.

6.1.1 THE LaTeX ECOSYSTEM

LaTeX is a document description language created by Leslie Lamport in 1984 (Lamport, 1994). It was designed as an extension of TeX, a more primitive document description language developed by Donald Knuth in the late 1970s (Knuth, 1984b) whose primary task is to compose documents—i.e., to arrange all the elements that form a document, which range from single glyphs to sequences of paragraphs and floats—to output a sequence of pages of definite sizes. Compared to plain TeX, LaTeX provides concepts that are more user-friendly for writing documents, such as by distinguishing between several classes of documents (articles, letters, etc.) using the `\documentclass` macro and by providing a standard set of macros for structuring them with sections, subsections, etc.

In addition to referring to a computer language, LaTeX also refers to systems designed for interpreting code written in the eponymous language and transforming it into a typeset document. Such systems usually combine a set of macro packages—including macros specific to each document class and macros provided by third-party libraries that can be loaded by the user—and TeX engines, such as pdfTeX (@129), XƎTeX (@130) or LuaTeX (@131), which are computer programs that take TeX code augmented with the aforementioned macros as input and generate a PDF or DVI file representing the coded document as output.

Because of its highly programmatic and configurable nature, LaTeX offers powerful abstraction mechanisms and a high level of control over the generated document. However, it comes at the cost of long compilation times, as generating a PDF document from LaTeX code often takes seconds to minutes. Moreover, precisely understanding how the system interprets a piece of code is difficult, as it requires to switch to a very unusual mental model and learn about advanced features of TEX, such as character classes, which let users to change the lexical meaning of any input character anywhere in the code of a document. LATEX is also famously known for being hard to debug, as errors are often difficult to understand by non-experts and poorly linked to the part of the code that caused them (which can be hard to determine). As a result, according to Knauff and Nejasmic, even expert users may "*experience a loss of productivity when LATEX is used, compared to other document preparation systems*" (Knauff and Nejasmic, 2014).

Yet, nearly four decades after its inception, and despite all its flaws, LATEX is still widely used to write technical documents in some communities, even though alternative computer languages for advanced typesetting such as SILE (@132) and Pollen (@133) have been developed. In particular, LATEX has been reported to be one of the few authoring systems that is extensively used in academia and research, along with Microsoft Word (Knauff and Nejasmic, 2014). Overleaf (@27), an online platform for authoring LATEX documents, recently reported a user base of over 6 million users, as well as partnerships with multiple universities and scientific publication venues—such as ACM conferences—to provide their users with appropriate templates for authoring papers (Reis et al., 2021).

6.1.2 INTERACTING WITH LATEX

Since LATEX was conceived as a macro-oriented language meant to be encoded and edited as text, most environments for authoring LATEX documents look and work like text editors. Some of them try to provide a user experience closer to that of WYSIWYG systems, but they usually either provide basic source code formatting, as do, e.g., AUCTeX (@134) and Overleaf's *rich text* mode, or a fully WYSIWYG interface that hides the code and only supports a limited set of features, as does, e.g., Compositor (@105). LyX (@135) represents documents in an intermediate format that focuses on the content and the structure rather than the final layout and style—a paradigm referred to as *What You See Is What You Mean* (WYSIWYM) by its authors—, but it uses its own document format, and the use of the LATEX language is restricted to importing/exporting the document as LATEX and inserting short pieces of code to write, e.g., mathematical formulae.

In some situations, LATEX code can also be synthesised, either using another programming language or an interactive code generator. As an example, LATEX code representing a table can be generated programmatically using the `pandas` data-science library in Python,[51] and interactively using dedicated applications such as Tables Generator (@124). The caveat of these approaches is their unidirectionality: once the code has been generated and imported into the user's document, these tools can no longer be used to edit the table unless the user is willing to overwrite all the changes they made to the code.

51. In Pandas, `DataFrame` objects have a `to_latex` method (@136) that outputs a string containing LATEX code laying out the data frame's content in a table.

Finally, a few tools address the lack of mapping between the code and its output. Some LATEX editors or plugins, such as LaTeXTools (@137) for Sublime Text, let users preview mathematical formulae and images by hovering over the code. SyncTEX (Laurens, 2008) helps find which region of the code corresponds to a part of the PDF—and vice-versa—and Gliimpse (Dragicevic et al.,

2011) further animates the transition between the LaTeX code and generated PDF. However, none of them let users modify the code by interacting with the output.

6.1.3 INTERACTING WITH DIGITAL DOCUMENTS

Besides work specific to LaTeX, I was inspired by a rich body of work on interaction with digital documents, which are not constrained to be as static as traditional paper-based documents, and can be interactively modified by their authors and readers. In his *Explorable Explanations* (@138) essay, Bret Victor argues in favour of making traditionally static documents more explorable, by encouraging the reader to modify parts of the document, such as numeric values, and see how it affects other parts. Multiverse analyses (Dragicevic et al., 2019) let readers try out different visualisations and analyses of the same data from within a scientific publication. SpaceInk (Romat et al., 2019) lets users move content around to make space for hand-written annotations. Textlets (Han et al., 2020) turn text selections into persistent objects, to which users can assign interactive behaviours, such as counting words or storing alternative content. ScolarPhi (Head et al., 2021) augments scientific papers with overlays that provide definitions and context for technical terms and symbols. Charagraph (Masson et al., 2023a) lets users extract numbers from selected sentences and turn them into plots to help users understand relationships and statistical properties of multiple values. Statslator (Masson et al., 2023b) helps reconstructing missing statistical information from the one present in a document by exploiting the mathematical relationships between different statistical tests.

Various directions have been proposed to help users create such interactive documents. This includes using variants of popular document description languages augmented with new primitives, such as Markdown-based languages in Idyll (Conlen and Heer, 2018) and Myst (@139) and HTML-based languages in Nota (@36), as well as new languages, such as Typst (@140) and FFL (Wu et al., 2023). It also includes more decoupled solutions, such as automatically recognising document parts to augment using computer vision techniques and overlaying them with interactive user interfaces (Masson et al., 2020). In addition, this move towards more interactive documents further calls for new systems for authoring them, which must be equipped to let users insert and configure interactive elements.

Because of the inherently programmable nature of such elements, these systems are likely to expose fragments of code to authors in one way or another, including when the overall user interface is mostly designed for pragmatic interaction, making them a good testbed for alternative projections of document description languages. For example, the recent surge in popularity of literate computing systems (Fog and Klokmose, 2019)—a modern take on Knuth's idea of *literate programming* (Knuth, 1984a)—, such as Jupyter notebooks (@16), has fostered the development of editing environments that weave pieces of rich text with blocks of executable code. Document editors such as Idyll Studio (Conlen et al., 2021) and Potluck (@141) illustrate how projections that support semantic and graphemic interaction with the underlying language can be embedded in environments that primarily display the final document to the user, either in separate panels or localised popups.

By interacting with a document description language at the graphemic or morphosyntactic level, users can reason about and manipulate every feature supported by the language, including advanced configuration and abstraction mechanisms, at the cost of decoupling their actions on the code from their effect on the final document. In contrast, by interacting at the semantic or pragmatic level—as in WYSIWYG software, which presents documents in their final form at all times—, users can perceive and edit the style and content of the document in a situated and live fashion, without having to map abstract linguistic concepts to concrete changes in the output in their mind. As a result, each approach has its own strengths and weaknesses, both are widely used, and neither seems to supplant the other. Inspired by the aforementioned work on tools for authoring interactive documents that embed code, I envisioned that projections of LaTeX code that are neither purely graphemic (as in text editors) nor pragmatic (as in WYSIWYG editors) may help users address the difficulties they face when they author LaTeX documents.

6.2 FORMATIVE STUDY

In order to better understand how alternative projections of LaTeX code could help LaTeX users, and given the scarcity of previous studies of LaTeX users, I conducted a series of interviews and performed a thematic analysis of the difficulties they encounter. This section presents the methodology, the generated themes, and the resulting recommendations for designing better LaTeX editors.

6.2.1 METHODOLOGY

Participants

I interviewed 11 participants (5 women and 6 men, age 21 to 40), recruited via an internal lab mailing list and a post on a Facebook group of university students. They did not receive any compensation for their participation. Eight were M.Sc. students, the others were a Ph.D. student, a high-school teacher, and an associate professor. Most of them were neither beginners nor experts with LaTeX.[52] Each of them had used LaTeX in the weeks or months prior to the interview, usually with Overleaf (5/11) or Texmaker (5/11). Additional details about the participants are presented in Table 6.1.

52. In both the interview and the evaluation studies, I refer to three levels of expertise with LaTeX defined as follows: *beginner users* do not feel comfortable when they use LaTeX and need help to use basic features; *intermediate users* feel comfortable writing basic documents and are able to use the most common features without help; and *expert users* feel comfortable with various kinds of document and are able to create or customise commands and environments when needed.

Setup

All the interviews were conducted remotely via screen sharing. They all lasted about one hour in average.

Procedure

The interviews were semi-structured: I used a list of predefined questions to guide the participants, but they were invited to speak freely about each

Participant	**Occupation**	**Domain**	**Expertise**	**Main presented document**
P1	M.Sc. student	Biology	Intermediate	Internship report
P2	Ph.D. student	Data visualisation	Intermediate	Ph.D. thesis
P3	M.Sc. student	Ecology	Intermediate	Biophysics assignment
P4	M.Sc. student	Geology	Intermediate	Hackathon project
P5	Associate prof.	Computer science	Expert	Mathematics paper
P6	M.Sc. student	Complex systems	Intermediate	Thesis proposal
P7	M.Sc. student	Geochemistry	Intermediate	Internship report
P8	M.Sc. student	Archaeology	Beginner	Internship report
P9	M.Sc. student	Computer science	Expert	Computer science paper
P10	High school teacher	Philosophy	Intermediate	Archive of philosophy exams
P11	M.Sc. student	Computer science	Expert	Mathematics class notes

Table 6.1. Details on the LaTeX users I interviewed.

problem they encountered. I adjusted the duration and the questions with three pilot interviews with colleagues. I started each interview by asking the participant to show me the last LaTeX document they worked on (both the code and the generated PDF). I first asked them questions related to the document itself (the type of document, the editor they used, etc). I then asked them to describe the different problems they faced, whether they eventually solved them or not and how, and to show me the related parts of the code when it was relevant. I also invited them to tell me about problems they encountered in other LaTeX documents. I concluded each interview with a few more general questions about the participants' experience with LaTeX.

Data collection

The screen and the audio of the participants were recorded during the study. In addition, I also took notes of the problems they faced and the solutions they used.

Data analysis

I conducted a thematic analysis (Braun and Clarke, 2019) of the collected data. To that end, I manually transcribed the first eight interviews, including descriptions of the screen content and participants' actions when it was relevant. I then annotated the transcripts, yielding over 650 codes. Codes that conveyed the same idea with different phrasings were merged, resulting in about 80 different codes that I used to generate three sets of themes: a first set that grouped codes by topic; a second set that grouped codes by problem, independently of the context in which they appeared; and a third and final set that mixed the first two sets based on discussions with colleagues, so as to highlight the most prevalent and interesting findings. In addition, I also listened to the three remaining interviews that I did not transcribe (for the sake of saving time) in order to ensure that they fit with the final set of themes and extract quotes from them to both reinforce and contrast what had already been highlighted in the other interviews.

I generated five themes through the thematic analysis, which can be summarised as follows: the code must be editable as text (T1), users struggle learning the language (T2), plain text is inappropriate to describe structured content (T3), some abstractions are difficult to visualise and formalise (T4), and the code-PDF duality slows users down (T5).

T1 — The code must be editable as text

While most participants were comfortable using graphical text processors, P2 complained that the documents are not properly structured and both P4 and P11 highlighted the quality of documents produced with LaTeX.

No participant was aware of or interested in WYSIWYG LaTeX editors, and no participant who used Overleaf actively used its *rich text* mode. When she had a look at it, P8 criticised that hiding the code made it harder to make certain changes, such as transforming a section into a subsection, therefore requiring "*more manipulation, more movement on the screen*" (P8) to regularly go back to the code. P7, the only participant who used it once to write a document, explained that it was because she only had to write plain text to take notes during a history class, without any kind of structured content.

Some participants (5/11) needed the ability to write code to program features that did not exist out of the box, and could therefore not be accessed through any existing user interface. Most notably, P10 created a template for a collection of hundreds of high-school philosophy examinations. It enabled him to generate multiple indexes (by author, by topic, by type of philosophical question, etc.) that are automatically updated when he adds new examinations later on—a level of automation he could not achieve with his usual text processor: "*with LibreOffice, you can only have an alphabetical index for one category*" (P10). The separation between content and style also enabled him to generate two PDFs with very different layouts by switching a single parameter in the code: one for teachers with extra metadata, and one for students that mimics the official examination's layout.

Several participants (7/11) explained that they appreciate reusing code. The most expert ones make extensive use of custom commands to reuse mathematical expressions or duplicate a parameterised drawing. P5 explained that having access to the code makes it possible to easily copy-paste snippets found on the internet, instead of following the steps listed in a tutorial one by one, as well as to reuse the code of his scientific papers to create presentations for conferences using Beamer (a LaTeX package for creating slideshows). Moreover, P5 also reported that having access to the code let him generate documents from scratch when needed. For example, when he was responsible for creating hundreds of similar badges for a Go contest, he wrote a Python program to generate the appropriate LaTeX code: "*If you ask me to make you two hundred badges in Word, you're on your own; I'm not doing it*" (P5).

Most participants (9/11) reported using comments within the code. They rely on them for different purposes, including (1) remembering the role of a package or command; (2) discussing with co-authors; (3) keeping unused pieces of code that might be useful later; (4) commenting off lines to find the source of an error iteratively; (5) reusing parts of an old piece of code to write a new version; or (6) planning what to write in each part of the document.

T2 — Users struggle or avoid learning the language

Five participants explained that they only learn LaTeX when they need to solve a problem they face. Yet, several of them also pointed out that this often happens in moments of rush, such as before a deadline, therefore leaving them no opportunity to take the time to understand the code that caused or solved a problem. As a consequence, fixing specific problems was often described as time-consuming when there was no ready-made solution they could readily reuse. For example, P7 reports that since she wanted a very self-specific bibliography design, "*[customising] the bibliography was the longest thing, because I couldn't find a ready-made solution. I had to invent a little bit.*" (P7). For this reason, P2 and P4 expressed regrets not taking a course on LaTeX. P6, on the other hand, said that he would not make such an effort. For him, using LaTeX means tweaking existing code until you get the desired result, without understanding how it works: "*you find a document that works, you copy and paste it, and you iteratively change it. [...] I will not learn the structure of the code.*" (P6).

Instead of trying to learn the language or reading the documentation of the packages they use, all the non-expert participants (8/11) reported that they would rather search for solutions to specific problems: creating a particular style or layout, adjusting a certain margin, fixing an error, etc. In addition, most participants (9/11) seem to forget or ignore the exact names of some macros or the order and the meaning of their parameters, even when they use them on a regular basis: "*I know I had to use a minipage, but I still had to look for how to use it*" (P3). Yet, all but one participant reported that it was not always easy to find a solution to their problems. For example, when she had to insert ancient Greek into her document, P8 first attempted to "*look into stuff for linguists and historians*", but could not find an answer.

T3 — Plain text is inappropriate for structured content

Many participants (8/11) complained about the difficulty of describing structured elements such as tables, sub-figures, and chemical formulae. In particular, creating or editing tables was often described as "*really annoying*" (P1). For example, according to P7, "*forgetting a column is hell*" because "*for every row I must count to put [the cursor] in the right place*" (P7).

In response to these difficulties, more than half the participants (6/11) use third-party tools. Both P2 and P4 started to create sub-figures by combining several images in LaTeX before switching to Inkscape or Adobe Illustrator because it gave them "*more control on the layout*" (P2). P8 reported that one of her co-authors used Paint to annotate a photography they used in their LaTeX document, something that would have been directly "*feasible with Word*" (P8). To create tables from data stored in Microsoft Excel spreadsheets, P7 exported them as CSV files that she loaded into Tables Generator (@124), an online table editor that generates LaTeX code. However, because of the complexity of the generated code, she would rather modify the data in Excel and repeat the whole process than edit the generated code. P1, P7 and P9 also reported using this website to create tables from scratch. In addition, P7 also complained about not being able to export the LaTeX code of the molecules she created in ChemDraw, a piece of software for chemists, because the textual syntax of the `chemfig` package, meant for achieving similar results in LaTeX, was "*the hardest thing I've ever used*" (P7).

T4 — Abstractions are difficult to visualise and formalise

Many macros in LaTeX require specifying dimensions, such as the size of an image or the margin around an element. However, according to several participants (7/11), it is difficult to express a length that they picture in their head as a value with a unit (difficulty to formalise), and it is difficult to imagine the length such a value represents (difficulty to visualise). P8 faced this issue when she inserted images into her document: "*I don't necessarily know the exact size I want the image to be, but [I know] I want it to be* that *size in my head*" (P8). Similarly, P1 complained about having to try several dimensions to find the right one, a time-consuming strategy, "*especially when you have a lot of figures, as it takes a long time to compile.*" (P1). To overcome these difficulties, some participants (4/11) mentioned using commands they are more familiar with in an alternative way. For example, P3 inserted white text in his document (using `\color{white}`) to skip several lines of text: "*I suppose there was a simpler way to do it, but since I was in quite a hurry, that's how it is*" (P3).

Unlike dimensions, which must usually be specified by the user, LaTeX often automatically determines the positions of various elements. This can result in a lack of control that is not always desirable. P1 explained that her way of positioning figures was "*not very rational*", and P2 complained about the difficulty of displaying an image next to a subsubsection that refers to it when LaTeX places it somewhere else. These difficulties were sometimes caused by a lack of understanding of positioning parameters, such as those of `figure` environments, which were often copied with the rest of a piece of code. Even the more expert participants struggled with positioning. In spite of reading about "*how the compiler positions images*" to better control the process, P4 admitted that she still had to lower her expectations concerning the positions of her figures. P5 explained that even though he felt comfortable with the drawing commands of the TikZ package, he would prefer to be able to directly manipulate some of the elements that compose his drawings instead of "*trying to guess*" the correct coordinates or doing "*some kind of trigonometry*" to calculate them.

T5 — The code-PDF duality is a source of errors and slowdowns

Participants who wrote a lot of mathematical formulae (3/11) complained about the difficulty of relating regions or glyphs in the PDF to the code that generated them. When writing mathematical papers, both P5 and P9 have trouble (1) locating the code of the mathematical formula displayed in the PDF and (2) finding the symbol they want to edit within the code of the formula. To solve the first problem, they often search for a few words from the text located just above or below the formula in the code editor, although P5 admitted that this technique regularly fails. This approach is not unrelated to the lack of support for SyncTeX in the LaTeX editor that P5 uses, who even explained that he considers switching to another editor for that reason. Neither P5 nor P9 have found a solution to the second problem: every time they want to edit a formula, they have to read the code to find the part to change.

Two participants complained about the time required to compile the code into a PDF, which not only increases the cost of trying alternative layouts, but also makes errors very time-consuming: "*if I get it wrong, it costs me a minute*" (P5). Both of them developed strategies to minimise this cost. In order to compile less often, P1 distinguishes between writing and formatting phases: "*when I'm in a writing phase [...] I just write [...] but when I'm formatting, I always open [the PDF] on the side, to compile regularly.*" (P1). P5, on the other hand, gave the example of a package that caches images created with TikZ: as long as the code is not modified, it "*re-injects the image instead of the TikZ code*" (P5). In addition, P4 mentioned that compilation time was an important factor for choosing a LaTeX editor.

6.2.3 RECOMMENDATIONS FOR DESIGN

This thematic analysis reveals a variety of problems faced by LaTeX users and suggests several directions for improving the design of LaTeX editors. I summarise them in the following recommendations.

R1 — Let users view and edit the code as text

No graphical user interface can completely hide the code of a LaTeX document without restricting its users, as no graphical user interface can support every LaTeX package nor every feature that can be self-programmed by advanced users. In addition, some users are willing to reuse pieces of code found online or in other documents, as well as automate the generation of document by synthesising code with the help of a program. I therefore recommend that LaTeX editors let their users read and modify the code in their documents freely and easily, so as to accommodate to this diversity of practices, and without trapping users in using a single piece of software for authoring their LaTeX documents.

R2 — Provide task-specific features in addition to generic language support

While the majority of participants were not opposed to the idea of learning the LaTeX language, none of them can or want to devote a lot of time to it. Instead, most participants were looking for help to solve specific problems, and only considered learning the concepts and syntax of the language along the way, which they sometimes forget. In addition to providing generic features intended for language experts, such as auto-completion of macros one must already know about, I therefore recommend that LaTeX editors include features that are task-specific, with the goal of targeting specific but common problems and needs—e.g., authoring tables and laying out sub-figures—and helping less experienced and intermittent users quickly find solutions to their problems.

R3 — Make structures and abstractions visible and interactive

Although the LaTeX language was initially meant to be encoded as text and edited in a graphemic interaction fashion, this type of projection poorly supports manipulating structures, such as mathematical and chemical formulae, and symbolic representations of visual properties, such as coded lengths and position information. I therefore recommend that LaTeX editors let users perceive and interact with structures and abstractions otherwise represented as text through other projections using bidirectional transformations, so as to maintain a two-way synchronisation between these alternative user interfaces and the textual projection of the code.

R4 — Strengthen links between the code and the generated document

The fact that the code and the generated document are two separate resources, and that recreating the latter when the former is modified can hardly be performed in real time,[53] forces users to work with two different and asynchronous representations of the same information and construct and maintain links between parts of their respective projections in their mind. I therefore recommend that LaTeX editors make links between the code and the generated document more visible and granular, such as by making it easier to identify and modify the macro responsible for a specific mathematical symbol in the PDF.

53. This limitation has been adressed by the Texifier editor (@142), which is capable of updating the typeset document in real time when the source code is modified. However, as their authors explain it (@143), achieving this requires to write a custom TeX engine that is deeply intertwined with the editor, so as to support partially typesetting the document and writing the visual output directly within the GPU's memory, without writing the document on the disk.

6.3 TRANSITIONAL REPRESENTATIONS

Most document authoring systems only provide a uniform projection of the editable document to the user, whether it is a textual projection of code written in a document description languages,[54] a tree of blocks representing the document structure in WYSIWYM editors such as LyX, or an editable version of the typeset document in WYSIWYG editors such as Microsoft Word. While this prevents users from having to switch between multiple representations—an operation that can sometimes be prohibitively costly, as exemplified by the seldom use of Overleaf's *rich text* mode by the LaTeX users I interviewed—, it also constrains users to perceive and interact with every element of every document through a single projection, which has its own limitations, as suggested in subsection 6.2.3.

54. I do not consider the generated document here, as it is usually *not* editable.

I argue that moving from uniform to protean projection of LaTeX code can help address several of the challenges identified in the previous section by enriching the environment for editing code available to LaTeX users beyond text while preserving a full compatibility with existing LaTeX workflows. To that end, I introduce the notion of *transitional representation*, or *transitional* for short, a specific kind of projection for document description languages. In this section, I frame the concept, define its key properties, compare it with related concepts and systems, and show how it applies to LaTeX.

6.3.1 DEFINITION

A *transitional* is an alternative projection of a fragment of code primarily projected as text that can be displayed by interacting with the region of the output document generated by this very fragment. The concept of transitional complements a number of existing document authoring paradigms, such as WYSIWYG, WYSIWYM and output-directed programming, as illustrated in Figure 6.1. For example, a transitional could be used to interact with a fragment of code representing a table by arranging the code of each cell within a directly manipulable grid structure whenever the user clicks on the typeset table in the output document (Figure 6.2).

A transitional does *not* turn a static document into a WYSIWYG editor: they usually provide a different representation than the output, and they are only meant to project the code of a specific fragment of code, not the code of the entire document. In the example given above, the grid is complementary to the output—which may, e.g., not show the grid of the table at all—and only contains the code of the table's cells. To edit the rest of the code, the user is still expected to use the textual projection (or, when appropriate, another transitional).

6.3.2 PROPERTIES

Transitionals are *local*, *bidirectional*, *persistent* and *embedded* projections of the code. They are primarily designed for document description languages, but could be used to interact with code written in other kinds of computer languages as well, as long as it generates an output.

Locality

Transitionals are local projections of the code: they only project a limited fragment of the code, which represents a single element of the document. In contrast with WYSIWYG systems such as Adobe Dreamweaver and output-direct programming systems such as Sketch-n-Sketch (Hempel et al., 2019), which require the whole output to be interactive, transitionals can be used with a static output, such as a PDF, as they can be displayed *in addition* to the code and the output—and not *in place* of them.

Bidirectionallity

Transitionals feature a bidirectional mapping between the code (the resource) and the user interface it displays (the representation), so that every change in the alternative representation is reflected in the code, and vice-versa. Moreover, this mapping is expected to run in real time, therefore making transitionals *live* projections of the code they represent. Unlike code generated once from a specification or a sketch, this approach enables users to dynamically switch between the textual projection and the transitional depending on which representation is the most adapted to their current needs. As an example, a user may use a transitional to drag handles shown on top of an image to resize it interactively, even if the entire document cannot be regener-

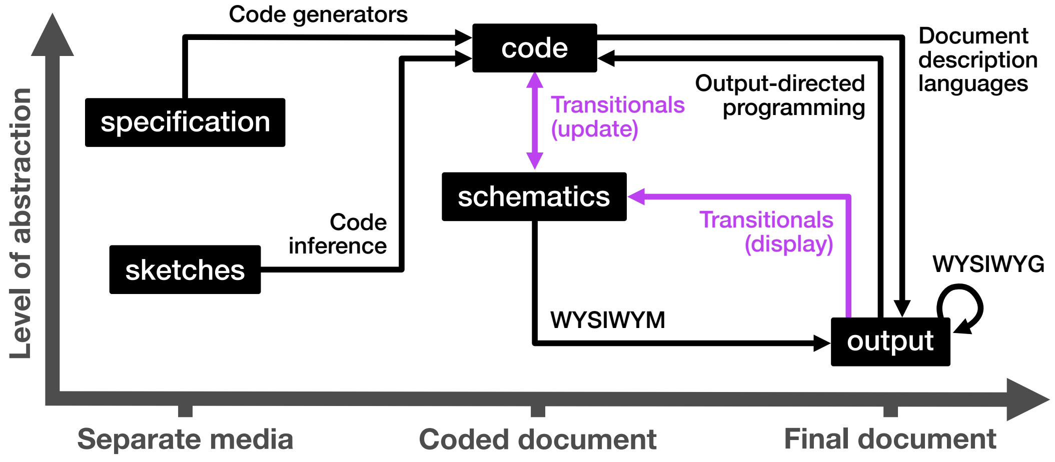
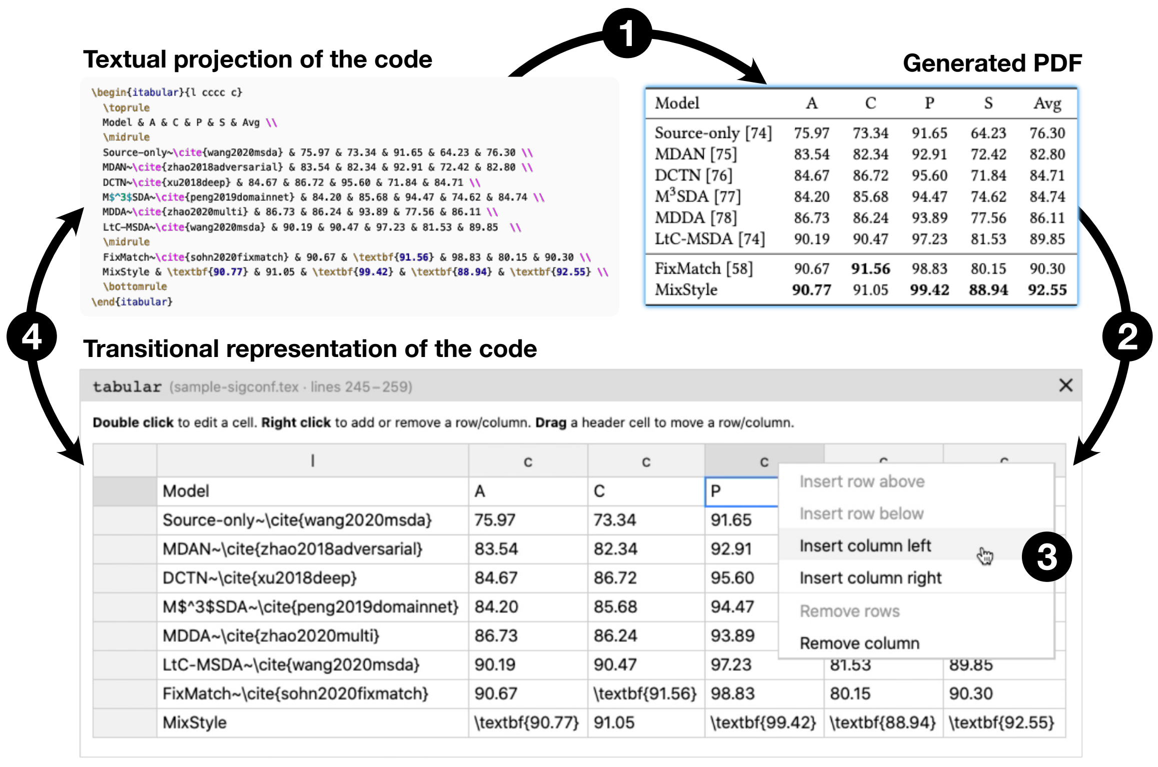


Figure 6.1. Space of document authoring paradigms. Nodes represent document representations, and arrows represent authoring paradigms that connect different representations. Transitionals can be displayed by interacting with the output, and modifying them updates the code, and conversely.



Textual projection of the code

```
\begin{itabular}{l cccc c}
  \toprule
  Model & A & C & P & S & Avg \\
  \midrule
  Source-only~\cite{wang2020msda} & 75.97 & 73.34 & 91.65 & 64.23 & 76.30 \\
  MDAN~\cite{zhao2018adversarial} & 83.54 & 82.34 & 92.91 & 72.42 & 82.80 \\
  DCTN~\cite{xu2018deep} & 84.67 & 86.72 & 95.60 & 71.84 & 84.71 \\
  M$^3$SDA~\cite{peng2019domainnet} & 84.20 & 85.68 & 94.47 & 74.62 & 84.74 \\
  MDDA~\cite{zhao2020multi} & 86.73 & 86.24 & 93.89 & 77.56 & 86.11 \\
  LtC-MSDA~\cite{wang2020msda} & 90.19 & 90.47 & 97.23 & 81.53 & 89.85  \\
  \midrule
  FixMatch~\cite{sohn2020fixmatch} & 90.67 & \textbf{91.56} & 98.83 & 80.15 & 90.30 \\
  MixStyle & \textbf{90.77} & 91.05 & \textbf{99.42} & \textbf{88.94} & \textbf{92.55} \\
  \bottomrule
\end{itabular}
```

Generated PDF

Model	A	C	P	S	Avg
Source-only [74]	75.97	73.34	91.65	64.23	76.30
MDAN [75]	83.54	82.34	92.91	72.42	82.80
DCTN [76]	84.67	86.72	95.60	71.84	84.71
M^{3}SDA [77]	84.20	85.68	94.47	74.62	84.74
MDDA [78]	86.73	86.24	93.89	77.56	86.11
LtC-MSDA [74]	90.19	90.47	97.23	81.53	89.85
FixMatch [58]	90.67	**91.56**	98.83	80.15	90.30
MixStyle	**90.77**	91.05	**99.42**	**88.94**	**92.55**

Transitional representation of the code

`tabular` (sample-sigconf.tex · lines 245–259)

Double click to edit a cell. **Right click** to add or remove a row/column. **Drag** a header cell to move a row/column.

	l	c	c	c	c	c
	Model	A	C	P		
	Source-only~\cite{wang2020msda}	75.97	73.34	91.65		
	MDAN~\cite{zhao2018adversarial}	83.54	82.34	92.91		
	DCTN~\cite{xu2018deep}	84.67	86.72	95.60		
	M^{3}SDA~\cite{peng2019domainnet}	84.20	85.68	94.47		
	MDDA~\cite{zhao2020multi}	86.73	86.24	93.89		
	LtC-MSDA~\cite{wang2020msda}	90.19	90.47	97.23	81.53	89.85
	FixMatch~\cite{sohn2020fixmatch}	90.67	\textbf{91.56}	98.83	80.15	90.30
	MixStyle	\textbf{90.77}	91.05	\textbf{99.42}	\textbf{88.94}	\textbf{92.55}

Figure 6.2. Three representations of the same table in *i*-LaTeX, which was adapted from Zhou et al. (2021). ❶ The code of the table is compiled into a static PDF element. ❷ The table can be clicked in the PDF (as suggested by the blue halo) to display its transitional—here, the code represented as text organised in a grid. ❸ The user can interact with the transitional to modify the structure and the content of the table. ❹ The transitional and the code are synchronised, so that every change in either one of them instantly updates the other representation. For example, inserting a new column in the grid automatically adds cell separators in the code, which show up as new ampersands in the text.

ated at every step, before switching to the textual projection to fine-tune the width and height parameters.

Persistence

Transitionals are persistent projections of the code: they can be hidden and displayed again at any time while remaining in sync with the code, even when the code is modified in the meantime. This makes transitionals similar to alternative representations offered by a number of projectional editors, such as MPS (Voelter and Lisson, 2014) and Livelits (Omar et al., 2021), but different from code generators and Graphite's palettes (Omar et al., 2012), which do not remain linked to the code once they have been used.

Embeddability

Transitionals can be displayed within the output of the code, by interacting with the region of the output generated by the piece of code that the transitional projects. This makes it possible to edit the code that describes an element in the output using a specialised representation (similar to that offered by WYSIWYG editors) without having to first locate and decipher the fragment of code in the entire source code—with all the challenges that this implies, as discussed in theme T5.

6.3.3 APPLICATION TO LATEX

The concept of transitional representation bridges the gap between the two interaction paradigms that inspired it: projectional editing, in which each node of the syntax tree can be projected arbitrarily, and output-directed programming, in which manipulating an element in the output of a program transforms the code that generated it. Transitionals address the following two limitations of these concepts by enabling a form of "output-directed projectional editing".

The first limitation is conceptual: the textual projection of the code or the output alone may not be sufficient or adapted for performing certain tasks. As an example, Mozilla Firefox's developer tools include a projection of the value of a CSS timing function property (previously shown in Figure 4.4c) that may be hard to interpret and modify as text, and whose animated target in the rendered webpage offer no affordance to inspect and modify the timing function. This is an example of a situation in which neither graphemic nor pragmatic interaction alone is adapted to the task, and where offering the user a projection that lets them semantically interact with the property complements other projections already provided by the system. The authors of Sketch-n-Sketch make a similar observation regarding the usage of output-directed programming in non-trivial documents: "*manipulation of the final output alone will be insufficient*", and therefore, "*some of the intermediate process should be exposed on the canvas for manipulation*" (Hempel et al., 2019, p. 5). Transitionals address this limitation by design, as a transitional can include more, less or different information to make up for what is missing in the output or help the user focus on what is important. In addition, by providing multiple transitionals for a single element, users can choose the representation that best fits their current need.

The second limitation is technical: depending on the document description language and the complexity of the code, evaluating the code may be too slow for real-time code synthesis and document rendering. In the case of LaTeX, non-trivial documents are usually too slow to compile to update the output in real time when the code is modified—let alone turning the static output generated by the LaTeX compiler into a fully interactive document, as previously noted by Laurens (2007). By displaying a projection of the code of a single element of interest, the whole compiler can be traded for an ad-hoc static analysis that extracts all the information required by the transitional in real time. Moreover, by displaying a user interface that is visually similar to what the code generates in the output, transitionals enable a form of local live programming: users can see the effects of changes they make in a specific region of the code in real time, even though the entire document is not updated.

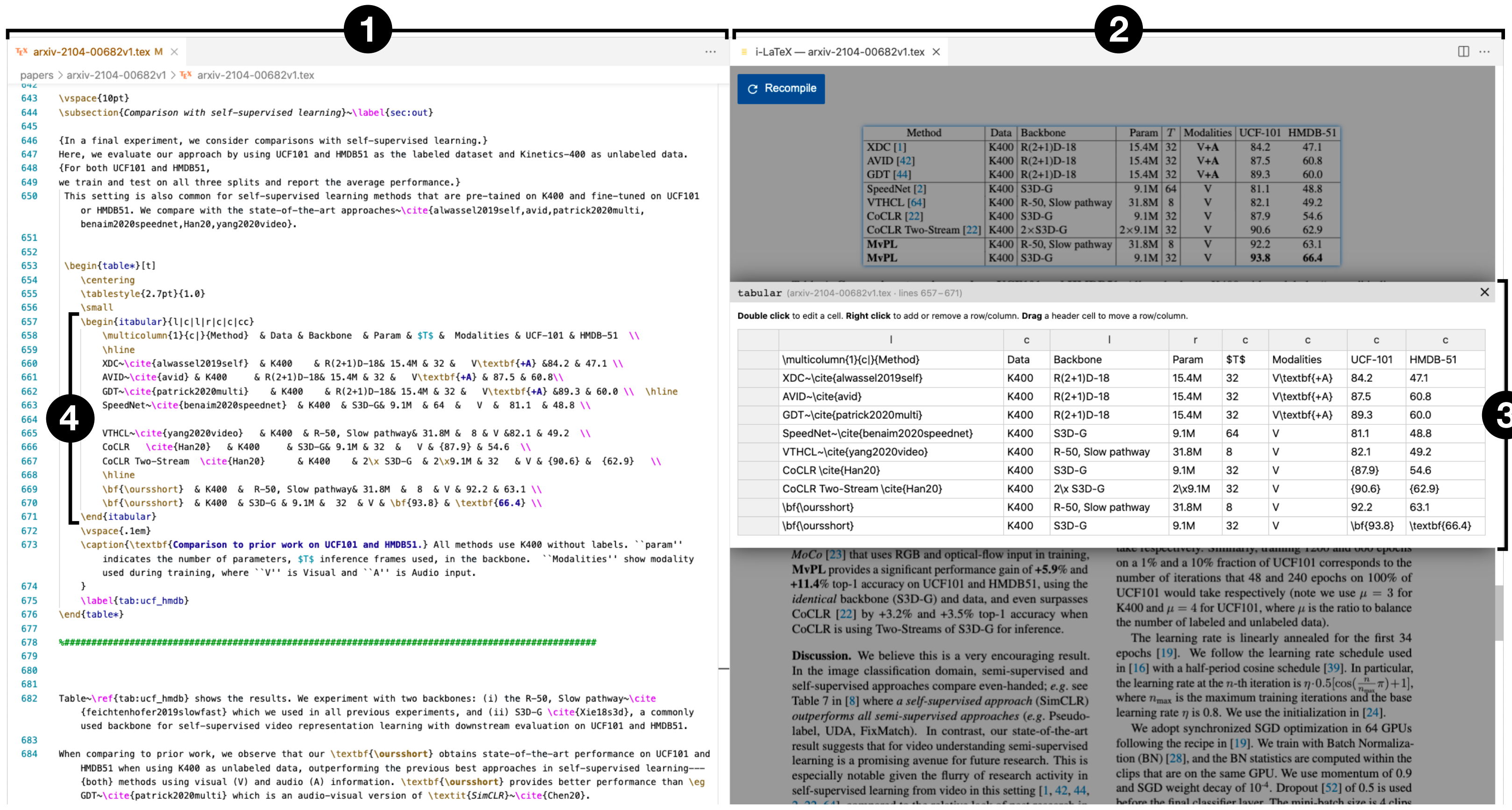
The properties of transitionals and the concepts they build upon make them conceptually and technically adapted to help LaTeX users author documents. The thematic analysis showed that while editing the code of a LaTeX document is often preferred or required by LaTeX users (T1), there are a number of situations in which a textual projection of the code is ill-adapted, such as when working with structured content (T3) and abstract values (T4). It also highlights two areas where current LaTeX editors fall short: supporting specific actions on common types of elements (T2), and connecting the code with its output (T5). Transitionals are a good option for improving LaTeX editors, as they adhere to the four recommendations presented earlier: they complement textual projections instead of replacing them (R1); they are tailored to help user perform common tasks on specific target elements (R2); they can visualise structures and abstractions that may be hidden in the text and lost in the output (R3); and they help linking fragments of code and their output in a semantically-aware fashion (R4).

6.4 THE *i*-LaTeX EDITOR

Informed by the results of the thematic analysis, I selected four candidates for transitional representations adapted to the needs of LaTeX users and created *i*-LaTeX, a LaTeX editor featuring transitionals for mathematical formulae, tables, images and grid layouts. In this section, I present the design of *i*-LaTeX's user interface, the four transitionals, the key points of the implementation, the current limitations and the possible extensions of the software.

6.4.1 USER INTERFACE

The interface of *i*-LaTeX resembles that of most traditional *i*-LaTeX editors, with the source code on the left and the generated PDF on the right. However, some elements in the PDF have a blue outline, indicating that they are interactive. Clicking on one of them displays an interactive visualisation of the piece of code that generated the element (Figure 6.3). These transitionals let users (1)


1
2
3
4
arxiv-2104-00682v1.tex M
papers > arxiv-2104-00682v1 > arxiv-2104-00682v1.tex
i-LaTeX — arxiv-2104-00682v1.tex
Recompile

```latex
\vspace{10pt}
\subsection{Comparison with self-supervised learning}~\label{sec:out}

{In a final experiment, we consider comparisons with self-supervised learning.}
Here, we evaluate our approach by using UCF101 and HMDB51 as the labeled dataset and Kinetics-400 as unlabeled data.
{For both UCF101 and HMDB51,
we train and test on all three splits and report the average performance.}
 This setting is also common for self-supervised learning methods that are pre-tained on K400 and fine-tuned on UCF101
    or HMDB51. We compare with the state-of-the-art approaches~\cite{alwassel2019self,avid,patrick2020multi,
    benaim2020speednet,Han20,yang2020video}.


 \begin{table*}[t]
   \centering
   \tablestyle{2.7pt}{1.0}
   \small
   \begin{itabular}{l|c|l|r|c|c|cc}
      \multicolumn{1}{c|}{Method}  & Data & Backbone  & Param & $T$ &  Modalities & UCF-101 & HMDB-51  \\
      \hline
      XDC~\cite{alwassel2019self}  & K400    & R(2+1)D-18& 15.4M & 32 &   V\textbf{+A} &84.2 & 47.1 \\
      AVID~\cite{avid} & K400     & R(2+1)D-18& 15.4M & 32 &   V\textbf{+A} & 87.5 & 60.8\\
      GDT~\cite{patrick2020multi}    & K400    & R(2+1)D-18& 15.4M & 32 &   V\textbf{+A} &89.3 & 60.0 \\  \hline
      SpeedNet~\cite{benaim2020speednet}  & K400  & S3D-G& 9.1M  & 64  &   V  &  81.1  & 48.8 \\

      VTHCL~\cite{yang2020video}   & K400  & R-50, Slow pathway& 31.8M &  8 & V &82.1 & 49.2  \\
      CoCLR   \cite{Han20}   & K400     & S3D-G& 9.1M & 32  &   V & {87.9} & 54.6  \\
      CoCLR Two-Stream  \cite{Han20}      & K400    & 2\x S3D-G  & 2\x9.1M & 32   & V & {90.6} &  {62.9}   \\
      \hline
      \bf{\oursshort}  & K400  &  R-50, Slow pathway& 31.8M  &  8  & V & 92.2 & 63.1 \\
      \bf{\oursshort}  & K400  & S3D-G & 9.1M &  32  & V & \bf{93.8} & \textbf{66.4} \\
   \end{itabular}
   \vspace{.1em}
   \caption{\textbf{Comparison to prior work on UCF101 and HMDB51.} All methods use K400 without labels. ``param''
      indicates the number of parameters, $T$ inference frames used, in the backbone.  ``Modalities'' show modality
      used during training, where ``V'' is Visual and ``A'' is Audio input.
   }
   \label{tab:ucf_hmdb}
\end{table*}

%##################################################################################################



Table~\ref{tab:ucf_hmdb} shows the results. We experiment with two backbones: (i) the R-50, Slow pathway~\cite
    {feichtenhofer2019slowfast} which we used in all previous experiments, and (ii) S3D-G \cite{Xie18s3d}, a commonly
    used backbone for self-supervised video representation learning with downstream evaluation on UCF101 and HMDB51.

When comparing to prior work, we observe that our \textbf{\oursshort} obtains state-of-the-art performance on UCF101 and
    HMDB51 when using K400 as unlabeled data, outperforming the previous best approaches in self-supervised learning---
    {both} methods using visual (V) and audio (A) information. \textbf{\oursshort} provides better performance than \eg
    GDT~\cite{patrick2020multi} which is an audio-visual version of \textit{SimCLR}~\cite{Chen20}.
```

Method	Data	Backbone	Param	T	Modalities	UCF-101	HMDB-51
XDC [1]	K400	R(2+1)D-18	15.4M	32	V+A	84.2	47.1
AVID [42]	K400	R(2+1)D-18	15.4M	32	V+A	87.5	60.8
GDT [44]	K400	R(2+1)D-18	15.4M	32	V+A	89.3	60.0
SpeedNet [2]	K400	S3D-G	9.1M	64	V	81.1	48.8
VTHCL [64]	K400	R-50, Slow pathway	31.8M	8	V	82.1	49.2
CoCLR [22]	K400	S3D-G	9.1M	32	V	87.9	54.6
CoCLR Two-Stream [22]	K400	2×S3D-G	2×9.1M	32	V	90.6	62.9
MvPL	K400	R-50, Slow pathway	31.8M	8	V	92.2	63.1
MvPL	K400	S3D-G	9.1M	32	V	**93.8**	**66.4**

`tabular` (arxiv-2104-00682v1.tex · lines 657–671)

Double click to edit a cell. **Right click** to add or remove a row/column. **Drag** a header cell to move a row/column.

	l	c	l	r	c	c	c	c
	\multicolumn{1}{c|}{Method}	Data	Backbone	Param	\$T\$	Modalities	UCF-101	HMDB-51
	XDC~\cite{alwassel2019self}	K400	R(2+1)D-18	15.4M	32	V\textbf{+A}	84.2	47.1
	AVID~\cite{avid}	K400	R(2+1)D-18	15.4M	32	V\textbf{+A}	87.5	60.8
	GDT~\cite{patrick2020multi}	K400	R(2+1)D-18	15.4M	32	V\textbf{+A}	89.3	60.0
	SpeedNet~\cite{benaim2020speednet}	K400	S3D-G	9.1M	64	V	81.1	48.8
	VTHCL~\cite{yang2020video}	K400	R-50, Slow pathway	31.8M	8	V	82.1	49.2
	CoCLR \cite{Han20}	K400	S3D-G	9.1M	32	V	{87.9}	54.6
	CoCLR Two-Stream \cite{Han20}	K400	2\x S3D-G	2\x9.1M	32	V	{90.6}	{62.9}
	\bf{\oursshort}	K400	R-50, Slow pathway	31.8M	8	V	92.2	63.1
	\bf{\oursshort}	K400	S3D-G	9.1M	32	V	\bf{93.8}	\textbf{66.4}

MoCo [23] that uses RGB and optical-flow input in training, **MvPL** provides a significant performance gain of **+5.9%** and **+11.4%** top-1 accuracy on UCF101 and HMDB51, using the *identical* backbone (S3D-G) and data, and even surpasses CoCLR [22] by +3.2% and +3.5% top-1 accuracy when CoCLR is using Two-Streams of S3D-G for inference.

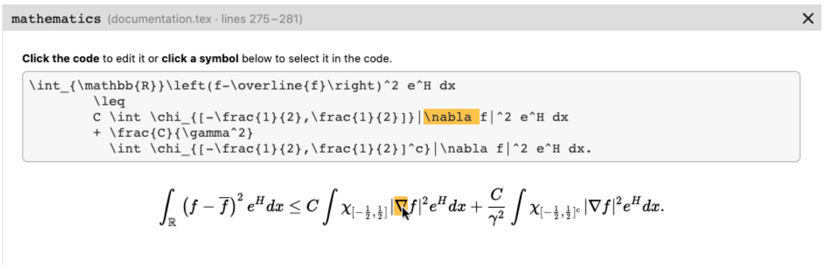
Discussion. We believe this is a very encouraging result. In the image classification domain, semi-supervised and self-supervised approaches compare even-handed; *e.g.* see Table 7 in [8] where *a self-supervised approach* (SimCLR) *outperforms all semi-supervised approaches* (*e.g.* Pseudo-label, UDA, FixMatch). In contrast, our state-of-the-art result suggests that for video understanding semi-supervised learning is a promising avenue for future research. This is especially notable given the flurry of research activity in self-supervised learning from video in this setting [1, 42, 44,

on a 1% and a 10% fraction of UCF101 corresponds to the number of iterations that 48 and 240 epochs on 100% of UCF101 would take respectively (note we use $\mu = 3$ for K400 and $\mu = 4$ for UCF101, where μ is the ratio to balance the number of labeled and unlabeled data).

The learning rate is linearly annealed for the first 34 epochs [19]. We follow the learning rate schedule used in [16] with a half-period cosine schedule [39]. In particular, the learning rate at the n-th iteration is $\eta \cdot 0.5[\cos(\frac{n}{n_{\max}}\pi)+1]$, where $n_{\max}$ is the maximum training iterations and the base learning rate η is 0.8. We use the initialization in [24].

We adopt synchronized SGD optimization in 64 GPUs following the recipe in [19]. We train with Batch Normalization (BN) [28], and the BN statistics are computed within the clips that are on the same GPU. We use momentum of 0.9 and SGD weight decay of 10^{-4}. Dropout [52] of 0.5 is used

Figure 6.3. User interface of *i*-LaTeX when a transitional has been displayed. ❶ Text editor of Visual Studio Code. ❷ Generated PDF. ❸ Transitional representation of the code of a table displayed on top of the PDF, just below the table that has been clicked. ❹ Textual representation of the code displayed in the transitional. The document was adapted from the source code of an article authored by Xiong et al. (2021) by replacing all the `tabular` environments by `itabular`.



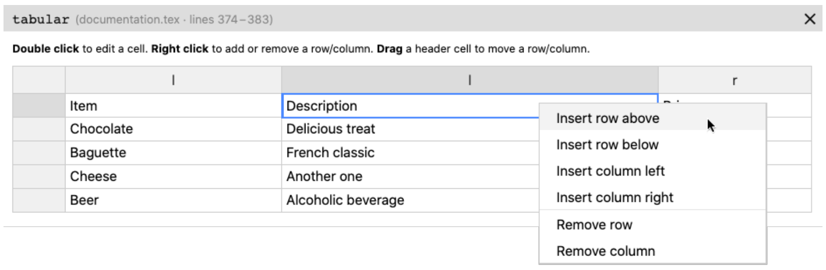
mathematics (documentation.tex · lines 275 – 281)

Click the code to edit it or **click a symbol** below to select it in the code.

```
\int_{\mathbb{R}}\left(f-\overline{f}\right)^2 e^H dx
        \leq
        C \int \chi_{[-\frac{1}{2},\frac{1}{2}]}|\nabla f|^2 e^H dx
        + \frac{C}{\gamma^2}
          \int \chi_{[-\frac{1}{2},\frac{1}{2}]^c}|\nabla f|^2 e^H dx.
```

$$\int_{\mathbb{R}} \left(f - \overline{f}\right)^2 e^H dx \leq C \int \chi_{[-\frac{1}{2},\frac{1}{2}]} |\nabla f|^2 e^H dx + \frac{C}{\gamma^2} \int \chi_{[-\frac{1}{2},\frac{1}{2}]^c} |\nabla f|^2 e^H dx.$$

a. Mathematical formula.

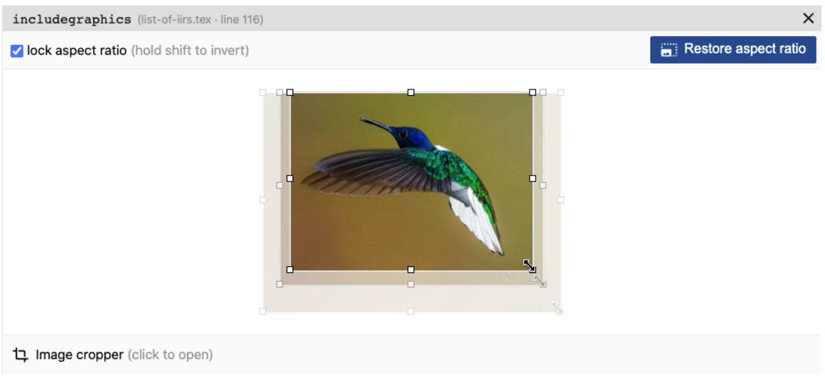


tabular (documentation.tex · lines 374 – 383)

Double click to edit a cell. **Right click** to add or remove a row/column. **Drag** a header cell to move a row/column.

	l	l	r
	Item	Description	
	Chocolate	Delicious treat	
	Baguette	French classic	
	Cheese	Another one	
	Beer	Alcoholic beverage	

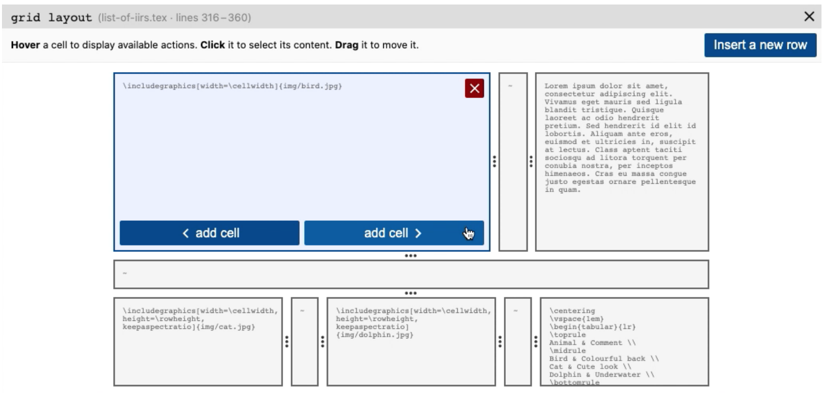
b. Table.



includegraphics (list-of-iirs.tex · line 116)

lock aspect ratio (hold shift to invert)
Restore aspect ratio
Image cropper (click to open)

c. Image.



grid layout (list-of-iirs.tex · lines 316 – 360)

Hover a cell to display available actions. **Click** it to select its content. **Drag** it to move it.

```
\includegraphics[width=\cellwidth]{img/bird.jpg}
```

```
~
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
Vivamus eget mauris sed ligula
blandit tristique. Quisque
laoreet ac odio hendrerit
pretium. Sed hendrerit id elit id
lobortis. Aliquam ante eros,
euismod et ultricies in, suscipit
at lectus. Class aptent taciti
sociosqu ad litora torquent per
conubia nostra, per inceptos
himenaeos. Cras eu massa congue
justo egestas ornare pellentesque
in quam.
```

```
~
```

```
\includegraphics[width=\cellwidth,
height=\rowheight,
keepaspectratio]{img/cat.jpg}
```

```
~
```

```
\includegraphics[width=\cellwidth,
height=\rowheight,
keepaspectratio]
{img/dolphin.jpg}
```

```
~
```

```
\centering
\vspace{1em}
\begin{tabular}{lr}
\toprule
Animal & Comment \\
\midrule
Bird & Colourful back \\
Cat & Cute look \\
Dolphin & Underwater \\
\bottomrule
```

d. Grid layout.

Figure 6.4. User interfaces of the four transitionals available in *i*-LATEX, showing examples of how they can be used. (a) Hovering over the ∇ symbol highlights the corresponding macro in the code. (b) Right-clicking a cell displays a contextual menu that enables to insert and delete rows and columns. (c) Dragging a handle resizes the image while preserving the same aspect ratio. (d) Hovering over a cell displays buttons for inserting adjacent cells or deleting the cell.

visualise invisible structures and abstractions and (2) modify the source code of the document in a more interactive way.

Depending on the position of the clicked element on the screen, the transitional is displayed in a panel either above or below it so as to leave the rendered element as visible as possible. The rest of the document output is darkened until the transitional is closed by clicking on the cross at the top-right of the panel or anywhere on the darkened document. Closing the transitional also recompiles the LaTeX document and updates the PDF.

The title bar of each transitional displays the name of the file and the range of the code that is visualised. Clicking on it displays the code in the code editor by opening the appropriate file if needed and scrolling to the relevant section of the code. The user can edit the code directly in the code editor. The code is re-parsed after every keystroke, to update the visualisation. If an error is introduced, the piece of code is highlighted in red in the code editor, and the visualisation is replaced by an error message that invites the user to fix the problem. The visualisation is restored as soon as the error is fixed.

6.4.2 TRANSITIONALS

I implemented four kinds of transitional in *i*-LaTeX: three for standard LaTeX structures (mathematics, images, tables), and one for a custom grid layout. They are shown in action in Figure 6.4. These transitionals mainly support semantic interaction with the code, as they target specific concepts and help users directly manipulate some constituents of these concepts, such as rows and columns in grid-like structures. To some extent, they can also be considered to support graphemic interaction, e.g., by letting users view and edit the content of grid cells as text, and pragmatic interaction, e.g., by helping users preview what a mathematical formula that was modified or an image that was resized will look like in the compiled document.

Mathematics

Interactive mathematical formulae can be added to the document with the `imaths` environment, a wrapper around the `align*` environment.[55] The interactive visualisation of a formula (Figure 6.4a) displays an editable copy of the code of the formula along with the typeset formula. Hovering over a symbol or a group of symbols in the typeset formula—such as a fraction—highlights the piece of code that produced it, and clicking on it selects that piece of code. Editing the copy of the code instantly updates the typeset formula in the transitional. If an error is detected, an error message is instantly displayed in the visualisation, so that the user can fix the code of the formula without having to recompile the whole document. This transitional provides a strong link between the code of a formula and its output (R4), while acknowledging that users may prefer editing the code as text rather than directly manipulating the formula (R1).[56]

55. As provided by the `amsmath` package (@144).

56. Even popular word processors such as Microsoft Word support a LaTeX-like syntax for writing mathematical formulae as text—in spite of also providing a user interface for writing formulae with the help of menus and buttons—, therefore suggesting that writing mathematical formulae as text is a widespread practice, including amongst non-expert LaTeX users.

Tables

Interactive tables can be added with the `itabular` environment, with the same syntax as the standard `tabular` environment. The interactive visualisation of a table (Figure 6.4b) displays the code of the table in a grid, as well as the type of each column in the header row. The raw content of each cell can be selected in the code by clicking and edited by double-clicking. Columns and rows can be inserted and deleted via a contextual menu as well as reordered by dragging their respective headers. This transitional lets users see and manipulate the table structure that is usually only visible in the PDF (R3), making common transformations such as inserting and rearranging rows and columns much easier than with a code editor (R2). By displaying the raw content of the table, the user is free to use arbitrary LaTeX code within each cell (R1).

Images

Interactive images can be added with the `\iincludegraphics` macro, with the same syntax as the standard `\includegraphics` macro.[57] The interactive visualisation of an image (Figure 6.4c) displays it at the same size as in the PDF. The image can be resized by dragging one of the handles. Clicking a button displays a cropper that lets the user select the region of the image to display. These manipulations automatically update the parameters of the macro by inserting, modifying, or deleting the `width`, `height`, `trim` and `clip` options. This transitional facilitates visualising and formalising dimensions (R3) and helps discover and use lesser-known macro options such as cropping (R2).

57. As provided by the `graphicx` package (@145).

Grid layouts

Interactive grid layouts can be added using the `gridlayout` environment, a custom environment I developed specifically for *i*-LaTeX. It consists of `minipage` environments arranged in a fixed-size area made up of rows of cells parameterised with relative dimensions to support local positioning of various types of content (text, images, tables, etc). The interactive visualisation of a grid layout (Figure 6.4d) displays the otherwise invisible grid-like structure. Cells can be resized by dragging a separator between two cells, reordered by dragging a cell, as well as inserted and deleted by clicking the appropriate button while hovering over a cell. Rows can be resized in a similar fashion, and a new row can be appended at the end of the grid by clicking a button. As with interactive tables, each cell displays its raw LaTeX content. It cannot be edited directly from the visualisation in the current version, but clicking on a cell selects its content in the code editor. This transitional supports the editing of structured content (R3) and the concrete representation of abstractions such as relative dimensions (R4). It also illustrates that transitionals may foster the development of new LaTeX environments that would otherwise be too difficult to use with raw code only.

6.4.3 IMPLEMENTATION

i-LaTeX is implemented as an extension for Visual Studio Code (@146) written in TypeScript, along with HTML and CSS for the user interface. The source code of the extension is open-source and has been made available on GitHub (@147). I present the key aspects of the implementation below.

Providing custom macros and environments

In order to use the special macros and environments presented above to create elements that can be visualised and manipulated through transitionals, a custom `ilatex` package must be included in the preamble of the document. Each use of one of these macros/environments is associated with a unique identifier that is written to an external file of *code mappings*, along with other metadata such as the location in the code (file path and line number) and the current values of several length macros such as `\textwidth`, so that lengths using them can be evaluated by *i*-LaTeX. In addition, a PDF annotation containing the same unique identifier is added to the generated PDF, with the same bounding box as the element produced by the macro/environment. Although I created custom macros and environments for the sake of simplicity, the existing ones they rely upon—such as `\includegraphics`—could be patched to behave in the way I just described, therefore enabling LaTeX users to benefit from transitionals without having to learn new macros/environments.

Extracting the code to visualise

Unlike most programming languages, LaTeX has no predefined grammar (Erdweg and Ostermann, 2011). Instead, it uses some unique features, such as TeX's category codes (Knuth, 1984b, ch. 7), which enable to modify the lexical meaning of every character (such as \ denoting the start of a macro) anywhere in the document—therefore making LaTeX theoretically impossible to parse using conventional parser generators.[58] Nevertheless, certain conventions are very commonly used, such as the structure of environments. In order to extract the pieces of code to visualise, I wrote a LaTeX parser that accepts a reasonable proportion of documents that follow these conventions. Every time the document is compiled, *i*-LaTeX reads and parses every file whose path exists in the file of code mappings into an abstract syntax tree (AST). For each code mapping, it then attempts to find the corresponding piece of code in the given file, at the given line and of the given type, and creates a model of the transitional with the matching AST node. The parser is designed to be simple enough to minimise the number of parsing errors and the execution time. Each transitional model can perform a more thorough analysis of its own AST node if necessary.

58. See a related thread on StackExchange (@148) for more details on this limitation.

Displaying the augmented PDF

Once the document has been compiled into an annotated PDF, it is displayed using a custom PDF renderer.[59] The renderer extracts all the annotations inserted by the custom macros/environments along with their unique identi-

59. The renderer is based on the PDF.js library (@149).

fiers and uses them to add a blue halo to every element of the PDF whose code can be visualised. When one of these elements is clicked, *i*-LaTeX matches the unique identifier of the element with the correct model. If a matching model is found, it is used to populate the view of the transitional with the appropriate data that was extracted by the model, such as the content of each cell of a table. When the user interacts with a transitional, the view notifies the controller of every action of interest. The latter forwards them to the model, which is responsible for modifying the code of the LaTeX document. Every time the code is modified by a visualisation model or by the user, the AST of the file is updated, and every model whose AST is modified updates its internal representation of the code and provides new data to the view.

6.4.4 LIMITATIONS

Features

A first limitation of *i*-LaTeX is the fact that transitionals cannot interpret certain pieces of code even though their syntax may be valid and they may produce the expected result in the PDF. Such limitations could be addressed by (1) improving the static analysis of the code performed by the transitionals to extract more information and (2) developing new features in these transitionals to exploit that information. As an example, while merged cells are currently not supported by the transitional for tables, its model could be modified to process the `\multirow` and `\multicolumn` macros and the view could be modified to enable users to merge/unmerge cells interactively. However, because of LaTeX's extensible nature, there is no way to ensure that all the features available as code will be available in a particular transitional.

Abstraction

A second limitation of *i*-LaTeX is the absence of support for transitionals that represent PDF elements generated by custom macros. Supporting this type of abstraction in *i*-LaTeX is challenging, because it requires to (1) identify the provenance (Williams and Gordon, 2021) of all the pieces of code that, put together, generate a certain PDF element and (2) deal with situations where a custom macro is used in multiple places, such as resizing an image inserted by a custom macro that is also used to insert the same image in other places. While some research prototypes make use of custom language interpreters designed to track the provenance of every value they compute, I could not readily use this approach in *i*-LaTeX since no LaTeX compiler currently tracks such information.[60] There is no consensus on how to solve the second challenge, which remains an open question for future work.

60. The difficulty of tracking the provenance of PDF elements generated by LaTeX is further discussed by Laurens (2007), who faced the same limitations regarding custom macros when developing SyncTeX (Laurens, 2008, sec. 5).

Performance

A third limitation of *i*-LaTeX is the lower performance on large LaTeX files. Since *i*-LaTeX updates the AST of a file that contains at least one transitional every time it is modified, transformations to perform in the code of a large file can accumulate. When too many edits are performed in a short amount of time, e.g., when an image is being rapidly resized, this accumulation can make *i*-LaTeX look jerky. In practice, performance is excellent for small to medium-size files, and larger documents can be split into multiple LaTeX files. For example, I could fluidly edit the source code of several long papers using *i*-LaTeX on a 2GHz MacBook Pro, such as the 750-lines long LaTeX file of the paper by Xiong et al. (2021) shown in Figure 6.3. In addition, the propagation of changes could be optimised to better support large ASTs.

6.4.5 EXTENSIBILITY

Although *i*-LaTeX is only malleable in the sense that it can be reprogrammed, it was conceived with a modular design that facilitates extension. Creating a new transitional requires creating (1) a model that can extract the information to be visualised from the AST node and make the necessary changes when it is modified, and (2) a view that represents the data provided by the model in the desired format. The source code provides controllers with an API for exchanging messages between the model and the view and registering callbacks for various events, as well as a number of utilities, such as a class for parsing, converting and manipulating LaTeX lengths and an API for operating on the AST. In addition, the `ilatex` package for LaTeX must also be modified to create—or patch—the LaTeX macro or environment that will benefit from the new transitional so that every time they are used, they behave as described in subsection 6.4.3. Creating transitional for pieces of code that are neither a macro nor an environment is also possible, but not as straightforward, as it may require adapting *i*-LaTeX's parser to create new types of AST nodes.

6.5 CONTROLLED EVALUATION

To assess whether transitionals can improve how LaTeX users perform a number of specific editing tasks that *i*-LaTeX's transitionals were designed to facilitate, I conducted a controlled experiment. I was interested in comparing quantitative metrics (task duration, number of compilations, participants' workload), as well as collecting qualitative observations and feedback on the use of *i*-LaTeX. In this section, I expose the methodology I used, present and analyse the results, and discuss the outcomes of the study.

6.5.1 METHODOLOGY

Participants

I recruited 16 participants (2 women and 14 men, age 20–65) by posting a message on the mailing lists of several computer science labs and a group of HCI practitioners, and on a Facebook group of university students and alumni. They did not receive any compensation for their participation. All participants had used LaTeX before. 5 participants had used it for less than 5 years, 8 participants for 5 to 10 years and the other 3 for more than 10 years. 3 participants had never used mathematical formulae and 1 had never used tables in LaTeX before. Slightly less than half the participants (7/16) self-ranked their overall expertise with LaTeX as 4 or 5 on a 5-point Likert scale.

Setup

The study was carried out remotely. Participants used *i*-LaTeX on their own computers and shared their screen. The study lasted between one and two hours per participant, including setup and debriefing.

Procedure

I used a 2×2 within-participant design with two independent variables, Transitionals (*Enabled*, *Disabled*) and Document (D_1, D_2). I adjusted the design of the study with two pilot participants to ensure it was understandable and not too long to complete.

For each participant, I started by explaining the steps of the study to the participant, asked them to read and sign the consent form, and helped them install the *i*-LaTeX extension in Visual Studio Code. I then asked participants to open, read, and edit an introductory LaTeX document with *i*-LaTeX. The document presented the features of *i*-LaTeX and the three transitionals used in the study (mathematics, tables, and images), with one interactive example per transitional. I also invited participants to ask questions about *i*-LaTeX or the study.

Once they were ready, I asked participants to perform a series of 9 tasks (T1–T9), as fast as possible, on one of two similar LaTeX document sets, D_1 and D_2. Participants had to perform the 9 tasks with the first document set in one of the Transitionals conditions, and again with the other document set in the other Transitionals condition. Each document set consisted of three LaTeX documents (one for each type of tasks). The first document contained tasks T1–T3 (maths); the second document tasks T4–T6 (tables); and the third document tasks T7–T9 (images). In the condition where transitionals were enabled, I explicitly told the participants that they were not required to use them, and could always edit the code directly if they believed it was faster. After completing all the tasks in a document set, participants had to fill in a workload questionnaire based on the NASA-TLX.[61]. The order of the two document sets and the two conditions was counterbalanced across participants. The four configurations of the experiment are presented in Figure 6.5.

61. I adapted five out of the six measures of the original NASA-TLX questionnaire, with no weighting process—a variant called the *raw* NASA-TLX questionnaire (Hart, 2006).

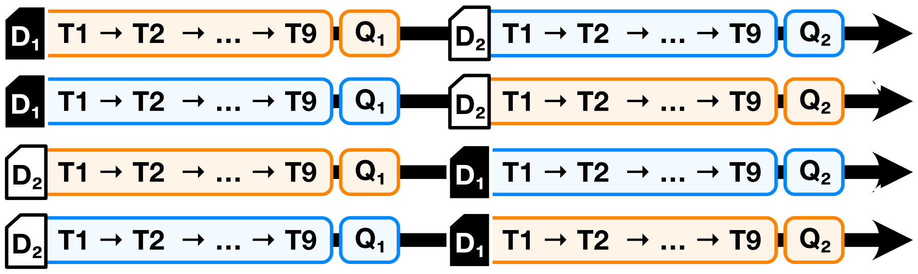
Once all the tasks were completed, I asked the participants to fill in a post-study questionnaire. I also debriefed them about their experience with *i*-LaTeX and answered their questions. Moreover, if participants were interested, they were offered to try the transitional for grid layouts (using a LaTeX document specially provided for the task) and give feedback on their experience.

Tasks

The nine tasks were similar in each of the two document sets (Table 6.2). Each task fits on a single page of the generated PDF that contains (1) the instructions and (2) the current output of the code to modify. The tasks were designed so that they could be completed in at most a few minutes. In order to move to the next task, participants had to compile the document with no error, and the generated PDF had to contain the expected result. Participants

Task	Type of content	Type of instruction
T1	Mathematics	Insert a term in a multi-line formula
T2	Mathematics	Remove parentheses around a term in a multi-line formula
T3	Mathematics	Modify a term in one formula among six
T4	Tables	Sort the rows of a table by a certain column
T5	Tables	Modify the values of specific cells in a table
T6	Tables	Remove a specific column from a table
T7	Images	Resize an image to make it as wide as another element
T8	Images	Remove the whitespace that surrounds an image
T9	Images	Hide a region of an image

Table 6.2. Description of the tasks participants were asked to perform. The tasks were grouped three by three, each group focusing on a single type of content to edit: mathematical formulae (T1–T3), tables (T4–T6) and images (T7–T9).



D1 T1 → T2 → ... → T9 Q1 D2 T1 → T2 → ... → T9 Q2
D1 T1 → T2 → ... → T9 Q1 D2 T1 → T2 → ... → T9 Q2
D2 T1 → T2 → ... → T9 Q1 D1 T1 → T2 → ... → T9 Q2
D2 T1 → T2 → ... → T9 Q1 D1 T1 → T2 → ... → T9 Q2

Figure 6.5. Scheme of the four configurations of the experiment. The colour of the blocks represent whether transitionals are 🟦 enabled or 🟧 disabled. D1 and D2 represent the two document sets, and Q_1 and Q_2 represent the two workload questionnaires.

were allowed to use external resources to complete the tasks, including online searches and other programs, as long as they did not reuse LaTeX code from other files of the study.

Most of the tasks were inspired by issues mentioned during the interviews, such as finding symbols in complex formulae and editing large tables, that I adapted to ensure that all tasks could be solved with transitionals. I decided not to include tasks with grid layouts after testing them in a pilot study, as using the `gridlayout` environment without transitionals confused participants and made the study last more than two hours.

Data collection

I recorded the screen and the audio of the participants and took notes of the strategies they used and the difficulties they faced. At the end of the study, I collected the log files generated by *i*-LaTeX on the participants' computers.

Data analysis

For each group of tasks, I measured the task completion times (*TIME*) and the number of compilation (*COMPILATIONS*) by processing the collected files using Python, R and SAS JMP. I also reviewed the participants' answers to the three questionnaires and watched parts of the recordings to compare their behaviours and collect examples of strategies they used to solve the tasks.

I eliminated data from one participant for the tasks with images, for both conditions, because that participant had to leave and could not solve tasks T7–T9 after spending more than 30 minutes trying without transitionals (with transitionals, this participant completed tasks T7–T9 in about 14 minutes).

Performance

A mixed ANOVA showed no significant effect of the order of the two TRANSITIONALS conditions ($F_{1,12} = 2.83$, $p = 0.12$) nor of the order of the two document sets ($F_{1,12} = 0.05$, $p = 0.82$) on *TIME*. Thereafter, I therefore ignore these two order factors. Since both task duration times (*TIME*) and numbers of compilation (*COMPILATIONS*) are strictly positive measures, I tested the log-normality of the distribution of each measure with Kolmogorov's D tests. Task completion times fit log-normal distributions for tasks with tables ($D = 0.127$, $p > 0.15$) and images ($D = 0.151$, $p = 0.06$), but not for tasks with mathematics ($D = 0.173$, $p = 0.02$). Numbers of compilation fit a log-normal distribution for tasks with images ($D = 0.15$, $p = 0.07$), but not for tasks with mathematics ($D = 0.266$, $p < 0.01$) or tables ($D = 0.219$, $p < 0.01$). Given these results, I performed paired t-tests on log-transformed data for task duration time and Wilcoxon signed-rank tests for the numbers of compilations. I also report effect sizes using Cohen's d for t-tests and Rank-biserial correlation (RBC) for Wilcoxon signed-rank tests. The detailed results of all tests are reported in Table 6.3.

Regarding task completion time (Figure 6.7), there is a significant effect of TRANSITIONALS on *TIME* for tasks with tables ($t_{15} = -4.95$, $p < 0.001$, $d = 1.39$) and images ($t_{14} = -3.75$, $p = 0.002$, $d = 1.17$), but not for mathematics ($t_{15} = -1.45$, $p = 0.17$, $d = 0.34$). According to mean task completion times, tasks were performed 44% faster when transitionals were enabled (42% faster for tasks with tables, 58% faster for tasks with images). Regarding the number of compilations (Figure 6.8), there is a significant effect of TRANSITIONALS on *COMPILATIONS* for tasks with tables ($W = 8$, $p = 0.008$, RBC = -0.82) and images ($W = 6$, $p = 0.001$, RBC = -0.90), but not for mathematics ($W = 38$, $p = 0.97$, RBC = -0.03). According to mean numbers of compilation, participants compiled 41% less often when transitionals were enabled (26% less often for tasks with tables, 58% less often for tasks with images).

In order to look further into the effect of the transitional for mathematics, I split participants into two groups based on their efficiency, defined as a combination of high speed (low task completion times) and high precision (low variance between task completion times). To form the groups, I plotted the distribution of task completion times per participant (Figure 6.9) and the mean task completion time of each participant against the standard deviation of the task completion times (Figure 6.10). The plots reveal that 6 participants form a distinct cluster (mean task completion time shorter than 5 minutes, standard deviation of task completion times lower than 2 minutes), which I distinguish as the *efficient* group (P2, P4, P8, P10, P12 and P15). The 10 other participants form the *non-efficient* group. These groups are consistent with the self-assessed levels of expertise with LaTeX collected in the post-study

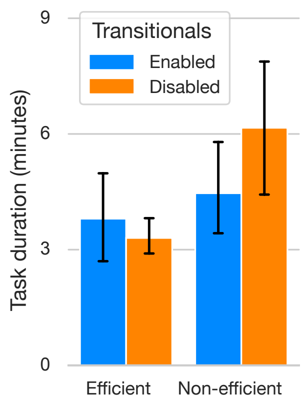


Figure 6.6. Effect of transitionals on task completion time for tasks with mathematics (*TIME* for T1–T3) for each efficiency group. Error bars represent 95% CIs.

questionnaires (Table 6.4), whose mean and median values are higher for *efficient* participants.

Results comparing the two efficiency groups for tasks with mathematics are reported in Table 6.3. Plotted means (Figure 6.6) and paired t-tests for *efficient* ($t_5 = 0.61$, $p = 0.57$, $d = 0.28$) and *non-efficient* ($t_9 = -2.37$, $p = 0.04$, $d = 0.64$) participants suggest that having access to transitionals has a different effect on each of the two groups. However, this difference cannot be considered significant after correcting p-values as per the Holm-Bonferroni method. Given the small sample size, the tests have a low power and the observed trend should therefore be tested with a larger sample.

Workload and feedback

The answers to the questionnaires are in line with the quantitative analysis. According to both the two post-condition questionnaires (Figure 6.11) and the post-study questionnaire (Figure 6.12), participants experienced a lower workload and consider that they performed better when they had access to transitionals. A large majority of participants reported that having access to transitionals to complete the tasks was less mentally demanding (16/16), less temporally demanding (15/16), less frustrating (13/16), and made them achieve better performance (13/16). A few participants (3/16) reported a slightly higher physical demand when they had access to transitionals, which some participants related to the increased use of the mouse required to interact with the transitionals. Several participants qualified the tool as "*very impressive*", and all the participants reported that they would *probably* (5/16) or *certainly* (11/16) use transitionals if they were available in their LaTeX editor.

Most participants made positive comments and suggestions to improve *i*-LaTeX. The suggestions include new features, mainly for tables, such as merging cells, resizing columns, manipulating row separators, and enabling multi-row or multi-column selections, as well as transitionals for other types of elements such as TikZ drawings and citations. Several participants were frustrated that the transitional would hide a part of the document they were interested in, and suggested to let users move transitionals up and down. Some participants also suggested to add a way to close a transitional without recompiling the document.

At the end of the study, all but two participants agreed to try the transitional for grid layouts. Their reaction was mostly positive. While some commented that the transitional currently lacks some features they would like to use, such as grouping cells by column, previewing the output of the cells' content, and equally distributing the available width/height, they also noted that it was already better than the solutions they use to locally position elements in LaTeX.

Strategies

I noticed that participants used different strategies depending on whether they had access to transitionals or not. When they were only allowed to edit the code as text, several participants had to search either online (10/16) or in a document/book (2/16) to solve some of the tasks. Two participants copy-

Type of tasks	**Participants**	**Paired t-tests on log(*TIME*)**					**Wilcoxon tests on *COMPILATIONS***			
		#DoF	t	p		d	W	p		RBC
Mathematics	All	15	−1.45	0.17		0.34	38	0.97		−0.03
	Efficient	5	0.61	0.57		0.28	5	1.00		0.00
	Non-efficient	9	−2.37	0.04		0.64	16	0.83		−0.11
Tables	All	15	−4.95	< 0.001	*	1.39	8	0.008	*	−0.82
Images	All	14	−3.75	0.002	*	1.17	6	0.001	*	−0.90

Table 6.3. Comparisons of task completion times and number of compilations between the two conditions, for each type of task, and for each subset of participants when applicable. Log-transformed task completion times are compared using paired t-tests and Cohen's d for effect sizes. Number of compilations are compared using Wilcoxon signed-rank tests and rank-biserial correlation (RBC) for effect sizes. Tests where $p < 0.05$ are marked with asterisks.

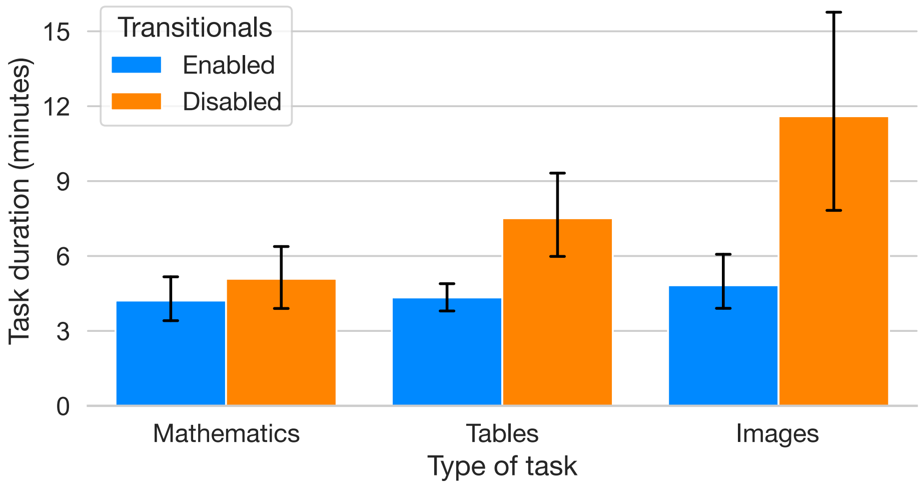


Figure 6.7. Effect of transitionals on task completion time (*TIME* per task type). Error bars represent 95% CIs.

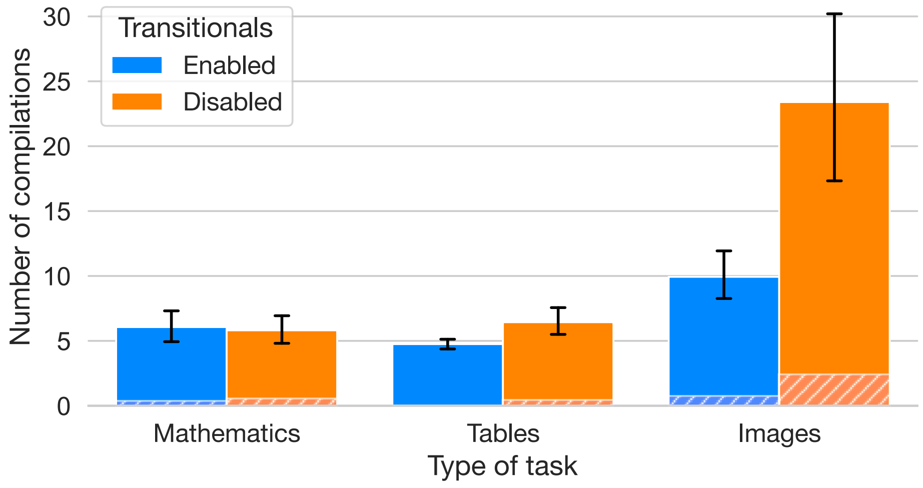


Figure 6.8. Effect of transitionals on number of compilations (*COMPILATIONS* per task type). Hatched bars represent failed compilations. Error bars represent 95% CIs.

	Mean	Median	Minimum	Maximum
Efficient participants	3.8	4	3	5
Non-efficient participants	2.8	3	1	5

Table 6.4. Statistics on the participants' self-assessed expertise with LaTeX on a 5-point Likert scale (1 = Beginner, 3 = Intermediate, 5 = Expert).

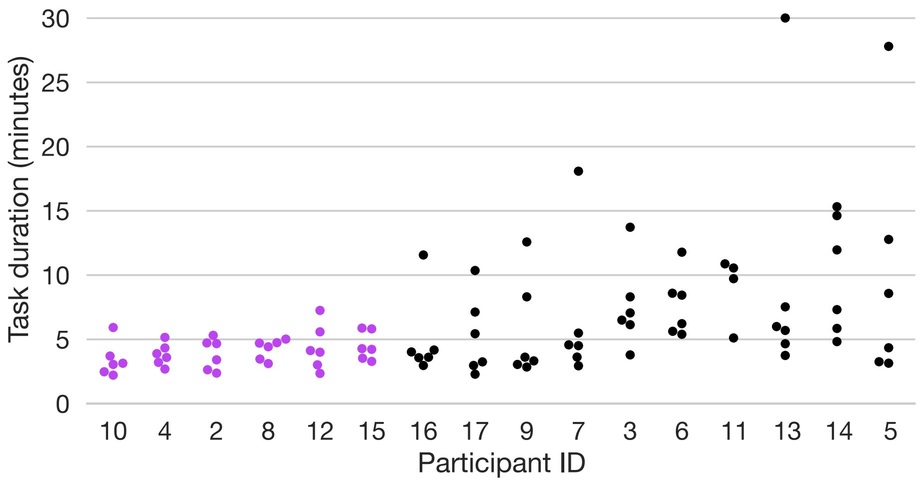


Figure 6.9. Distribution of task completion times per participant. Each data point corresponds to a group of tasks (mathematics, tables or images) and a condition (with transitionals, without transitionals). Participants are sorted by their mean task completion time over both conditions. Efficient participants correspond to the six participants with the lowest mean task completion time (shown in purple), whose distribution of task completion times are notably less sparse than those of the other participants.

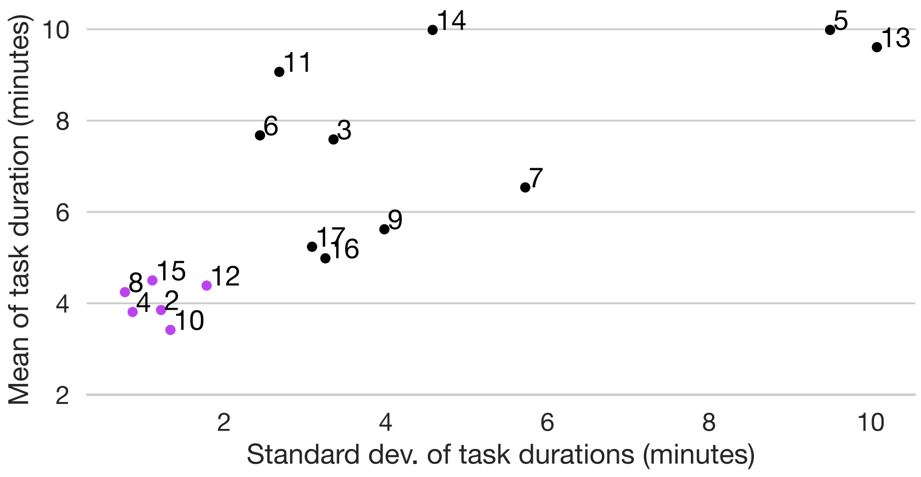


Figure 6.10. Plot of the mean task completion time (Y axis) against the standard deviation of task completion times (X axis) of each participant. Each mark is labelled with the ID of the participant. Efficient participants correspond to the six participants in the bottom-left hand corner (shown in purple), who are notably faster (low Y value) and more consistent (low X value) than other participants.

pasted code into Emacs to sort table rows in task T4, and four participants used an image editor to crop the images in tasks T8 and T9. Some participants also tried to come up with elaborate solutions, including computing the size of an element in LaTeX, measuring an image displayed on their screen with a ruler, and playing with negative whitespace. In these situations, most participants eventually admitted that they could not achieve what they had in mind after a few minutes of trying, often resolving to simpler solutions and approximation by trial-and-error.

I did not observe such behaviours when participants were allowed to use transitionals. In this condition, although most participants (13/16) edited the code as text at some point during the study, the majority of the edits were performed through a transitional. Two participants used the code editor to search for values to replace in T5; four participants approached the expected image width using the transitional and fine-tuned the value by editing the code in T7; and seven participants used the code editor to fix errors introduced by *i*-LaTeX when resizing or cropping images in T7 and T9.[62]

62. Due to a bug in the implementation of *i*-LaTeX used in the study, fast successive changes sometimes caused transitionals to become out-of-sync with the code, resulting in erroneous code generation.

No participants attempted to find "clever" solutions when transitionals were available, with one exception. One participant completed task T4 very efficiently by using both the transitional and the code editor. He opened the transitional to move the column with the values to sort by to the left of the table, switched to the editor to select the code of all the rows in the editor, triggered a command to sort the selected lines, and switched back to the transitional to move the column back to its original position.

6.5.3 DISCUSSION

The results of the study show that participants solved common tasks with tables and images 44% faster and recompiled the document 41% less often when they had access to transitionals, with large effect sizes ($d > 0.8$ for t-tests, $|\mathrm{RBC}| > 0.8$ for Wilcoxon tests). While the difference is not significant for tasks with mathematics, it suggests that transitionals may be more beneficial to the least efficient participants. This might be explained by the higher proficiency and experience with the syntax of mathematics in LaTeX of the most efficient participants, who might be more used to, e.g., finding the location of a certain symbol in the code (for tasks T1 and T3), and remembering to delete both `\left` and `\right` commands along with parentheses (for task T2).

In addition to improving performance, transitionals helped participants solve tasks with a lower workload and using more straightforward strategies. I hypothesise that this difference mainly stems from two characteristics of *i*-LaTeX's transitionals. The first characteristic is that transitionals enable to modify the code of the document by interacting with a possible mental model of the code, without requiring participants to (1) build their own mental model of the code and (2) map changes in their mental models to changes in the code. The second characteristic is that transitionals can help discover and use features that participants were not always familiar with, such as cropping an image directly via the `\includegraphics` command, which reduced the need for searching for tutorials and documentation. Interestingly, these characteristics may also encourage participants to solve tasks in more

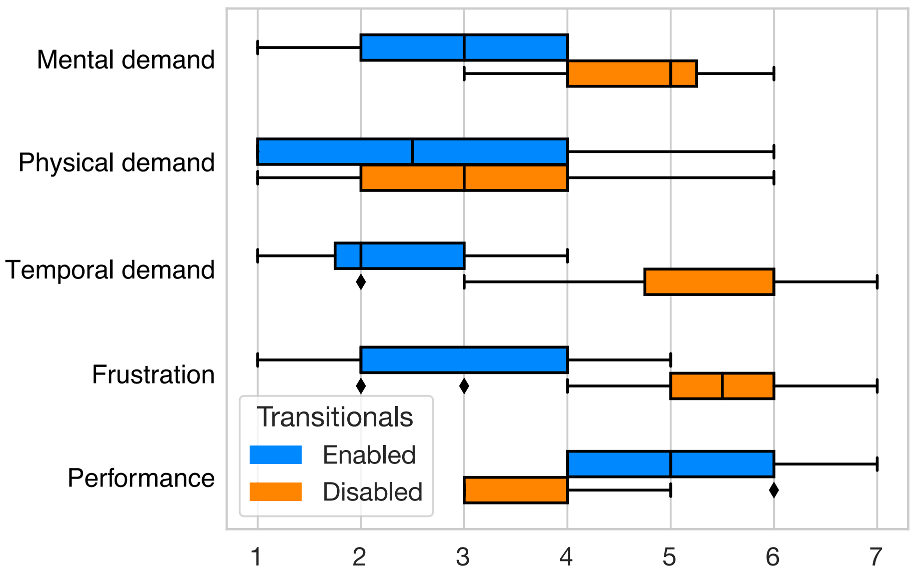


Figure 6.11. Effect of transitionals on participant workload, as reported by participants after each condition. Participants were asked to rate the workload of the condition they just completed on 7-point Likert scales (1 = very little, 4 = normal, 7 = very much).

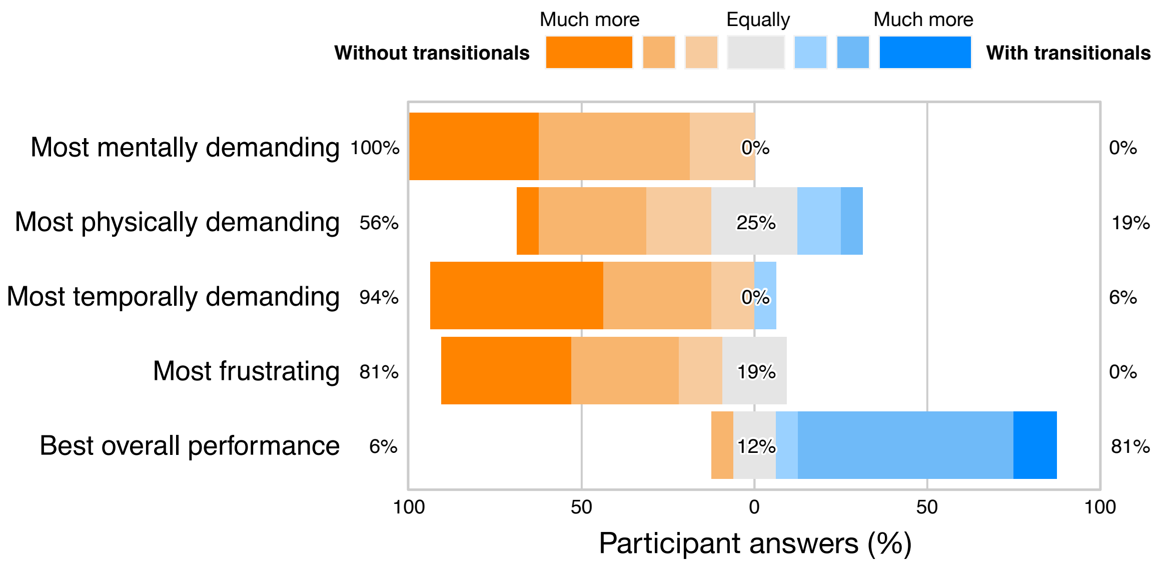


Figure 6.12. Comparison of the two conditions in terms of workload, as reported by participants at the end of the study. The five scales are the same as those used in the post-condition questionnaires, but participants were asked how much one condition applies more than the other instead—e.g., *"Which condition was the most mentally demanding for you?"*—using symmetrical 7-point Likert scales (1 = much more the 1st condition, 4 = equally, 7 = much more the 2nd condition).

direct ways, without resorting to tools designed to automate sub-tasks such as sorting lines and searching and replacing text.

In summary, this study shows that transitionals can be useful to beginner and expert LaTeX users alike. Transitionals can be used by beginners to learn about common LaTeX commands and environments and try alternatives for, e.g., mathematical symbols, column orders, or image sizes, without the time-consuming burden of recompiling after every change. Since transitionals are optional by design, expert users can freely decide if they prefer to use them or to edit the code directly. As an example, they could use a transitional to make changes in a table only when it is large enough, or to find the command that produced a symbol in a formulae when they cannot readily find it in the code.

6.6 LONGITUDINAL EVALUATION

The controlled experiment showed that transitionals can help LaTeX users solve specific editing tasks. Yet, the very design of the experiment did not let users appropriate the tool and use it with their own document, for their own goals, at their own pace. To further investigate how LaTeX users would use transitionals in a more ecological setting, free of any instruction and restriction, I conducted a longitudinal evaluation during the summer of 2021. I was interested in understanding how LaTeX users would bring *i*-LaTeX into their workflow and collect feedback on what features they use, when, and why. In this section, I expose the methodology I used, analyse the findings, discuss how this study complements the controlled experiment, and explain why it directed me to a new research direction.

6.6.1 METHODOLOGY

Participants

I recruited 6 participants (6 men, age 20–39) by posting a message on the mailing list of a computer science lab and recontacting participants of the controlled experiment who explicitly expressed interest in using *i*-LaTeX to edit their own documents. They did not receive any compensation for their participation. 3 participants were computer science researchers, 1 was a Ph.D. student in computer science, and 2 were M.Sc. students in biology. All participants had used LaTeX before, with an experience ranging from 3 to 18 years of use. All participants reported writing multiple LaTeX documents every year—two participant even reporting writing more than ten every year. All of them reported using images in their documents, and almost all (5/6) reported using mathematics and tables as well.

Setup

The study was carried out remotely. Participants used *i*-LaTeX on their own computers. The study lasted three months in total, but apart from the initial setup, participants were free to decide if and when they would like to use *i*-LaTeX.

Procedure

Before the start of the study, I explained participants the purpose of the study, presented the features of *i*-LaTeX, and asked them to read and sign the consent form. I asked them to fill in a pre-study questionnaire to collect demographics and information about their use of LaTeX, and helped them install the *i*-LaTeX extension in Visual Studio Code, which they were asked to pre-install on their computer along with a full LaTeX distribution. I then asked them to try it on a demonstration document provided to them, which contained code compatible with the four transitionals available in *i*-LaTeX, so that participant could try each of them and ask questions. I further reminded participants that they were free to decide if and when they would like to use *i*-LaTeX, and could reach out to me at any time during the study if they had questions or faced issues with the software. During the entire duration of the study, I contacted participants every week by email and asked them to fill in a questionnaire to briefly report on whether they authored LaTeX documents during the week, whether they used *i*-LaTeX to that end, and, if applicable, why and how.

Data collection

Before and during the study, I collected the participant's answers to the questionnaires. At the end of the study, I collected the log files generated by *i*-LaTeX on the participants' computers every time they used it to edit a LaTeX document.

Data analysis

I measured when participant used *i*-LaTeX and what features they used by processing the collected log files using Python. I also reviewed the participants' answers to the weekly questionnaires to verify and explain the patterns of use observed in the logs.

6.6.2 RESULTS

According to the logged data and the questionnaire answers, only participants P1, P3 and P6 used *i*-LaTeX during the study (Figure 6.13). Among them, only P1 and P6 used some of *i*-LaTeX's transitionals, and no participant used the transitional for mathematical formulae (Figure 6.14).

Participant P4 and P5 respectively answered the questionnaire only 5 and 4 times during the first two months, reporting no editing of any LaTeX document at all. Both ceased to answer starting from the 9th week, and neither answered me when I contacted them to collect log files at the end of the study, suggesting that they decided to stop participating in the study altogether. In the rest of the analysis, I therefore focus on the activity of the other four participants.

On 9 weeks out of 12, P2 reported that he edited one to two LaTeX documents, sometimes for more than 8 hours a week, but never used *i*-LaTeX to perform the edits. He once commented that he was working on an article on Overleaf, just before a deadline, and was mostly writing text, though he also authored a figure using TikZ, possibly suggesting that *i*-LaTeX would not have been of any help and/or incompatible with the collaborative nature of his work. On three other weeks, he commented on not using transitionals for images because he was using TikZ, adding that "*help with this in i-Latex would be appreciated*" (P2). The rest of the time, he did not further explain why he did not use *i*-LaTeX.

Similarly to P2, P3 reported that he edited LaTeX documents during 8 different weeks, often for 3 to 8 hours a week. Contrary to P2, he did use *i*-LaTeX on three occasions, though he never explained why or why not. According to the log data, P3 only used *i*-LaTeX to edit text, compile and preview the PDF document, but never used any transitional, in accordance with his questionnaire answers, suggesting that he used *i*-LaTeX merely as a regular LaTeX editing environment.

P1 only used *i*-LaTeX a few times, often explaining that since he was mostly writing text, he preferred to keep using his regular LaTeX editor, Overleaf. He mainly used *i*-LaTeX for editing tables, commenting that "*editing and copy/pasting cells with iTable is really cool*", though "*duplicating or inserting many rows at once is faster by editing the code*" (P1). One another week, he also commented using *i*-LaTeX only to preview the generated PDF in Visual Studio Code.

P6 was the most enthusiast user of *i*-LaTeX in this study. Although he only used it during the last few weeks of the study,[63] he mainly used *i*-LaTeX to write his internship report, actively making use of *i*-LaTeX's transitionals for images, tables and grid layouts. In his comments on the transitionals he used, P6 states that it was "*very easy*" and "*extremely handy*" to crop images, "*very easy and intuitive to manipulate rows and columns*" in tables, and "*very easy to add/remove an element, and to change size*" in grid layouts. Regarding images and grid layouts, P6 also complained that not knowing how a change in the size of an image would affect the rest of the document still required him to compile a few times. Regarding tables, and just like P1, he also noted that "*for very easy changes (typically one row I have to move), just moving the corresponding code line is as quick as using the transitional*", further adding that this approach also prevented the document from recompiling, "*which can be long and useless*" (P6).

63. The activity in June shown in Figure 6.13 corresponds to a single text edit. Since P6 did not report authoring any LaTeX document during that month, it might just have been an involuntary action, such as opening a LaTeX file using Visual Studio Code and making a change in it by mistake.

Overall, even though he had to adapt his template a little bit to make it compatible with *i*-LaTeX, P6 concluded that "*the time spared with interactive editing + the satisfactory feeling when moving table rows or columns make i-LaTeX totally worth it*", even adding that "*it even becomes a little addictive*" the following week. However, he also wished he could have prevented *i*-LaTeX from recompiling the document every time he uses a transitional, and explained that when he did not have to edit any table, image or grid layout, he preferred to use the TeXMaker editor as he found it faster to compile than *i*-LaTeX.

6.6.3 DISCUSSION

Although the results must be interpreted with care, as only four participants actively participated and only two of them used *i*-LaTeX's transitionals, the outcomes of this longitudinal study are in line with what I previously observed and discussed regarding the controlled experiment. Comments and activity

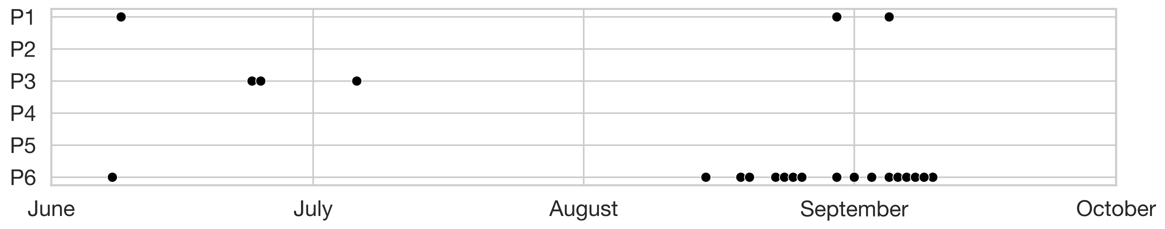


Figure 6.13. Timeline of the longitudinal study, which ran from early June to mid-September 2021. Each individual line corresponds to a participant, and each black dot corresponds to a day during which at least one event was logged by *i*-LaTeX. As explained in the text, neither P2, P4 nor P5 used *i*-LaTeX during the entire duration of the study, which explains why their individual lines are empty.

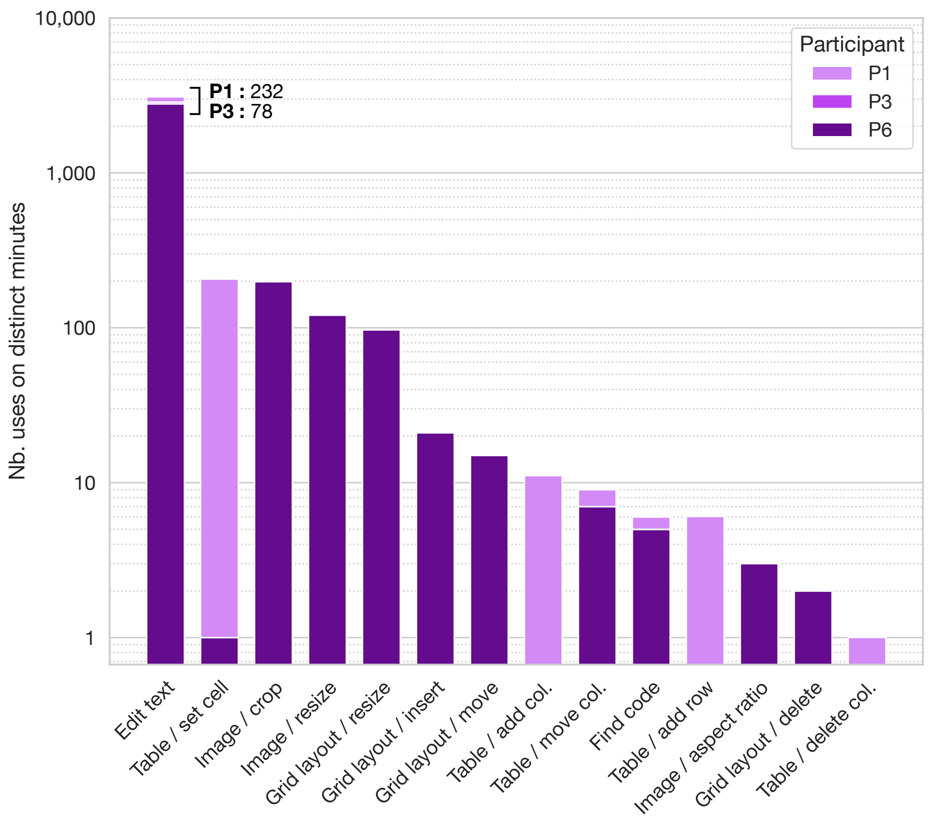


Figure 6.14. Distribution of the different types of events logged by *i*-LaTeX during the study. Instead of counting individual events, which can be hard to compare as is, the Y axis corresponds to the number of distinct minutes in which each participant used each feature present on the X axis at least once. For example, the *Find code* item corresponds to the number of events fired when a participant located the code of an element by clicking on the title of its transitional that were logged at least one minute apart from each other. Event types with a slash correspond to features that are part of a transitional, whose type come before the slash, and whose (kind of) feature come after the slash.

from P1, P2, P3 and P6 suggest that while *i*-LaTeX's transitionals were helpful for editing specific pieces of code which are otherwise harder to edit as text, most of the time required to author a LaTeX document is spent writing the body of the document by editing the code as text, often in their preferred editor. Several participants also highlighted that when an action that could be performed through a transitional was faster to perform by editing the text, such as when inserting or duplicating rows in a table, they would rather pick that second option. This further demonstrates that participants are indeed able to jump between two projections of the same piece of code, without complaining about having to switch between contexts. In addition, this study shows the medium-term impact of two limitations that were mentioned during the controlled experiment but could not be observed empirically.

The first limitation is that using transitionals require *i*-LaTeX users to switch to a different LaTeX editor. During the three months of the study, most participants preferred to keep using their usual LaTeX editor, and only considered using *i*-LaTeX when they believed the could benefit from *i*-LaTeX's non-textual projections for editing specific fragments of code. In particular, participants explicitly commented that *i*-LaTeX lacked (1) the collaborative aspect of Overleaf for writing multi-author documents, such as research articles (P1, P2), and (2) the efficiency of TeXMaker for compiling the document into a PDF. Moreover, although P6 did not regret spending some time adapting his template to make it compatible with *i*-LaTeX, the extra effort it required might be another reason why LaTeX users may prefer to stick with another editor that is readily compatible with their document's format.

The second limitation is that the lack of malleability of *i*-LaTeX is constraining its users. By not being able to extend *i*-LaTeX with new transitionals, P2 never considered using *i*-LaTeX to write his documents, as no transitional could help him author the pictures he was describing with TikZ. Similarly, by not being able to modify *i*-LaTeX, P6 had to repeatedly experience the same bugs, such as *i*-LaTeX's erroneous code generation during rapid image resizing,[64] without being able to investigate the source of the bug and fix it. Although I did not expect that any participant in this study would actually fix bugs or create new transitionals from scratch themselves, I believe such limitations—which are often pointed out in open-source software communities—would currently hinder *i*-LaTeX from being used at a larger scale.

64. This is the same bug as the one mentioned in subsection 6.5.2.

6.7 CONCLUSION

In this work, I demonstrated that LaTeX users can benefit from protean projection of LaTeX code, by combining a textual projection (the text editor) with other kinds of projections (the transitionals) in a single environment. The interview study highlighted a number of difficulties faced by LaTeX users, yielding four recommendations for design. I used these recommendations to ground the design of *i*-LaTeX, whose successful development demonstrated the technical feasibility of providing transitionals for code describing mathematical formulae, tables, images and grid layouts. The controlled evaluation showed that with transitionals, LaTeX users perform a number of tasks faster and with fewer recompilations, resulting in a lower workload and simpler strategies. The longitudinal evaluation increased the ecological validity of these findings, as several participants were able to use *i*-LaTeX's transitionals to edit their own document during extended periods of time, despite the limitations they sometimes faced. Overall, *i*-LaTeX was beneficial to beginner and expert users alike, many of whom reported that they would like to keep using them.

However, the two evaluation studies also suggest that LaTeX users would like to use transitionals with more advanced features, such as merging cells in tables, as well as other types of transitionals, such as for editing pictures created with TikZ. Moreover, participants in the first evaluation study reported that they would be very keen on using transitionals if they were available in their usual LaTeX editor; and participants in the second evaluation study indeed used *i*-LaTeX only if and when they needed to use transitionals, often switching back to another editor when they were done. These observations support the assumption that creating projections for established computer languages such as LaTeX is not enough to support their widespread adoption, even though they have demonstrated benefits and successfully became part of some LaTeX users' workflows. Furthermore, although I posit that transitionals initially developed for LaTeX could be adapted to benefit users of other document description languages, who may experience similar difficulties than those faced by LaTeX users, the language-specific nature of *i*-LaTeX unfortunately hinders reuse of its transitionals.

This conclusion therefore directed me towards a new research question: how to give users of computer languages the ability to create and adapt their own projections, within and across computer languages, without being restricted by the choices made by the authors of the code editing environment they use? I argue that addressing this problem is key to spreading the use of protean projection of document description languages. With a solution to this problem, LaTeX users could tweak transitionals to use representations that include features they lacked, add transitionals for fragments of code such as TikZ drawings and chemical formulae, and adapt transitionals to languages such as Markdown and HTML, without having to switch to a different environment. The next chapter proposes an experimental solution to this new problem.

7

Creating malleable projections

The work I conducted on projecting LaTeX code in a protean fashion, presented in the previous chapter, illustrated that complementing a textual projection with other substrates has a number of benefits, such as decreasing the time needed to complete certain tasks and lower the workload they induce. Yet, it also demonstrated that spreading the use of a tool such as *i*-LaTeX—and therefore spreading the use of protean projection—is a challenge in itself for at least two reasons. First, making users switch to a different editor than the one they are used to is difficult. Doing so prevents users from benefiting from features that made them choose a specific editor rather than another, such as its collaborative features, which may be prohibitively costly to reimplement in a new editor, especially if it is designed as a research prototype. Second, when designing a new text editor, one cannot reasonably plan ahead of the needs of all its future users nor the text fragments that will benefit from alternative representations in the languages that can be edited using this editor.

In this second line of my applied work, I introduce a new way to complement a text editor with additional projections of the code that fulfil these constraints. More specifically, I investigate a way to improve the malleability of these projections while achieving a diversity of projections and use cases at least as good as the state-of-the-art solutions presented in subsection 5.3.2, including those implemented in systems such as mage (Kery et al., 2020) and Livelits (Omar et al., 2021). Again, I used a user-centred design methodology to frame my research by grounding the work in a study of existing projections and by evaluating the framework I developed on a number of concrete use cases. The results were published in an article in the international ACM UIST'23 conference (Gobert and Beaudouin-Lafon, 2023).

This chapter presents my work on this project as an extended version of this publication. Section 7.1 defines the notion of malleable software, both in a general sense of the term and when applied to text editors specifically, and illustrates it with a number of classic examples. Section 7.2 presents a formative study in which I collected 62 projections from the literature and from a design workshop I organised with 9 participants, whose results shed light on the projections' diversity and reuse opportunities. Section 7.3 describes LORGNETTE, a framework for instrumenting existing text editors so as to equip them with additional projections of the code that can be created and modified by their users. Section 7.4 demonstrates how LORGNETTE can be used to implement projections in five use cases, ranging across multiple computer languages. Section 7.5 concludes with several directions to address the limitations of LORGNETTE, as a means of both supporting more diverse types of projections and easing their appropriation by end-users.

7.1 BACKGROUND

Although the notion of making software malleable was introduced about 50 years ago, its definition greatly varies from one piece of work to another, ranging from mere customisation to complete reprogramming. This section presents key steps in the evolution of the concept, focusing on recent stands and examples in the field of HCI, as well as specific forms of malleability in the context of letting users tailor their text editors to their own needs.

7.1.1 MALLEABLE SOFTWARE

The idea to let end-users appropriate software in "*personal and idiosyncratic ways*" (Klokmose et al., 2015, p. 1), i.e., to make software *malleable*, is far from new. Back in the early 1970s, Smalltalk was designed as an operating system in which everything could be accessed and modified by the user, including kernel aspects such as the process scheduler. Smalltalk's approach was a far cry from the layered approach that had been popularised by Multics' privilege rings only slightly earlier (Schroeder and Saltzer, 1972), in which the system distinguishes between modes, such as the *kernel mode* and *user mode*, therefore limiting what parts of the system users have access to. Yet, Smalltalk never took off outside of academic circles, and all the major operating systems in use today (and the microprocessors they are designed for) are designed with compartmentalisation in mind.

Although Smalltalk failed at making malleable operating systems a commercial success, it still paved the way for the more restricted and popular approach of *end-user customisation*, in which certain aspects of a piece of software can be reconfigured by its users without going all the way back to editing the source code. Yet, although Myers et al. (2000) state that "*end-user programming will be increasingly important in the future*" (§3.5) and attribute the success of spreadsheets and webpages to the fact that everyone can indeed self-program them—to some extent—without being a trained programmer, they also highlight that customising most applications still requires to know how to program using a specific programming language. As such, they conclude that "*the important research problem with scripting and customization is that the threshold is still too high*" (§3.5), therefore drawing attention to the need of finding better solutions to let non-programmers customise software.

More recently, malleable software has received renewed attention from the HCI community. In his doctoral work on this very topic, Tchernavskij argues that malleability goes beyond the now classic notion of adaptability and aims to let users break free of the application-centric vision that has been central in how we can and cannot interact with software in the past few decades.

Malleable software aims to increase the power of existing adaptation behaviors by allowing users to pull apart and re-combine their interfaces at the granularity of individual UI elements, such as toolbars, widgets, menus, documents, and devices. In other words, the goal is to erase the boundaries between apps and create an end-user accessible "physics of interfaces" that dictate how different interfaces and documents can be assembled. Malleable software should enable end users to reuse their favorite digital tools in different digital environments, combine the behaviors of multiple interfaces created by different developers, and recruit developers to modify or substitute interface elements.

— Tchernavskij (2019, ch. 4)

This vision echoes related arguments from other researchers, such as Dix's call for *designing for appropriation* (Dix, 2007), Beaudouin-Lafon's vision of *a world without apps* (@150), and Litt's blog post on the freedom of *bringing your own client* (@151). It was also implemented to various extents in a number of systems, including Webstrates (Klokmose et al., 2015), Codestrates (Rädle et al., 2017), Videostates (Klokmose et al., 2019), Stratify (Beaudouin-Lafon, 2023) and Mirrorverse (Grønbæk et al., 2023).

For example, Webstrates (Klokmose et al., 2015) are computational media that multiple users can concurrently view and modify by each opening the same Webstrate in a web browser. Changes to the Webstrate's data are synchronously and transparently replicated across every user; but users can also decide to modify the webpage in a purely local fashion, such as by editing the DOM to customise the user interface of the document or application implemented as a Webstrate without imposing their personal preferences on other users. Stratify (Beaudouin-Lafon, 2023) goes one step beyond by explicitly describing the dependency graph of resources and transformations between them and making every substrate they are projected onto a potential target for a number of *interaction instruments* (Beaudouin-Lafon, 2000), which determine the semantics of input events, such as whether a click should activate or delete the pointed element, without determining what the target of the action is. As such, instruments are *polymorphic*:[65] the same instrument can equally be used to delete an element in a list of items or a user interface component, without having to consider what else they should update in turn. Any change is automatically propagated through the dependency graph in a reactive fashion, therefore automatically updating other substrates that form the user interface. Stratify therefore helps users tailor a specific part of the user interface to their needs in an entirely interactive fashion by giving them instruments to modify the user interface in a coherent and programming-free fashion.

65. In this context, *polymoprhism*, as introduced by Beaudouin-Lafon and Mackay (2000), means that a single instrument (action verb) can be applied to different targets (subjects).

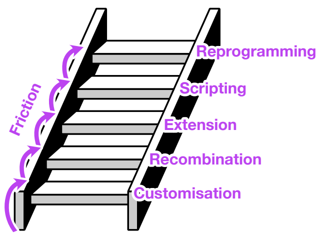


Figure 7.1. Schema of the five levels of the ladder of tailorability and the friction caused by climbing it. It is inspired by the schema published by Grønbæk et al. (2023, §4).

To clarify what constitutes the malleability continuum, which may include changing a single setting as well as appropriating the source code of a system, Grønbæk et al. (2023) build on earlier work from Mørch (1997) to introduce a *ladder of tailorability* that includes five levels—customisation, recombination, extension, scripting and reprogramming—defined as follows:

Customisation adapts software functionality or aesthetics using only predefined settings;

Recombination assembles ready-made blocks or primitives to produce new functionality;

Extension adds new functionality from external sources;

Scripting adds new behavior using an exposed API or a built-in scripting language;

Reprogramming changes an application's functionality by editing its source code or by interfacing with it directly.

In addition, Grønbæk et al. also introduce the notion of *friction* to describe the cost of moving up the ladder of tailorability from the user's point of view. The five levels of the ladder and the notion of friction are schematised in Figure 7.1.

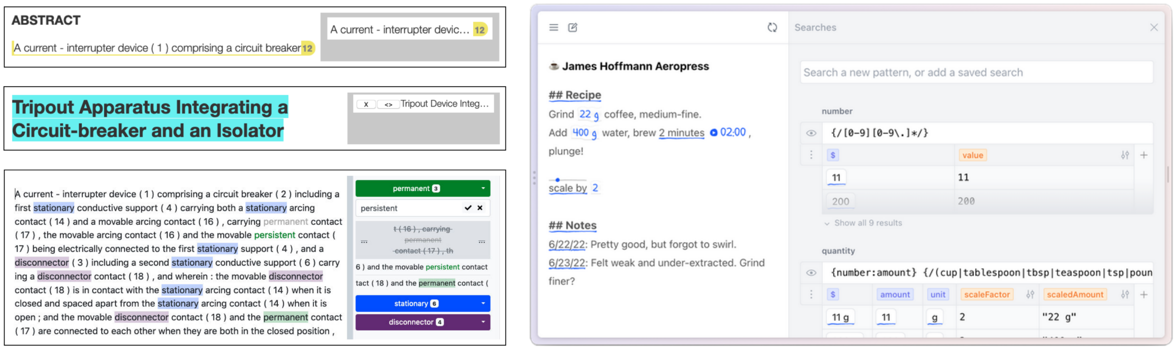
Friction sheds light on the fact that, in addition to making software more malleable, lies the challenge of making malleability itself more inclusive. As such, releasing source code under an open-source licence is not enough to give the power to modify a piece of software to a broad audience, as editing source code requires skills that most users do not have. Giving users access to multiple steps of the ladder of tailorability helps address this challenge, as each step offers a different trade-off between the amount of tailorability that can be achieved and the amount of skills and effort it requires. Moreover, even skilled programmers can benefit from this multiplicity, as they too can experience friction. For example, while scripting and reprogramming cause friction by excluding users who do not possess programming skills, they may also cause friction by forcing skilled programmers to use a language they are not familiar with or to write code using a specific programming environment they dislike, which may refrain those with the appropriate skills from using them altogether.

7.1.2 TAILORING TEXT EDITORS

Among all pieces of software, text editors have a long history of being malleable. Early interactive text editors from the late 1970s and early 1980s, such as vim and Emacs, already supported several types of malleability, from customising specific settings such as key bindings to scripting vim (using the Vim script language), and even reprogramming most of the Emacs editor using the same Lisp dialect it is written in. Eventually, package managers were developed to ease sharing and installing extensions, therefore providing an intermediate step between customisation and reprogramming.[66] Similarly, most text editors specialised for editing computer languages that have been developed in the past two decades support customisation and extension. In addition, since several of them have been open-sourced, they also support reprogramming too—some, such as Atom (@153), are even being described as "*a hackable text editor for the 21st century*".

66. For instance, Emacs includes a primitive to install extensions from ELPA (@152), the official Emacs package repository.

While most of the tailoring proposed by the editors presented above are unrelated to the content of the text that is being edited, several directions have been proposed to tailor text editors in a content-aware fashion. For example, Textlets (Han et al., 2020) let users reify arbitrary text selections as different sorts of interactive objects, each with their own behaviour, such as counting the number of words or keeping trace of alternative versions


ABSTRACT

A current - interrupter device (1) comprising a circuit breaker 12

A current - interrupter devic… 12

Tripout Apparatus Integrating a Circuit-breaker and an Isolator

Tripout Device Integ…

A current - interrupter device (1) comprising a circuit breaker (2) including a first stationary conductive support (4) carrying both a stationary arcing contact (14) and a movable arcing contact (16) , carrying permanent contact (17) , the movable arcing contact (16) and the movable persistent contact (17) being electrically connected to the first stationary support (4) , and a disconnector (3) including a second stationary conductive support (6) carry ing a disconnector contact (18) , and wherein : the movable disconnector contact (18) is in contact with the stationary arcing contact (14) when it is closed and spaced apart from the stationary arcing contact (14) when it is open ; and the movable disconnector contact (18) and the permanent contact (17) are connected to each other when they are both in the closed position ,

permanent 3

persistent

~~t (16) , carrying permanent contact (17) , th~~

6) and the movable persistent contact

tact (18) and the permanent contact (

stationary 6

disconnector 4

James Hoffmann Aeropress

Recipe

Grind 22 g coffee, medium-fine.

Add 400 g water, brew 2 minutes 02:00 , plunge!

scale by 2

Notes

6/22/22: Pretty good, but forgot to swirl.

6/23/22: Felt weak and under-extracted. Grind finer?

Searches

Search a new pattern, or add a saved search

number

```
{/[0-9][0-9\.]*/}
```

$	value
11	11
200	200

Show all 9 results

quantity

```
{number:amount} {/(cup|tablespoon|tbsp|teaspoon|tsp|poun
```

$	amount	unit	scaleFactor	scaledAmount
11 g	11	g	2	"22 g"

a. Textlets. **b.** Potluck.

Figure 7.2. Examples of text editors that let users turn specific ranges of text into persistent and interactive elements of the user interface. (a) In Textlets, users can assign different behaviours to arbitrary text selections. For example, a *countlet* (top) counts and displays the number of word in the selected range; a *variantlet* (middle) allows users to switch between two versions; and a *searchlet* (bottom) reifies a search as a persistent collection of matching ranges. (b) In Potluck, users can search for patterns and write formula to transform each match (right panel) and see and interact with modified occurrences directly in the text (left panel).

(Figure 7.2a). Similarly, Potluck (@141) supports searching for patterns of text and replacing matching occurrences with the result of a user-specific formula, akin to a spreadsheet formula, optionally augmented with a widget to, e.g., control a numeric value using a slider or add an inline timer next to a duration (Figure 7.2b). More recently, a new version of Potluck demonstrated how integrating artificial intelligence could help writing and explaining formula, whose domain-specific syntax may discourage non-expert users, therefore decreasing the friction of adding text-aware interaction to one's document.[67]

67. This version was presented by Geoffrey Litt during a talk titled *Dynamic documents as personal software* (@154), which was given at the Causal Islands 2023 conference.

Interestingly, there is little work on applying this approach to text editors designed for editing computer languages. On the one hand, systems such as CodeQL (@155) and Tree-sitter (@156) support finding relevant pieces of code in a code base by writing patterns, with the goal of helping developers transform syntax trees and detect vulnerabilities. Sporq (Naik et al., 2021) infers patterns of code from user-selected text, which can be further refined by marking candidates as positive or negative matches; and tools such as structural search-and-replace (Mossienko, 2004) and reCode (Ni et al., 2021) help users generalise a text edit by either describing or inferring a syntactic transformation pattern that can then be applied to all matching pieces of code. Yet, neither of these systems is designed to assign a persistent user interface element to matching pieces of code in order to help users understand and modify them. On the other hand, systems such as Codelets (Oney and Brandt, 2012), Visual syntax (Andersen et al., 2020), mage (Kery et al., 2020) and Livelits (Omar et al., 2021) do offer such user interfaces, but none of them let users specify the pattern of the fragments of code that can benefit from alternative projections. Codelets only works with code snippets when they are inserted in the text editor; mage's projections can only be invoked by writing and evaluating specific magic commands in notebook cells; and Visual syntax and Livelits only provide projections for specific macros, which must be defined in the same computer language as the text is written in.

The work of Beckmann et al. (2023b) on visual replacements, published one week before the publication of my own work on creating malleable code projections in text editors at the ACM UIST'23 conference (Gobert and Beaudouin-Lafon, 2023), also proposes to combine non-textual projections of code fragments with a search pattern-based approach (based on Tree-sitter). However, although the directions we took share a number of similarities, their work focuses on helping users tailor a new, standalone syntax-directed editor, which internally represents code as a tree structure, whereas my work focuses on augmenting existing, standard text editors with additional projections.

7.2 FORMATIVE STUDY

To better understand which existing projections are used for and what type of features should be prioritised, I analysed a set of projections that I collected by organising a design workshop with nine programmers and by reviewing projections published in the literature. In this section, I present the methodology I used for collecting projections and report on the results of the analysis.

7.2.1 DESIGN WORKSHOP

To collect projections from programmers who commonly use computer languages, I organised a participatory design workshop with the goal to ask participants to imagine and design projections that they would like to use in their own work.

Participants

I recruited 9 participants (6 men, 2 women, 1 prefer not to say; age 18–44) by posting messages on the mailing list of a computer science department. 2 participants were research engineers, 3 were Ph.D. students, and 4 were associate professors or researchers in computer science, human-computer interaction and information visualisation. All participants had at least 5 years of experience with programming, and 6 (66%) had more than 10 years of experience. All of them program at least once a month, and 7 participants (78%) do it at least once a week. In addition, all were familiar with multiple programming languages. Participants did not receive any compensation for their participation.

Setup

The workshop was organised in person and lasted about 2 hours. Based on data collected in a questionnaire sent to participants beforehand,[68] I put participants in groups of 3 so as to maximise the diversity of the profiles in each group. All the tasks were on paper. Participants were provided with all the drawing material they needed.

68. See the procedure below for details.

Procedure

Before the workshop, I asked participants to fill in an online questionnaire to collect demographics and information about their experience with programming. To prompt participants, I also asked them to write about a situation in which they wished they had access to an alternative representation of a piece of code. At the start of the workshop, each participant was asked to summarise the situation they wrote about.

I then introduced the concept of projection and presented the three tasks to perform. I showed participants videos of projections that I created that demonstrated code manipulation using a colour picker, an interactive grid and a style inspector. I also briefly explained how projections work, insisting on the fact that the same user interface can be reused in different contexts.

In the first task, I asked participants to list as many ideas of projections as they could think of—first by proposing three "good" ideas and two "bad" ideas individually, and then by discussing ideas within their group. Each idea had to include a title, a short description of the situation it applies to, a short description of the user interface of the projection, as well as one or several goals among: discovering features, understanding code, modifying code, debugging, explaining, or any freely specified goal.

In the second task, I asked each participant to choose two ideas generated by their group and to design the projections they describe. Each design had to include a description of the context in which the projection could be used, an example of the fragment of code or data that that would be projected, and an annotated sketch of the user interface of the projection.

In the third task, I gave each group the designs created by another group and asked participants to choose one design and to apply it to a new situation. Each redesign had to include a description of the new context the projection could be used in and an example of the new fragment of code or data that would be projected. They could optionally include a new version of the sketch if the user interface had to be adapted to work in the new situation.

Data collection

At the end of the workshop, I collected and transcribed all the documents produced by the participants.

Data analysis

I manually annotated the transcription of each design (prefixed with D) and redesign (prefixed with R) to comment on the nature of the source and the representation of each projection, the context it appears in, and challenges it poses in terms of implementation. I ignored 3 designs and redesigns that I categorised as generic code editor features rather than projections: making type-aware suggestions (D5), performing global renaming (R6) and displaying the definition of a symbol (R8). I did not ignore D15 (highlighting polyglot code) because providing syntax highlighting for specific fragments of code may require user knowledge to, e.g., create projections for strings written in a different language than the host language, such as SQL queries embedded in a Java or Python program.

7.2.2 EXAMPLES FROM THE LITERATURE

In order to increase the diversity of the projections collected during the workshop, I also collected projections from systems published in the literature. I specifically considered non-textual projections, therefore excluding syntactic interaction techniques and visual augmentations of textual projections (Sulír et al., 2018).

Since the term *projection* is not commonly used to describe the kind of interactive system I am interested in, I did not use a systematic approach such as a keyword search. Instead, I reviewed systems presented in articles published in the past few years to major conferences/journals in the fields of both human-computer interaction and programming systems and others systems recursively referenced therein.[69] I also included the four projections that I implemented in *i*-LaTeX, which had been published at the time I carried out this study.

69. I notably reviewed articles from ACM conferences such as CHI, UIST, SPLASH and PLDI, as well as the IEEE VL/HCC conference and the Programming journal.

During this collection phase, I only retained projections that were part of working systems. I therefore ignored unimplemented ideas of potentially useful projections, such as table-shaped diagrams for TCP messages (Andersen et al., 2020), although these ideas could lead to useful projections if they were implemented. This method should not be considered as an extensive nor systematic review of the literature on projections, and I do not claim that the results fully capture the variety of projections available in the literature.

7.2.3 RESULTS

The participants produced 44 projection ideas in the first part of the workshop, of which 23 were developed into actual projection designs, which are listed in Table 7.1 and illustrated in Figure 7.3. In addition, I collected 39 projections from the literature, resulting in a total of 62 projections.

I then analysed the collected projections in terms of contexts of use (which situation are they used in?), source (what is projected?) and representation (what substrate is it projected on?). The results are presented below and summarised in Table 7.2, which classifies the 62 projections by representation and type of source data.

Contexts of use

Projections can target different audiences such as software developers, domain experts, e.g., to create electronic circuit (Voelter et al., 2019), teachers, e.g., to create assignments (D14, D16, R2, R3), and students, e.g., to learn Rust (Almeida et al., 2022). They can also serve very diverse goals: among the 44 projection ideas that were proposed by workshop participants, 26 (59%) were designed to help them understand code, 24 (55%) to modify it, 16 (32%) to debug it, and 12 (27%) to explain it or to discover new features. One participant was also interested in using a projection to generate synthetic data (D12).

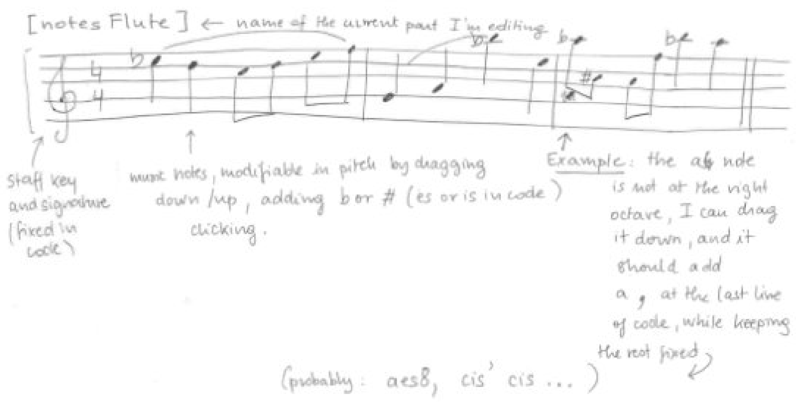
Projections can be used in diverse types of programming systems. In the literature, they are used in regular text editors (27/39), notebooks (5/39) and specialised environments (7/39). In the workshop, this was often un-

N°	Description of the design
D1	An interactive music staff to edit LilyPond music scores.
D2	A textual explanation of a binary optimisation program written in Python with Google's OR-tools.
D3	A form to write and edit a configuration file for, e.g., connecting to a database or a SMTP server.
D4	A form to configure the options of the `ffmpeg` command line utility.
D5 *	A menu to paste previously copied code that matches the expected type at the cursor position.
D6	A text editor to modify a Javadoc comment in the code from within the generated documentation.
D7	A graph whose nodes represent editable function bodies and arcs represent calls between them.
D8	A 3D trajectory editor for a CSV file whose rows contain timestamps and 3D coordinates.
D9	Previews of files and patterns that will be expanded when typing a path in a terminal.
D10	A form to configure the arguments of a Swift function for using Pandoc.
D11	A grid representing a webpage to create HTML elements spanning over the selected cells.
D12	A map to draw trajectories and interact with a distribution defined along them to generate data.
D13	An interactive image that can be moved and resized in a PDF generated from LaTeX code.
D14	A list of code regions that can be hidden to generate Python code with holes for students.
D15	An interface to syntactically highlight SQL requests written as strings in Java.
D16	An interface to replace code regions by "TODO" comments to generate code with holes for students.
D17	A sequence diagram representing property accesses between objects over time.
R1	A 3D space showing the evolution of the reference point for each operation in Processing.
R2	An interface to show the expected solution of a coding assignment next to the student's code.
R3	An interface to replace code regions differently for assignments with multiple levels of difficulty.
R4	A typeset mathematical formula that can be edited to modify mathematics written in LaTeX.
R5	A form to configure the fields of a front matter of a Markdown document written in YAML.
R6 *	A text input to rename an "id" property of an HTML element that also updates CSS rules.
R7	A preview of the triangles/rectangles formed by successive 3D coordinates in an OBJ file.
R8 *	Previews of the definition of certain expressions at the location where they are used in the code.
R9	A grid that allows to configure a grid layout to be applied to the children of an HTML element.

Table 7.1. Description of the designs (prefixed with D) and redesigns (prefixed with R) created by the participants of the design workshop. Asterisks (*) indicates that I ignored the design in the analysis, for reasons explained in the text.

LilyPond code

```
Notes Flute =
  \relative { \time 4/4
    ees’4(d c8 ees f)
    G,4(a b’)e(
    Aes8 cis, cis f bes4 aes4 \bar “||”
}
```


[notes Flute] ← name of the current part I'm editing
staff key and signature (fixed in code)
music notes, modifiable in pitch by dragging down/up, adding ♭ or # (es or is in code) clicking.
Example: the a♭ note is not at the right octave, I can drag it down, and it should add a , at the last line of code, while keeping the rest fixed
(probably: aes8, cis' cis ...)

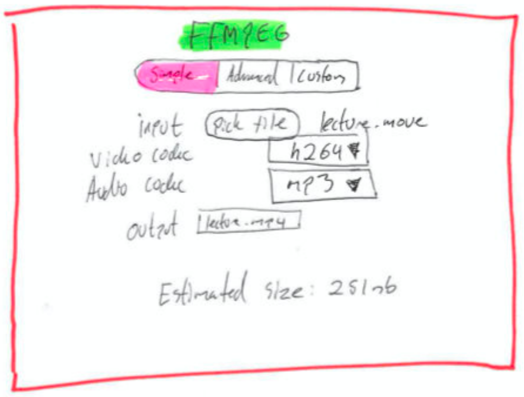
a. Design D1.

Command line instruction

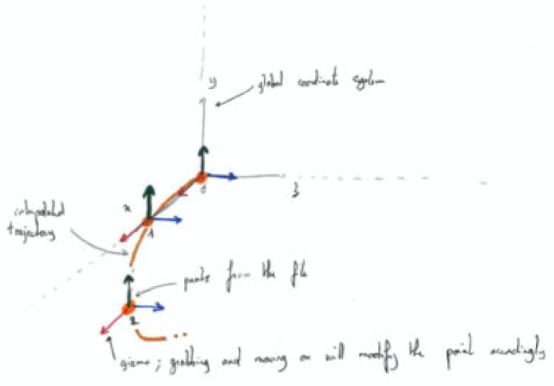
```
ffmpeg -I lecture.mov \
  -vcodec h264 -acodec mp3 \
  lecture.mp4
```

CSV file

```
id ; time ; x   ; y ; z
0  ; 0    ; 0   ; 0 ; 0
1  ; 0.1  ; 0.1 ; 0 ; 0
2  ; 0.2  ; 0.3 ; 0 ; 0,1
```



b. Design D4.



c. Design D8.

Figure 7.3. Examples of three projections designed by workshop participants, including the sample code (top) and the sketch (bottom). (a) Code written in LylyPond is projected onto an interactive music staff, which can be used to modify by dragging notes and inserting accidentals. (b) A command line instruction to run the `ffmpeg` program is projected onto a form, which helps configure some of the many options that can be specified using arguments. (c) A timed sequence of 3D positions encoded in the CSV format is projected onto a 3D plot showing the position of each point, which can be dragged to modify the numeric coordinates in the code.

specified, but most projections seem to be designed for text editors and two for command-line interfaces (D4, D9). They can also work with multiple languages—not just programming languages—including Python, Java, C, HTML, CSS, JavaScript, Swift, Rust, Racket, Haskell, Hazel, $\LaTeX$, Markdown, YAML and others.

Source

Projections can rely on both *static* data available at all times, e.g., the code of a $\LaTeX$ table, and *dynamic* data available only during execution, e.g., the content of a Python dataframe. Some projections use both: for instance, mage's image editor (Kery et al., 2020, fig. 3) gets the image to edit from the memory at runtime but reads the area to crop in the code. Furthermore, one workshop participant proposed a projection idea (that was not turned into a design) to visualise the introduction of a merge conflict with another branch of their versioning system, which suggests that some projections may need to access other data that exist in the computational context, even though it is not directly related to the code or its execution, such as the history of a version control system.

When a projection represents code, the scale of the code also differs from one projection to another. Most projections only represent a local fragment of code (50/62), e.g., a colour value or a local tree transformation. Others either represent entire files (6/62), e.g., a list of timestamped 3D positions or an assignment for students, or code located in multiples files (6/62), e.g., the structure of a container in a HTML file and the related style in a CSS file. This echoes the multiple scales of concepts implemented as code that have been identified in the literature: beacons (Wiedenbeck, 1986) and nano-patterns (Gil et al., 2019) at the instruction/line level; micro-patterns (Gil and Maman, 2005) at the structure/file level; and design patterns (Gamma et al., 1995) at the project/multi-file level.

Representations

More than half the projections (37/62) take the form of either a grid, a graph or a form, which seem to be very versatile substrates. The rest of the projections use other sorts of user interfaces, including some highly specific ones that were used in only one situation, such as a map (D12) and a music staff (D1, shown in Figure 7.3a). While the same user interface can be reused to represent different concepts, some concepts can also be represented by different substrates, e.g., a state machine can be represented both by a grid and a graph. Regarding interactivity, about three substrates out of four allow to modify the code interactively (45/62), while the rest only serve as static representations of the projected data.

Projections can also be displayed at different locations, regardless of the substrate used by their representation. In the literature, this includes in-line projections (11/39), which mix with textual code; float projections (14/39), which appear next to the code; standalone projections (10/39), which appear in a different panel; and embedded projections (4/39), which appear in a different document related to the code. Besides a few projections created with

Representation	Projected concept	Origin
Form	Colours	Graphite (Omar et al., 2012)
		Livelits (Omar et al., 2021)
	GUI component properties	Codelets (Oney and Brandt, 2012)
	Graphical properties	Codelets (Oney and Brandt, 2012)
	Animation properties	Codelets (Oney and Brandt, 2012)
	Form building	Visual syntax (Andersen et al., 2020)
	Regular expression	Graphite (Omar et al., 2012)
	Configuration file	D3, R5
	FFmpeg configuration	D4
	Pandoc configuration	D10
	Documentation comment	D6
	File path	D9
	Code assignment	D14, D16, R2, R3
Graph	List	Heterogeneous languages (Erwig and Meyer, 1995)
		Vital (Hanna, 2002)
	Tree	Heterogeneous languages (Erwig and Meyer, 1995)
		Alectryon (Pit-Claudel, 2020)
		Visual syntax (Andersen et al., 2020)
	State machine	Heterogeneous languages (Erwig and Meyer, 1995)
		MPS (Voelter et al., 2019)
	Rust's ownership	RustViz (Almeida et al., 2022)
	Runtime stack and heap	Python Tutor (Guo, 2013)
	Reactive stream	Poker (Descheemaeker et al., 2021)
	Call graph	Reacher (LaToza and Myers, 2011)
		D7
Grid	Dataframe	The Gamma (Petricek, 2020)
		mage (Kery et al., 2020)
		Livelits (Omar et al., 2021)
	State machine	MPS (Voelter et al., 2019)
	Document table	*i*-LaTeX (Gobert and Beaudouin-Lafon, 2022)
	Grid layout	*i*-LaTeX (Gobert and Beaudouin-Lafon, 2022)
		D11, R9
2D plot	Dataframe	mage (Kery et al., 2020)
		B2 (Wu et al., 2020)
	Machine learning metrics	Skyline (Yu et al., 2020)
	Loop parallelisation	Clint (Zinenko et al., 2015)
3D plot	3D trajectory	D8
	3D object	R7
	Transform reference point	R1

Representation	Projected concept	Origin
Typeset mathematics	Coded formula	Barista (Ko and Myers, 2006)
		Alectryon (Pit-Claudel, 2020)
		i-LaTeX (Gobert and Beaudouin-Lafon, 2022)
		R4
	Optimisation problem	D2
2D game board	Conway's *Game of Life*	Alectryon (Pit-Claudel, 2020)
	Tsuro	Visual syntax (Andersen et al., 2020)
Circuit diagram	Hardware circuit	Visual HDL blocks (Lin et al., 2021)
	Quantum circuit	Notate (Arawjo et al., 2022)
Image editor	Image transformation	mage (Kery et al., 2020)
		Livelits (Omar et al., 2021)
		i-LaTeX (Gobert and Beaudouin-Lafon, 2022)
		D13
Marble diagram	Reactive stream	RxFiddle (Banken et al., 2018)
Music staff	LilyPond score	D1
Map	Geodata synthesis	D12
Sequence diagram	Variable access	D17
Highlighted code	Polyglot code	D15

Table 7.2. List of projections I reviewed. The origin can either be a workshop design or redesign number, as specified in Table 7.1, or a reference to a system published in the literature.

MPS, none of the systems I reviewed let users change the display location of a projection—even though that may benefit some user groups, as discussed by Omar et al. (2021, §5.3).

7.2.4 DISCUSSION

The results of the study depict a very eclectic collection of projections. They can be used in different sorts of programming environments and languages by both experts and non-experts. They project very different types of data, which may originate from the code, the runtime state during its execution, or other resources that exist in the same computational context; and they can be represented by both very generic or very specific substrates. A single substrate can be used in different situations, and the same data can be projected on different substrates. Overall, this diversity contrasts with current approaches for engineering code editors, which are hardly malleable and flexible enough to let end-users tailor their editors by exploring the design space of projections that emerges from this study.

For example, model-driven engineering systems and language workbenches such as Barista (Ko and Myers, 2006) and MPS (Voelter and Lisson, 2014) can only create editors for a single language, in which only predefined nodes of the syntax tree can be projected onto more or less arbitrary substrates. Since the code is internally represented as a tree, it makes them ill-adapted to project patterns of text; and apart from a few examples, such as the Cornell Program Synthesizer (Teitelbaum and Reps, 1981), they usually cannot exploit runtime data.[70] In terms of malleability, this approach barely supports reprogramming, making the cost of creating or modifying projections very high for end-users, who must recompile modified sources into a new editor before they can use them. Image-based programming systems are more appropriate for exploiting runtime data since they blur the distinction between code and data, but just like the aforementioned systems, they require to use their own language and do everything in their isolated ecosystem.

70. When using this approach to create an editor for a programming language meant to be executed from a textual encoding, mapping runtime data to code in the editor is particularly challenging as the internal representation of the editor must first be transformed into text and possibly be compiled before it can be executed, requiring to map the runtime state back to text *and* to map the text back to the internal representation used by the editor.

Visual syntax systems, such as visual syntax for Racket (Andersen et al., 2020) and Livelits for Hazel (Omar et al., 2021), lower the cost of creating custom projections by enabling end-users to write special macros that specify a user interface for configuring the macro (and therefore the actual code it generates on expansion). While this approach lowers the tailoring level from reprogramming to scripting, rather than mere reprogramming, it is deeply language-specific, does not favour reuse, and requires to use the special macros in order to benefit from their non-textual projections, meaning that all existing code will not benefit from them.

Overall, existing engineering approaches are specific to a single language and target predefined AST nodes rather than all the pieces of code that match user-specified patterns. Although the editors themselves may be reconfigured or extended with plugins, to the best of my knowledge, this is hardly ever used for creating alternative projections of the code. At best, providing such projections is possible by exploiting a generic API to write a plugin, at the cost of a friction almost as high as that of plain reprogramming. For example, although Visual Studio Code includes a colour picker interface to interact with colour codes in some languages, there is no way for end-users to adapt the projection to make it work with a different syntax or language unless they create a plugin providing an entirely new text editor interface to Visual Studio Code.

To address this kind of limitations, I focused on equipping text editors with projections that can be adapted using reconfiguration, recombination, extension and scripting. In particular, I was interested in making the same user interface easy to reuse across different languages and situations; a feature not available in other text editing systems with protean projections, even though the results of my formative study show that forms, graphs and grids are used in 62% of the projections that I collected.

7.3 THE LORGNETTE FRAMEWORK

In light of the limitations of current approaches to engineer projections in text-based code editing environment, I developed LORGNETTE, a new framework for augmenting text editors with alternative projections of the content that can be created and modified by end users. LORGNETTE's malleability targets multiple categories of users, from novices who only want to tweak an existing projection to adapt it to a new use case to seasoned programmers willing to create projections with custom user interfaces to accompany a library they work on. In this section, I present the architecture and concepts of LORGNETTE, describe its implementation and compare its features to those of related systems.

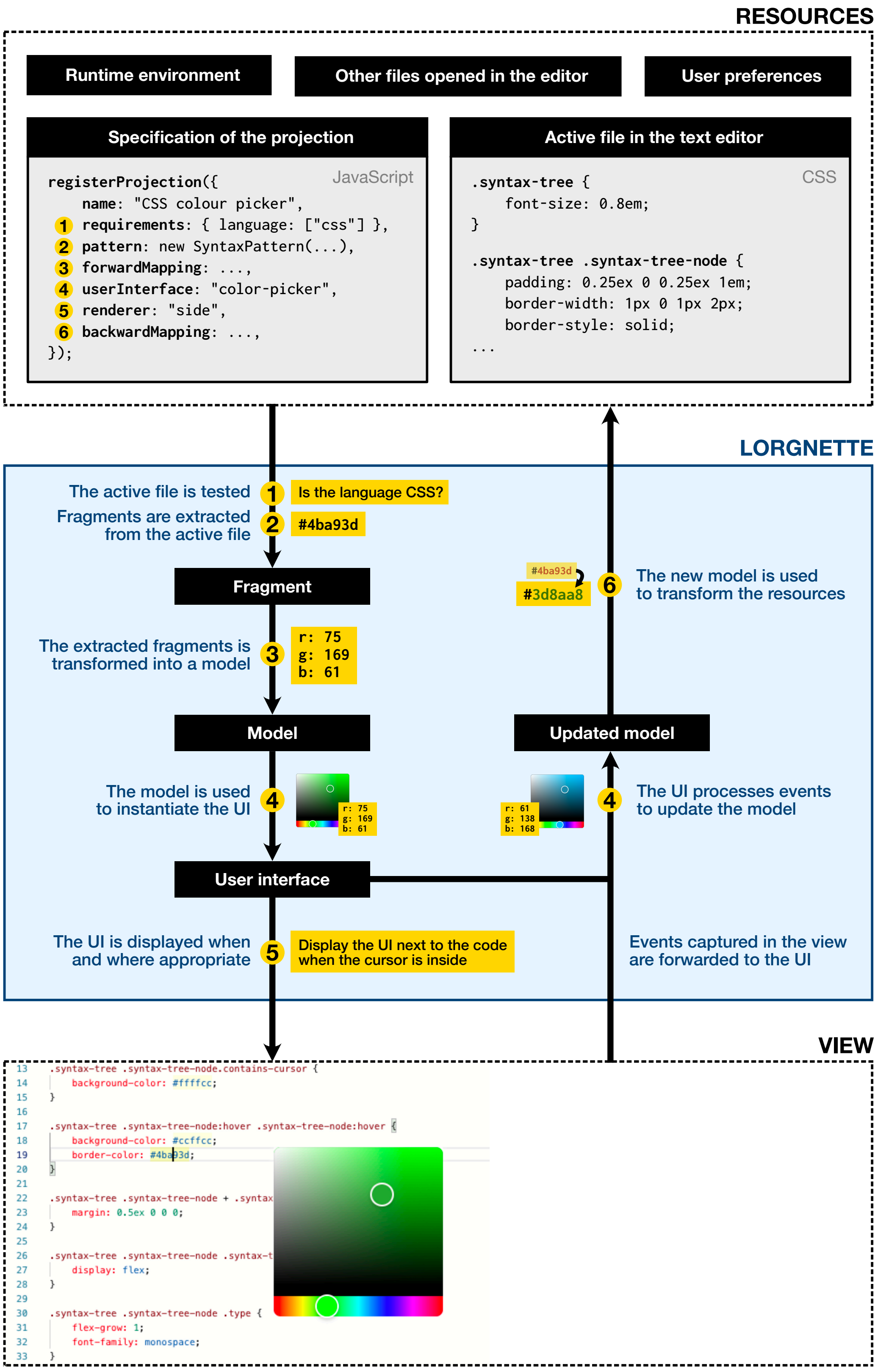
7.3.1 CONCEPTS

LORGNETTE is a framework for augmenting code editors with projections. It is not a code editor in itself, but a means to instrument a text editor so that its users can freely create, modify and delete projections of the code already projected as text without needing to modify the source code of the editor or waiting for the developers of the text editors to do so. To be compatible with LORGNETTE, the system that includes the text editor must give LORGNETTE access to *resources*, such as the content of the active file and a means of communication with the runtime environment, as well as a *view* in which LORGNETTE can display its own projections and capture events.

Once a code editor has been augmented with LORGNETTE, users can start augmenting it with projections by writing projection specifications. A *specification* is a blueprint for a projection that tells LORGNETTE when and how to create it. It has three main responsibilities: extracting the appropriate resources, mapping them to a model, and pairing it with a user interface. The whole process of turning a specification into a projection is schematised in Figure 7.4. The numbers in yellow discs in the text below refer to the six steps shown in the figure.

Extracting resources

The first responsibility of a specification is to list its *requirements* (1), i.e., the conditions that must be fulfilled for the projection to be created. They typically represent assertions about the environment, such as the language(s)


RESOURCES
Runtime environment
Other files opened in the editor
User preferences
Specification of the projection
JavaScript
Active file in the text editor
CSS
LORGNETTE
The active file is tested
1
Is the language CSS?
Fragments are extracted from the active file
2
#4ba93d
Fragment
The extracted fragments is transformed into a model
3
r: 75 g: 169 b: 61
Model
The model is used to instantiate the UI
4
r: 75 g: 169 b: 61
User interface
The UI is displayed when and where appropriate
5
Display the UI next to the code when the cursor is inside
#4ba93d
#3d8aa8
6
The new model is used to transform the resources
Updated model
r: 61 g: 138 b: 168
4
The UI processes events to update the model
Events captured in the view are forwarded to the UI
VIEW

```
registerProjection({
    name: "CSS colour picker",
1   requirements: { language: ["css"] },
2   pattern: new SyntaxPattern(...),
3   forwardMapping: ...,
4   userInterface: "color-picker",
5   renderer: "side",
6   backwardMapping: ...,
});
```

```
.syntax-tree {
    font-size: 0.8em;
}

.syntax-tree .syntax-tree-node {
    padding: 0.25ex 0 0.25ex 1em;
    border-width: 1px 0 1px 2px;
    border-style: solid;
...
```

```
13  .syntax-tree .syntax-tree-node.contains-cursor {
14      background-color: #ffffcc;
15  }
16
17  .syntax-tree .syntax-tree-node:hover .syntax-tree-node:hover {
18      background-color: #ccffcc;
19      border-color: #4ba93d;
20  }
21
22  .syntax-tree .syntax-tree-node + .syntax
23      margin: 0.5ex 0 0 0;
24  }
25
26  .syntax-tree .syntax-tree-node .syntax-t
27      display: flex;
28  }
29
30  .syntax-tree .syntax-tree-node .type {
31      flex-grow: 1;
32      font-family: monospace;
33  }
```

Figure 7.4. Description of the process followed by LORGNETTE to process the specification of a projection stating that the text editor must display a colour picker to preview and modify hexadecimal colour codes in CSS whenever the cursor is inside one. The resources and the view (dashed areas) correspond to the interface between LORGNETTE and the text editor, which must provide certain resources and a view to be compatible with LORGNETTE via, e.g., an extension API. In this example, only the specification and the active file are used by LORGNETTE, and the view displays the text editor augmented with a colour picker representing the colour code at the cursor's position (line 19).

the projection is designed for. Moreover, they must also specify what are the *resources* (2) needed by the projection, i.e., to create a model for the user interface. Resources represent different sorts of data that can be used by projections. The main type of resource is the *fragment* of code that is being projected, either in the form of a range in the text (a textual fragment) or in the form of a node in the syntax tree if the language can be parsed (a syntactic fragment). Specifications must include a *pattern* that will be used by LORGNETTE to search for matching fragments in the code, each of which will be projected. These patterns can either be textual, e.g., using a regular expression, or syntactic, e.g., using an assertion that must be true for syntax tree nodes that can benefit from the projection. If the environment supports it, resources can also include runtime information, such as the value of a variable at a certain point in time. This requires to write *runtime requests* in the specification, which specify when and how LORGNETTE should query the runtime (such as a debugger running the code that is being projected) and whose responses will be provided to the projection as resources. LORGNETTE currently supports these three types of resources (textual fragments, syntactic fragments and runtime information), but other types of resources could be made available to projections as well, such as local files or environment variables.

Mapping resources to a model

When all the requirements to create a projection are met, the requested resources are extracted and passed to the *forward mapping* (3). The forward mapping is an arbitrary function that is responsible for processing the resources in order to return a valid model M, such as by extracting useful information from the code fragment. Whenever the state is modified, typically through the user interface, the new model M' is passed to the *backward mapping* (6), possibly along with extra data (such as the previous model and other information on what changed). The backward mapping is an arbitrary function that is responsible for updating the underlying resources, e.g., by modifying the code fragment, so that applying the forward mapping would produce M', i.e., so that the set of forward and backward mappings form a *well-behaved lens* (Foster et al., 2007). In case the state is never modified, such as when using a projection to display a static representation, the backward mapping can be left undefined.

Rendering the user interface

Once the model has been generated by the forward mapping, the *user interface* (4) can be instantiated using M and provided to the *renderer* (5). The user interface has the responsibility of deciding *how* to represent the data, process events such as clicks and key presses, and update the model when appropriate; while the renderer has the responsibility of deciding *when* and *where* the user interface should be displayed in the code editor. For instance, by switching between two different renderers, the same user interface could be displayed either in a separate panel or next to the cursor when it is inside the code being visualised.

Reusing patterns and mappings as templates

While requiring users to write arbitrary mappings between resources and a model allows LORGNETTE to support very diverse situations, writing bidirectional mappings from scratch is effortful and hard to reuse across similar situations. To avoid this complexity, other systems such as Codelets (Oney and Brandt, 2012) and mage (Kery et al., 2020) only allow to specify text templates with "slots". The slots can be written when the user interacts with the projection and read again when the code is directly modified. This solution has the advantage of being easy to use, but it is too simple to support a number of common situations. For example, it does not support creating slots in an unsorted key-value list—such as objects in Python or JavaScript—since all the possible orders would have to be enumerated.

To help users create projections for this kind of common situations, LORGNETTE includes its own type of templates. In LORGNETTE, a *template* reifies a procedure for creating a model containing key-value pairs. Each template specifies slots, which are not restricted to predefined ranges and whose meaning is completely template-specific. Each slot must have a *key*, which identifies the slot's value in the model, and an *evaluator*, which tells the template how to turn the text content of the slot into a usable value. A slot specification may also contain other information, such as a default value that should result in the slot's deletion in the code, e.g., to avoid cluttering a configuration object with default options. For example, a template for named arguments in Python lets users specify which function names they want to target, which arguments they are interested in (keys) and whether to parse each value into a string, a number, a boolean, etc. (evaluators). This template automatically generates resource specifications and mappings that result in a model where the keys are argument names and the values are evaluated argument values. Users are not restricted to use the model as is: templates can specify arbitrary functions to further transform the model into the shape expected by the user interface.

7.3.2 IMPLEMENTATION

LORGNETTE is implemented as a library written in TypeScript with React (@157). The source code of the library is open-source and has been made available on GitHub (@158). React allows to exploit the large ecosystem of React components to quickly experiment with new user interfaces for projections, but it is not a core requirement of the implementation. The current version of LORGNETTE includes a number of presets that can be readily used: five languages that can be parsed (JSON, CSS, Markdown, JavaScript/TypeScript and a subset of Python), six user interfaces (a colour picker, a table, a form, a file tree, a 2D plot and a regular expression diagram), three renderers (next to the code, in a popover and in a popup) and four templates (named groups in regular expressions, JSON objects, JavaScript objects and arguments of Python function calls). New languages, user interfaces and renderers can be registered via an API, just like specifications, allowing users to refer to them by name in a specification. At the moment, adding new templates still requires to modify the source code of the framework, but they could eventually benefit from the same kind of API. This would make it possible to write specifications in a very descriptive fashion, without having to write mappings in JavaScript, similar to what Vega Lite (Satyanarayan et al., 2017) and Varv (Borowski et al., 2022) propose.

I used LORGNETTE to instrument the Monaco editor (@159), which I then used to create two different code editing environments with malleable projections: a playground in the form of a webpage, and a custom editor for the Visual Studio Code (@146) editor (VSC). The playground includes a number of examples to try projections for different situations and in different languages. When the language can be parsed, the syntax tree of the document can also be displayed next to the code editor, a convenience for writing syntactic patterns. The custom VSC editor is a custom extension for VSC that replaces the standard code editor of VSC by a similar-looking code editor that can be augmented with projections. It consists of two parts: a webview that runs an instance of the Monaco code editor instrumented with LORGNETTE, and a core that gives the webview access to some of VS Code's extension APIs via message passing.

One of the major differences between the two environments is that the custom VSC editor supports runtime queries whereas the playground does not. To do so, the extension converts runtime queries emitted by LORGNETTE in the webview into messages sent to debuggers supported by VSC using the Debug Adapter Protocol (DAP, @88). For each runtime query, the core sets a breakpoint at the start of the code fragment of the projection that emitted the query. When the breakpoint is hit by the debugger, the core (1) retrieves the current thread and frame IDs, (2) uses them to ask the debugger to evaluate the runtime query's expression using an `evaluate` request, and (3) resumes the execution. When the debugger answers the request, the extension captures the response, pairs it with the corresponding query, and forwards it to the webview, so that LORGNETTE can update the projection that made this query. Since the DAP is designed to be language-agnostic, this mechanism works with multiple languages and debuggers without requiring any change to the code (besides writing runtime queries in the appropriate language). When testing this feature, I was able to successfully use runtime queries with VSC's JavaScript debugger, Google Chrome's JavaScript debugger and VSC's Python debugger.

7.3.3 COMPARISON WITH EXISTING SYSTEMS

To explain what makes LORGNETTE unique, I compare it to eleven other systems for editing computer languages that feature a text editor coupled with alternative projections of the code. The comparison focuses on two aspects of each system: the type of resources that can be projected, and the properties of the projections that complement the text editor. Table 7.3 lists the technical characteristics and the origin of each system, and Table 7.4 summarises the observations made below.

System	Implementation strategy	Target languages	Reference
Barista	Standalone	Any *	Ko and Myers, 2006
Pres. extension	Extension (Eclipse)	Java	Eisenberg and Kiczales, 2007
Graphite	Extension (Eclipse)	Java	Omar et al., 2012
Codelets	Extension (Ace)	Any	Oney and Brandt, 2012
Moonchild	Standalone	JavaScript	Dubroy (🔗126)
MPS	Standalone	Any *	Voelter and Lisson, 2014
Envision	Standalone	Java, C++	Asenov, 2017
Visual syntax	Extension (Dr. Racket)	Racket	Andersen et al., 2020
mage	Extension (Jupyter Notebooks)	Python	Kery et al., 2020
Livelits	Extension (Hazel)	Hazel	Omar et al., 2021
Vis. replacement	Standalone	Any *	Beckmann et al., 2023b

Table 7.3. Details on the systems LORGNETTE is compared to. For systems such as Barista and MPS, the comparison concerns the editors that can be created with the help of these systems, which are not code editors themselves. The implementation strategy designates whether it is implemented as an extension of an existing editor, which is specified between parentheses, or as a standalone program. The target languages designate the set of computer languages supported by the editor, or, in the case of generic extensions, those that can benefit from the extension. Systems that support "any" language only as long as the editor has been configured to support it, such as by specifying the language's grammar in a particular format, are marked with an asterisk (*).

System	Supported resources			Properties						
	Text	Syntax tree	Runtime	Persistent	Compositional	Types of malleablility				
Barista	–	Yes	–	Yes	–	–	–	–	–	P
Pres. extension	–	Yes	–	Yes	–	–	–	E	–	P
Graphite	–	Constructors	–	–	–	–	–	E	–	P
Codelets	Snippets	–	–	–	–	–	–	E	–	P
Moonchild	–	Yes	–	Yes	–	–	–	E	–	P
MPS	–	Yes	–	Yes	Yes	–	–	–	–	P
Envision	–	Yes	–	Yes	–	–	–	–	–	P
Visual syntax	–	Macros	–	Yes	Yes	–	–	E	–	P
mage	Yes	–	Yes	Partially	–	C	–	E	–	P
Livelits	–	Macros	Yes	Yes	Yes	C	–	E	–	P
Vis. replacement	–	Yes	Yes	Yes	Yes	–	–	–	–	P
LORGNETTE	Yes	Yes	Yes	Yes	–	C	R	E	–	P

Table 7.4. Comparison of LORGNETTE with eleven other systems that explicitly support projections in code editing environments. Orange values represent partially positive answers, and dashes represent negative answers. The five types of malleability correspond to the five levels of tailorability identified by Grønbæk et al. (2023): customisation (C), recomposition (R), extension (E), scripting (S) and reprogramming (P). The characteristics and origin of each system are given in Table 7.3. The meaning of the columns is explained in subsection 7.3.3.

Supported resources

I compared the type of resources that can be used to create projections: the text of the code itself, the nodes of the syntax tree that models it, as well as information only available at runtime. To the best of my knowledge, LORGNETTE is the only framework that supports all three.

Regarding the code itself, only LORGNETTE supports both textual and syntactic fragments. Barista, Graphite, presentation extensions, Moonchild, MPS, Envision, visual syntax, Livelits and visual replacements only support syntactic patterns, whereas Codelets and mage only support textual patterns. Some of these systems are even more restrictive regarding what they can project, as indicated by mentions in orange in Table 7.4. Graphite can only project class constructors, Codelets can only project predefined code snippets, and visual syntax and Livelits can only project specific macros, meaning that their projections are not compatible with certain types of code fragments or with existing code bases.

Regarding runtime data, only mage, Livelits, visual replacements and LORGNETTE support projections that rely on information issued from the execution of the code. mage's projections are invoked using special commands executed by the runtime environment. Since these commands can receive arguments, the runtime can evaluate them before forwarding them to the representation shown to the user, which can therefore benefit from that data as long as the user specified the right parameters when typing the special command. Livelits heavily rely on the concept of hole supported by Hazel, which enable partial evaluation of incomplete programs. Since Hazel automatically attempts to evaluate the expressions referenced by the projection, the projection may either be provided with a value (if the evaluation succeeded) or a special token representing the absence of a value (if the evaluation failed), leaving the responsibility of exploiting the values or using a fallback to the user interface. Visual replacements and LORGNETTE use a similar solution: they both support message passing between the code editor and a runtime capable of evaluating expressions that are part of the projected code, whose results are forwarded to the user interface. However, unlike visual replacements, LORGNETTE support language-agnostic projections regardless of the availability of a runtime (as long as the source of the projection contains no runtime resource). In contrast, with visual replacements, "*users need to define a language runtime mapping that informs the programming environment how programs can be executed*" (Beckmann et al., 2023b, §4.4) for each language the projection should be compatible with, regardless of whether that language runtime is used or not.[71]

71. Although I do not understand the purpose of that constraint, the fact that Beckmann et al. (2023b) write that "*users* need *to define a language runtime mapping [. . .]*" leaves little doubt that this is *always* required, even if that runtime mapping is never used.

This last point highlights that most of the systems I compare LORGNETTE to are either specific to a single language (7/11) or only partially language agnostic (3/11), as shown in Table 7.3, in the sense that they can project a computer language only as long as the editor has been configured to support it by, e.g., providing a parser capable of turning the code into a syntax tree. Besides LORGNETTE, only Codelets supports every language that can be edited as text out of the box, though it does so by not supporting syntactic fragments and runtime data. In contrast, LORGNETTE is as flexible as it can be: syntactic fragments and runtime data are supported, but as long as a projection does not rely on a syntactic pattern, the language does not need to be parsed, and as long as no runtime data is needed, connecting the appropriate runtime environment to LORGNETTE remains optional.

Persistence

Persistent projections remain available after they have been used once. LORGNETTE, like most other systems, offers persistent projections, as it automatically (re-)creates projections every time the text is modified. Graphite and Codelets' projections do not persist, as they are only designed to help users configure snippets of code when they are inserted in the text editor. mage's only partially persist, as they require users to manually evaluate a code cell to turn a magic command into a projection every time the code is modified—unless the user only modifies specific fragments of code that have been inserted by the projection as text, in which case it automatically parses the modified text and updates the projection's model.

Compositionality

Compositional projections can include other projections that were not hard-coded in their user interfaces. For example, a typical applications of compositionality would be to project text located in a text field of a form or in a cell of a grid onto a non-textual substrate, such as a slider or a colour picker, as demonstrated in Livelits. Currently, only MPS, visual syntax, Livelits and visual replacements support this property. At the moment, LORGNETTE does not support compositional projections because a projection can only be created from the content of the text editor. It could however be extended to let user interface declare temporary resources that can benefit from LORGNETTE's projections, such as the content of every text field.

Malleability

Malleable projections can be created, modified and deleted by their users. LORGNETTE is specifically designed to support malleability: users can create, modify and delete projections by editing specifications, which does not require modifying the source code of the text editor itself. Yet, as mentioned earlier and explained by Grønbæk et al. (2023), there are many ways for a system to be malleable. As a result, I must explain how LORGNETTE's approach to malleability differs from previous work.

Among the eleven systems I compare LORGNETTE to, all might be considered to be malleable in the sense that the projections they offer might be modified by editing the source code of the editor, recompiling it or regenerating it (if required), and using the newly created program in place of the old one. Yet, this form of malleability—called *reprogramming*—is not only highly exclusive, as it requires good programming skills, but also highly costly, even for seasoned developers, who may have to invest a significant amount of time to understand the code of the editor and determine how to implement what they would like to do. To let users add new projections without programming them, most (7/11) of the systems include an extension mechanism by which users can add projections that others have created to their code editor. Although this solution decreases the cost of customising the editor's projections, it does not help end-users create their *own* projections, as they must still rely on someone else's expertise.

To further lower the bar, LORGNETTE is designed to support *customisation* and *recomposition*, the two most accessible forms of malleability according to the ladder of tailorability presented earlier. First, LORGNETTE supports customisation, as user interfaces, renderers and templates can be parameterised by simple settings that the user can freely modify. For example, when selecting the side renderer, the user can set parameters to specify the relative position and the offset of the user interface. In addition, LORGNETTE supports recomposition by letting users freely combine user interfaces, renderers and templates to create new projections while writing as little code as possible to glue them together.

Although mage and Livelits also support customisation by letting users specify parameters when they invoke a projection, neither support recombination. Unlike LORGNETTE's specifications, those of mage and Livelits must be self-contained. They cannot refer to "building blocks" the systems can be extended with, such as by naming a user interface that was independently registered in the system without including it in the specification, therefore hindering reuse—one of the design goals I identified in the formative study.

7.4 CASE STUDIES

Since LORGNETTE is a framework for creating special kinds of user interfaces, I follow Olsen's recommendations for evaluating such systems (Olsen, 2007) and demonstrate what LORGNETTE can currently do by using it to conceive projections to help with five different situations. I use these case studies as a means to show that LORGNETTE fulfils Resnick et al.'s *low threshold, high ceiling, wide walls* goals (Resnick et al., 2005): simple projections can be created with low effort using recombination and templates; more advanced projections can be created by writing custom patterns and mappings; and multiple types of resources, computer languages and user interfaces are available out of the box. In this section, I present each of the five situations, explain how LORGNETTE helped me implement a projection for each of them, and highlight key differences with what could have been done with other systems.

Although technically, nothing prevents a programmer from creating a projection designed to support graphemic interaction, e.g., by projecting code on another textual substrate, or morphosyntactic interaction, e.g., by turning a fragment of textual code into a syntax-directed block editor, LORGNETTE is not the most appropriate solution for these levels of interaction. Since LORGNETTE is meant to augment existing text editors, they already provide a textual substrate for editing code at the graphemic level. Similarly, since targetting specific fragments of code within a single file strongly favours local projections, LORGNETTE is meant to help users interact with specific pieces of code representing specific concepts rather than replacing the textual substrate with a tree or a graph substrate whose structure represents the syntax of the language. To some extent, the runtime request mechanism is a sign that LORGNETTE's projections can help interact with the code at the pragmatic level. However, although LORGNETTE's projections can indeed exploit runtime information, they are hardly meant to interact with other resources than the code, unlike systems in which projections of the output itself can be directly manipulated, as in the examples shown in Figure 4.5. As a result, I chose to focus on creating projections that support semantic interaction and complement the graphemic interaction already supported by the text editor, for this is the most adapted use of LORGNETTE I envision.

7.4.1 MANIPULATING COLOURS

Colour pickers are one of the few projections that are found in several established programming systems. For example, they are available in professional code editing environments such as JetBrains' IntelliJ IDEA for Java and Microsoft's Visual Studio Code for CSS. Yet, even such a simple projection cannot be adapted to work in other situations than the one they were hardcoded for: users of these systems have no way to make the colour picker work with other languages and colour formats, even when their syntax is very similar.

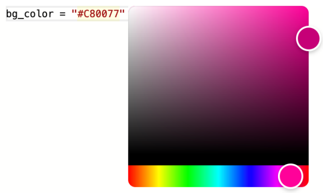
I used LORGNETTE to create a projection to view and manipulate colours encoded as hexadecimal colour codes such as `#c80077` using a colour picker (Figure 7.5a). It makes it easy to manipulate any such sequence of text as a colour, no matter whether it is a primitive value, a string, a comment, etc. Since it only relies on text, it is automatically available in every language, including languages that LORGNETTE cannot parse.

To create this projection with LORGNETTE, I used a regular expression pattern template that lets me create a slot for each named capture group of the regular expression. I created three slots, one for each successive pair of symbols following the initial # sign, that are each evaluated as numbers written in base 16. Finally, I added transformer functions that (1) wrap the RGB values into an object at the end of the forward mapping, to create a model with a `color` field, which is what the colour picker user interface expects; and (2) unwrap the modified `color` object at the beginning of the backward mapping, which is what the template expects.

Among the two other systems I compared LORGNETTE to that support textual patterns, neither can achieve the generality of this colour picker. Codelets only works with predefined code snippets, meaning that it could not project hexadecimal codes that are already written, e.g., in an existing codebase or in a piece of text copied from the internet; and mage's approach requires a runtime environment for executing code, whereas LORGNETTE makes it optional.

Once written, the specification of the projection can be adapted to support 3- and 8-digits hexadecimal colours, which are respectively used as a shorthand of the 6- digit version and to encode a transparency level, as well as other common notations such as `rgb(200, 0, 119)` and `hsl(324, 1.0, 0.39)`. Doing so would only require using templates with slightly different regular expressions and adding conversions between colour spaces in the transformer functions when needed. I used this approach to adapt the specification described above to create a colour picker projection for a custom `Color` function in JavaScript that accepts three numeric arguments. As demonstrated in Figure 7.5b, in order to adapt the first projection to this new

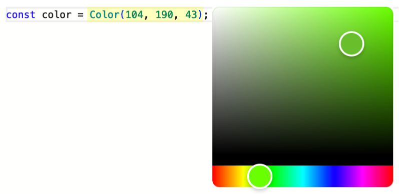
```javascript
const hexadecimalEvaluator = new NumericEvaluator({
  isIntegerValue: true,
  integerBase: 16
});

const template = new RegexTemplate(
  "#(?<r>[a-fA-F0-9]{2})(?<g>[a-fA-F0-9]{2})(?<b>[a-fA-F0-9]{2})",
  {
    "r": hexadecimalEvaluator,
    "g": hexadecimalEvaluator,
    "b": hexadecimalEvaluator
  },
  {
    transformTemplateModel: model => {
      return { color: model };
    },
    transformUserInterfaceModel: model => {
      return { ...model.color };
    }
  }
);

registerProjection({
  name: "Hexadecimal colour picker",
  ...template.resourcesAndMappings,
  userInterface: "color-picker",
  renderer: "side"
});
```



```
bg_color = "#C80077"
```

a. Hexadecimal colour codes (any language).

```javascript
const decimalEvaluator = new NumericEvaluator({
  isIntegerValue: true,
  integerBase: 10
});

const template = new RegexTemplate(
  "Color\\(\\s*(\\d+)\\s*,\\s*(\\d+)\\s*,\\s*(\\d+)\\s*\\)",
  {
    "r": decimalEvaluator,
    "g": decimalEvaluator,
    "b": decimalEvaluator
  },
  {
    transformTemplateModel: model => {
      return { color: model };
    },
    transformUserInterfaceModel: model => {
      return { ...model.color };
    }
  }
);

registerProjection({
  name: "Color function colour picker",
  requirements: { languages: ["javascript"] },
  ...template.resourcesAndMappings,
  userInterface: "color-picker",
  renderer: "side"
});
```



```
const color = Color(104, 190, 43);
```

b. `Color` function (JavaScript only).

Figure 7.5. Specifications (left) and results (right) of two projections for viewing and selecting a colour encoded as text, either for (a) 6-digits hexadecimal colour codes in any language (such as `#C80077`) or for (b) calls to a custom `Color` function in JavaScript. Lines in orange highlight the only changes required to turn the specification of the first projection into the specification of the second projections. They do not include changes in identifiers, which are only useful to keep the code understandable and have no effect on the resulting projection.

use case I only had to modify two lines of code—the base used by the numeric evaluator (line 3), the regular expression specifying the textual pattern (line 7)—and insert a new line—the language requirement to target JavaScript code only (line 25).

7.4.2 AUTHORING TABLES

Tables written in languages such as HTML, Markdown and LaTeX are notoriously tedious to manipulate as text. As an example, in all of these languages, cells are grouped by row, which means that inserting, reordering or deleting a column requires as many edits as the number of rows. In reaction to the need for a more interactive interface to create and modify such tables, authors have reported using spreadsheets to organise their data beforehand and online code generators to synthesise code in the appropriate language, as reported by several of the LaTeX users I interviewed.[72] Yet, these strategies increase the time needed to author these tables and the workload induced by repetitive switches between programs every time the table must be modified.

72. Subsection 6.2.2 describes these observations in more detail (see theme T3).

I used LORGNETTE to create a projection for manipulating Markdown tables using an interactive grid (Figure 7.6). The grid offers features similar to those offered by the aforementioned code generators: cells can be edited by double clicking on them, and rows and columns can be inserted and deleted using a contextual menu and moved by dragging their headers.

To create the projection, I used a syntactic pattern to target nodes representing tables. I then iterated over the row and cell nodes to collect the content of every cell into a two dimensional array to create the appropriate model in the forward mapping. I did the exact opposite to replace the table's content with the new model in the backward mapping, only with the help of a library to format a two-dimensional array as a Markdown table.[73]

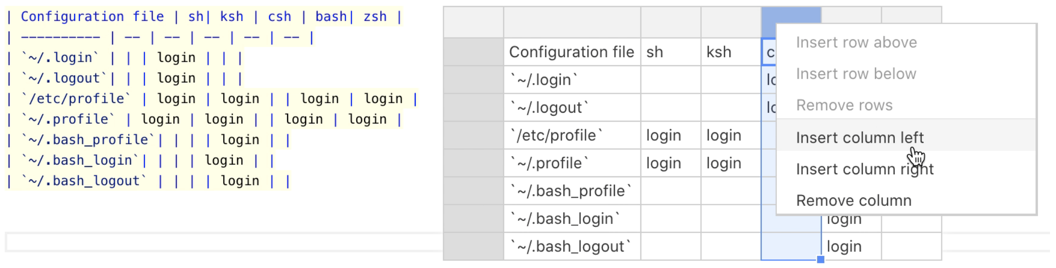
73. Since the Markdown parser used by LORGNETTE relies on `mdast-util-from-markdown` (@160), which complies with the *mdast* (@161) syntax tree specification, I could readily use `mdast-util-to-markdown` (@162) to serialise a syntax tree created from the two-dimensional array as text.

Projections for manipulating tables have previously been demonstrated in The Gamma (Petricek, 2020), mage and Livelits, but they were designed to interact with tables which that exist at runtime—such as data frames created by reading a CSV file—whereas authoring tables written in document description languages requires to modify tables written directly in the code. Software such as Adobe Dreamweaver also let users interact with HTML tables in a WYSIWYG fashion, but it is specialised for a single language and requires to interact with the formatted output instead of letting users focus on the content and the structure of the table. In contrast, LORGNETTE allowed me to easily create a tool to write and edit tables written in Markdown, akin to a non-embedded Markdown version of *i*-LaTeX's projection for LaTeX tables. Since tables usually have a clear hierarchy in syntax trees, creating similar projections for other document description languages by adapting this specification would likely be almost as straightforward as adapting the colour picker projection presented above to a different encoding.

7.4.3 WRITING REGULAR EXPRESSIONS

Regular expressions are a powerful mechanism for specifying textual patterns, but they must often be written in a domain-specific language embedded in a more general-purpose language, e.g., as a literal value or as a string,

```
const pattern = new SyntaxPattern(node => node.type === "table");

const forwardMapping = new ForwardMapping(({ fragment }) => {
  const tableNode = fragment.node;
  const tableContent = [];

  for (let rowNode of tableNode.childNodes) {
    const row = [];
    tableContent.push(row);

    for (let cellNode of rowNode.childNodes) {
      if (cellNode.childNodes.length === 0) {
        row.push(null); // Cell with no content.
        continue;
      }

      const text = cellNode.childNodes
        .map(node => node.text)
        .join("");
      row.push(text);
    }
  }

  return { content: tableContent };
});

const backwardMapping = new BackwardMapping(({ userInterfaceOutput, fragment, documentEditor }) => {
  const { cellChanges, content } = userInterfaceOutput;

  if (cellChanges) {
    const mdastTable = createMdastTable(content);
    documentEditor.replace(
      fragment.range,
      convertMdastToMarkdownString(mdastTable)
    );

    documentEditor.applyEdits();
  }
});

registerProjection({
  name: "Interactive Markdown table",
  requirements: { languages: ["markdown"] },
  pattern: pattern,
  forwardMapping: forwardMapping,
  backwardMapping: backwardMapping,
  userInterface: "table",
  renderer: "side"
});
```

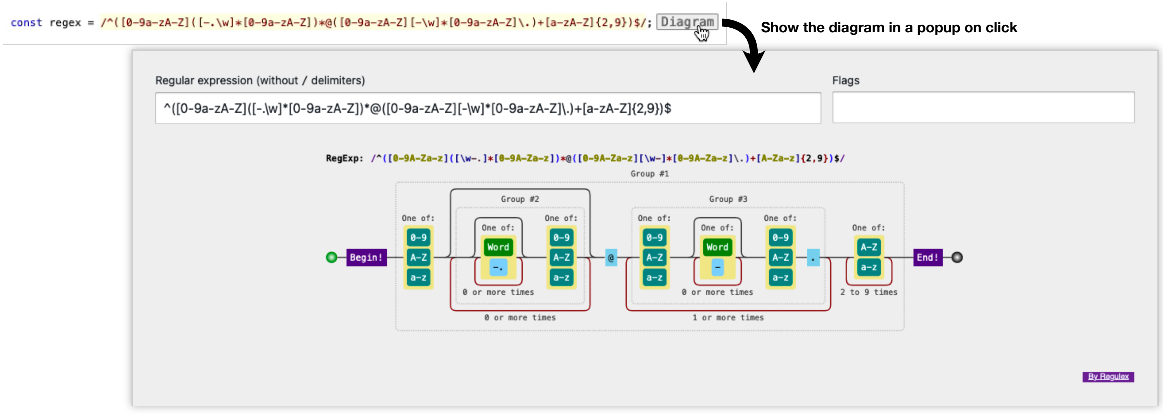


```
| Configuration file | sh| ksh | csh | bash| zsh |
| ---------- | -- | -- | -- | -- | -- |
| `~/.login` | | | login | | |
| `~/.logout`| | | login | | |
| `/etc/profile` | login | login | | login | login |
| `~/.profile` | login | login | | login | login |
| `~/.bash_profile`| | | | login | |
| `~/.bash_login`| | | | login | |
| `~/.bash_logout` | | | | login | |
```

Configuration file	sh	ksh	c		
`~/.login`			lc		
`~/.logout`			lc		
`/etc/profile`	login	login			
`~/.profile`	login	login			
`~/.bash_profile`					
`~/.bash_login`				login	
`~/.bash_logout`				login	

Figure 7.6. Specification (top) and result (bottom) of the projection for modifying the structure and the content of a Markdown table using an interactive grid.

```
const pattern = new SyntaxPattern(node => node.type === "RegularExpressionLiteral");

const forwardMapping = new ForwardMapping(({ fragment }) => {
  const regexAsString = fragment.text;
  const lastSlashIndex = regexAsString.lastIndexOf("/");
  const regexBody = regexAsString.slice(1, lastSlashIndex);
  const regexFlags = regexAsString.slice(lastSlashIndex + 1);

  return { regex: new RegExp(regexBody, regexFlags) };
});

const backwardMapping = new BackwardMapping(({ userInterfaceOutput, documentEditor, fragment }) => {
  const regex = userInterfaceOutput.regex;
  const regexRange = fragment.node.range;
  documentEditor.replace(regexRange, regex.toString());
  documentEditor.applyEdits();
});

registerProjection({
  name: "Literal regular expression diagram",
  requirements: { languages: ["typescript"] },
  pattern: pattern,
  forwardMapping: forwardMapping,
  backwardMapping: backwardMapping,
  userInterface: "regex-editor",
  renderer: {
    name: "button-popup",
    settings: { buttonContent: "Diagram" }
  }
});
```



```
const regex = /^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,9})$/;
```

Regular expression (without / delimiters)

^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,9})$

Flags

RegExp: /^([0-9A-Za-z]([\w-.]*[0-9A-Za-z])*@([0-9A-Za-z][\w-]*[0-9A-Za-z]\.)+[A-Za-z]{2,9})$/

Figure 7.7. Specification (top) and result (bottom) of the projection for displaying the railway diagram associated to a regular expression produced by Regulex (@93).

making them hard to use and hard to understand (Michael et al., 2019). Various techniques have been proposed to help users write regular expressions, ranging from synthesising them from positive and negative examples (Zhang et al., 2020) to explaining their meaning using visual augmentations in a text editor (Beck et al., 2014). Among these techniques, a popular approach is to explain the purpose of every symbol used in the regular expression, such as using nested blocks of text, as in RegExr (@94), or railway diagrams, as in Regulex (@93). Unfortunately, using these alternative representations requires to use a separate program which, just like code generators for tables, forces programmers to switch to a different program every time they need help authoring or analysing a regular expression.

I used LORGNETTE to create projections to display regular expressions as railway diagrams (Figure 7.7). More specifically, I created two projections that target JavaScript code: one for regular expression literals, e.g., `/ab*/g`, and one for the arguments of the regular expression constructor, e.g., `new RegExp("ab*", "g")`, whose user interface display the railway diagram created by Regulex.

To create the projections, I used two syntactic patterns to target the syntax nodes of the two contexts these regular expressions may appear in. I then separate the body of the expression from the optional flags by computing the appropriate substrings in the forward mappings. Since I did not find any open-source implementation of Regulex I could reuse, I created a custom user interface that embeds the Regulex website in an `<iframe>` element using a dynamically constructed URL that includes the regular expressions to visualise.

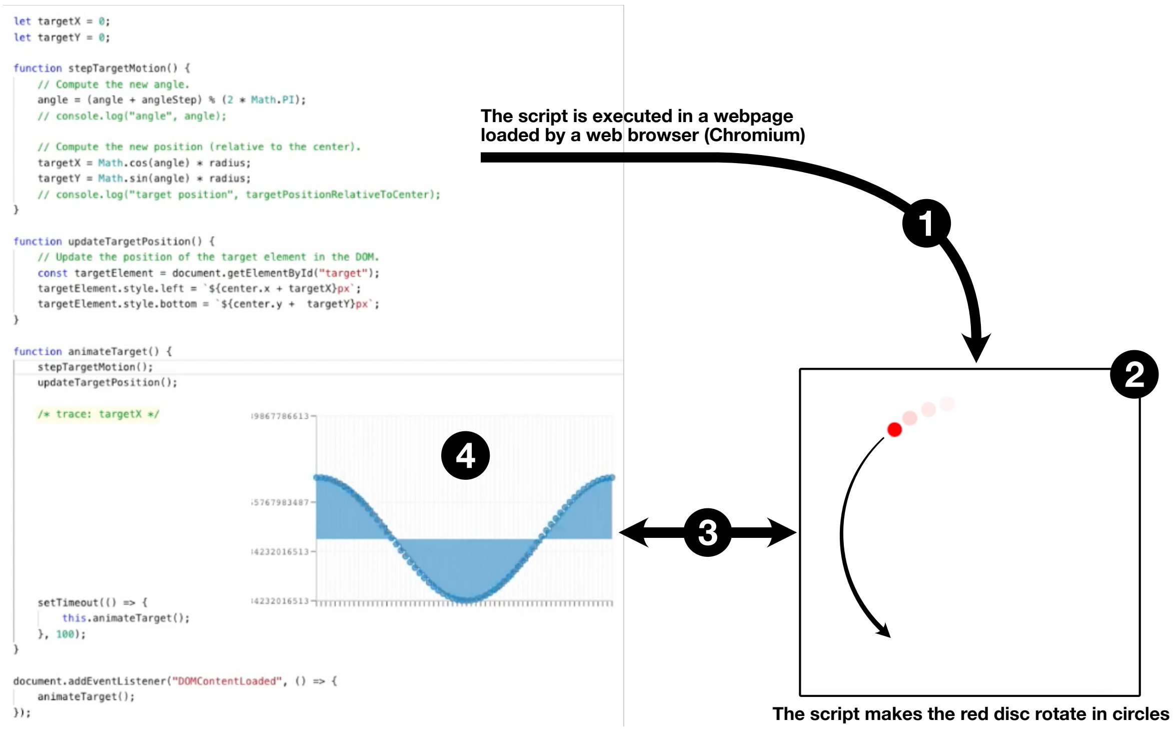
Since Regulex does not allow to modify the regular expression by interacting with the diagram, the projections I created do not either. Nonetheless, they demonstrate how LORGNETTE supports creating projections that embed online tools in a text editor in a contextually relevant fashion, therefore reducing the number of context switches, even if the source code of the tool is not publicly available. Again, since most languages use a similar regular expression dialect, the specifications could be reused to visualise regular expressions written in other programming languages simply by changing the target language and the pattern to search for in the code.

7.4.4 TRACING VARIABLES AT RUNTIME

Inspecting the current value of a variable is a staple of debugging. Displaying the values of the variables in the current namespace is a standard feature in many debuggers, and systems such as Light Table (@90) and Google Chrome's debugger can even display the value of expressions next to where they appear in the text. In addition, even when debuggers are available, many programmers also rely on print statements that they manually add to their code. Yet, none of these techniques allows to visualise how the value of a variable evolves over time or to compare it to previous values. At best, programmers are left with a sequence of raw values printed in a console that they must carefully analyse.

I used LORGNETTE to create a projection to plot the values taken by a variable on a 2D graph (Figure 7.8). The projection is attached to JavaScript

```
 1 const pattern = new RegexPattern("/\\*\\s*trace\\s*:\\s*\\w+\\s*\\*/");
 2
 3 const runtimeRequestProvider = new ProgrammableRuntimeRequestProvider(({ fragment }) => {
 4   const colonIndex = fragment.text.indexOf(":");
 5   const variableName = fragment.text
 6     .slice(colonIndex + 1, fragment.text.length - 2)
 7     .trim();
 8
 9   return [RuntimeRequest.createForFragment(fragment, "variableValue", variableName)];
10 });
11
12 const forwardMapping = new ForwardMapping(({ runtimeResponses }) => {
13   return {
14     valueChanges: runtimeResponses.map(response => {
15       return {
16         value: Number(response.content),
17         timestamp: response.receptionTime
18       };
19     })
20   };
21 })
22
23 registerProjection({
24   name: "Runtime value tracer",
25   requirements: { languages: ["javascript"] },
26   pattern: pattern,
27   runtimeRequest: runtimeRequestProvider,
28   forwardMapping: forwardMapping,
29   userInterface: "value-history",
30   renderer: {
31     name: "side",
32     settings: { onlyShowWhenCursorIsInRange: false }
33   }
34 });
```



```
let targetX = 0;
let targetY = 0;

function stepTargetMotion() {
    // Compute the new angle.
    angle = (angle + angleStep) % (2 * Math.PI);
    // console.log("angle", angle);

    // Compute the new position (relative to the center).
    targetX = Math.cos(angle) * radius;
    targetY = Math.sin(angle) * radius;
    // console.log("target position", targetPositionRelativeToCenter);
}

function updateTargetPosition() {
    // Update the position of the target element in the DOM.
    const targetElement = document.getElementById("target");
    targetElement.style.left = `${center.x + targetX}px`;
    targetElement.style.bottom = `${center.y +  targetY}px`;
}

function animateTarget() {
    stepTargetMotion();
    updateTargetPosition();

    /* trace: targetX */

    setTimeout(() => {
        this.animateTarget();
    }, 100);
}

document.addEventListener("DOMContentLoaded", () => {
    animateTarget();
});
```

9867786613
5767983487
4232016513
4232016513
4
The script is executed in a webpage loaded by a web browser (Chromium)
1
2
3
The script makes the red disc rotate in circles

Figure 7.8. Specification (top) and result (bottom) of the projection for plotting the values taken by a variable in JavaScript at runtime. In the example of use shown here, the projection is used to plot the successive X position taken by a red disc that moves in a circular pattern in a webpage. ❶ The script shown in the code editor on the left is executed in a webpage opened in the Chromium web browser. ❷ The webpage runs the scrips, which updates the position of the element represented as a red disc at a fixed time interval. ❸ Every time the execution of the code reaches the comment that benefit from a projection, Lorgnette instructs Chromium's runtime engine to evaluate the runtime request to evaluate the identifier representing the element's X position (line 9). ❹ Every time Lorgnette receives a runtime response, it applies the forward mapping again, which, in turn, updates the user interface's model with the newly received value (lines 14–19).

comments that start with `trace:` followed by the name of the variable whose value must be plotted. Every time the execution of the code reaches the comment, the current value of the variable is added to the plot.

The comments are targeted using a regular expression pattern. For each matching fragment, a runtime query is created in order to evaluate the expression formed by the variable's name, which is extracted from each fragment as a substring. Finally, the forward mapping maps the runtime response (which are automatically accumulated by the projection) to a model formed by a list of objects containing the values of the variable and the time at which it was modified, which is displayed by the graph user interface.

To test this projection, I connected the Visual Studio Code editor instrumented with LORGNETTE to a Chromium instance that was displaying a webpage running a script written in JavaScript that makes an element of the webpage move in a circular pattern. I then used the projection to trace the values of the variables representing the X and Y positions of the element after each update, using comments such as `/* trace:  targetX */`. The result of tracing the values taken by the variable containing the successive X positions is shown in the bottom-left hand corner of Figure 7.8.

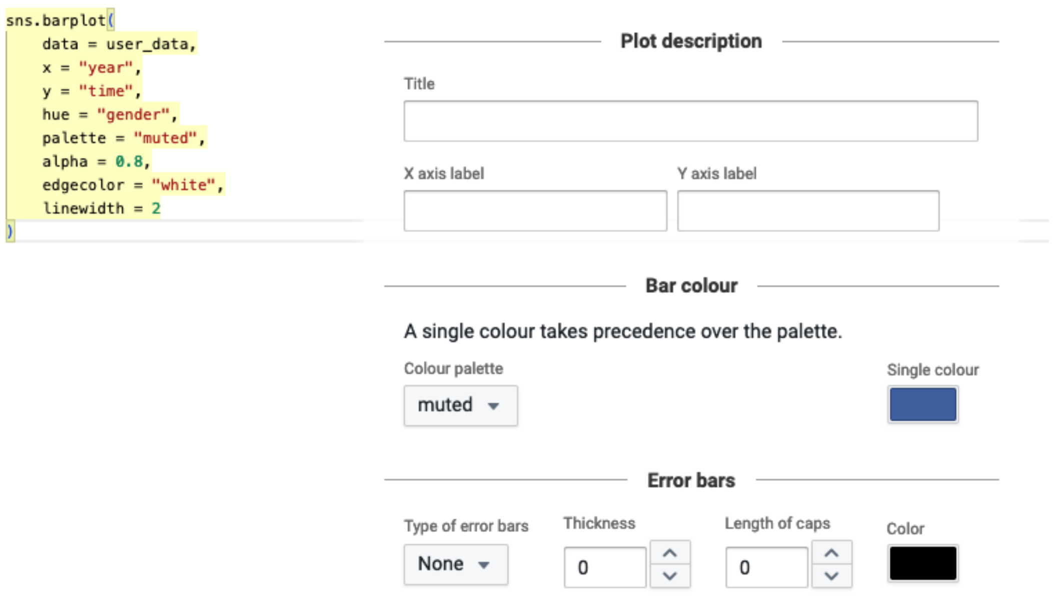
Multiple approaches to help programmers understand the dynamic aspect of their code have previously been proposed, as demonstrated by systems such as Whyline (Ko and Myers, 2008), Python Tutor (Guo, 2013), Omnicode (Kang and Guo, 2017), in-situ visualisations (Hoffswell et al., 2018), Reactive Inspector (Mogk et al., 2018), RxFiddle (Banken et al., 2018), Poker (Descheemaeker et al., 2021), Log-it (Jiang et al., 2023) and CrossCode (Hayatpur et al., 2023). Yet, although many of these approaches offer advanced interaction mechanisms to explore the runtime behaviour of a piece of code, they are not malleable and require users to use specific languages and specialised pieces of software. In contrast, LORGNETTE gives end-users the freedom to create their own debugging tools on the fly, within standard and widespread text editors such as Visual Studio Code. For example, with little effort, one user may decide to adapt the specification presented in this section to list the values taken by a variable in the table user interface shown in the previous use case instead of as a graph.

7.4.5 CONFIGURING LISTS OF PROPERTIES

It is common to specify a number of properties by hand when programming. This typically happens when configuring the graphical properties of a drawing primitive or a plotting function; when editing a configuration file or the front-matter of a document; and when configuring tools such as Pandoc and FFmpeg, as shown by several situations described by participants of the design workshop (R5, D10, D4). Since there is no systematic way to guess what properties can be configured, or which values can each of these properties take, specifying such properties usually requires to switch back and forth between the code and the documentation. This can be a particularly daunting task when there are numerous and complex properties to chose among, as in, e.g., multiple command-line tools and plotting functions from popular Python data visualisation libraries. For example, the documentation of the `barplot` function provided by the Seaborn library (@164) describes

```
const formContent = <>
  <Section title="Plot description">
    <StringInput
      formEntryKey="title"
      label="Title"
      defaultValue=""
    />
    <SingleRow>
      <StringInput
        formEntryKey="xlabel"
        label="X axis label"
        defaultValue=""
      />
      <StringInput
        formEntryKey="ylabel"
        label="Y axis label"
        defaultValue=""
      />
    </SingleRow>
  </Section>
  <Section title="Bar colour">
    <p>A single colour takes precedence over the palette.</p>
    <SingleRow>
      <Select
        formEntryKey="palette"
        label="Colour palette"
        items={["deep", "muted", "pastel", "bright", "dark", "colorblind"]}
        defaultItem="deep"
      />
      <ButtonColorPicker
        formEntryKey="color"
        label="Single colour"
        defaultValue={Color.fromCss("#3C5CA0")}
      />
    </SingleRow>
  </Section>
  <Section title="Error bars">
    <SingleRow>
      <Select
        formEntryKey="errorbar"
        label="Type of error bars"
        items={["None", "ci", "pi", "se", "sd"]}
        defaultItem="None"
      />
      <NumberInput
        formEntryKey="errwidth"
        label="Thickness"
        defaultValue={0}
      />
      <NumberInput
        formEntryKey="capsize"
        label="Length of caps"
        defaultValue={0}
      />
      <ButtonColorPicker
        formEntryKey="errcolor"
        label="Color"
        defaultValue={Color.fromCss("black")}
      />
    </SingleRow>
  </Section>
</>;
```

```
const template = PythonFunctionCallNamedArgumentsTemplate.createForFunctionNamed(
  name => name.endsWith("barplot"),
  [
    createSlotSpecification("title", FormEntryType.String, ""),
    createSlotSpecification("xlabel", FormEntryType.String, ""),
    createSlotSpecification("ylabel", FormEntryType.String, ""),
    createSlotSpecification("color", FormEntryType.Color),
    createSlotSpecification("palette", FormEntryType.String, "deep"),
    createSlotSpecification("errorbar", FormEntryType.String),
    createSlotSpecification("errcolor", FormEntryType.Color),
    createSlotSpecification("errwidth", FormEntryType.Number, 0),
    createSlotSpecification("capsize", FormEntryType.Number, 0)
  ],
  { ...FORM_DATA_TEMPLATE_TRANSFORMERS }
);

registerProjection({
  name: "Seaborn barplot style form",
  requirements: { languages: ["python"] },
  ...template.resourcesAndMappings,
  userInterface: {
    name: "form",
    settings: { content: formContent }
  },
  renderer: "side"
});
```



```
sns.barplot(
    data = user_data,
    x = "year",
    y = "time",
    hue = "gender",
    palette = "muted",
    alpha = 0.8,
    edgecolor = "white",
    linewidth = 2
)
```

Plot description

Title	
X axis label	
Y axis label	

Bar colour

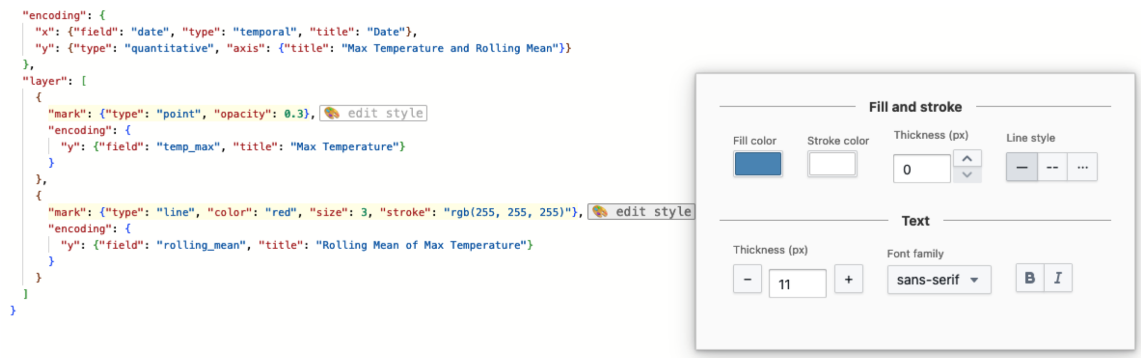
A single colour takes precedence over the palette.

Colour palette	muted
Single colour	🟦

Error bars

Type of error bars	Thickness	Length of caps	Color
None	0	0	⬛

Figure 7.8. Specification (top) and result (bottom) of the projection for configuring the style of Seaborn's bar plots in Python using a form.



```
  "encoding": {
    "x": {"field": "date", "type": "temporal", "title": "Date"},
    "y": {"type": "quantitative", "axis": {"title": "Max Temperature and Rolling Mean"}}
  },
  "layer": [
    {
      "mark": {"type": "point", "opacity": 0.3}, edit style
      "encoding": {
        "y": {"field": "temp_max", "title": "Max Temperature"}
      }
    },
    {
      "mark": {"type": "line", "color": "red", "size": 3, "stroke": "rgb(255, 255, 255)"}, edit style
      "encoding": {
        "y": {"field": "rolling_mean", "title": "Rolling Mean of Max Temperature"}
      }
    }
  ]
}
```

Figure 7.9. Demonstration of the projection for configuring the style of Vega-Lite marks using a form. The Vega-Lite code used in this example is adapted from the *Layering Rolling Averages over Raw Values* example available on Vega-Lite's website (🔗163).

over 30 different arguments, among which many are related to the plot's style. Moreover, additional keyword arguments are forwarded to the underlying `Rectangle` object provided by the Matplotlib library (🔗165), which accepts about 30 more arguments.[74]

74. These numbers of arguments respectively correspond to Seaborn version 0.13.2 and Matplotlib version 0.13.3.

Forms are a convenient tool for configuring properties without knowing what exactly is feasible in advance. Accordingly, I used LORGNETTE to create several projections to configure lists of properties using form-based user interfaces. More specifically, I focused on configuring graphical properties in two specific situations: when using the `barplot` function of the Seaborn library in Python (🔗166), as shown in Figure 7.8, and when describing a Vega plot in JSON (Satyanarayan et al., 2016), as shown in Figure 7.9. Each form includes several form elements to modify a number of properties, such as a the colour of the marks, the size of the text and the presence/absence and type of error bars.

For each projection, I used the appropriate template with one slot per property of interest. To conceive the user interfaces, I took advantage of the fact that LORGNETTE's form user interface can be entirely customised by describing the actual form in JSX (🔗167), directly from within the specification. The user interface provides form elements that can be automatically bound to a property of the model simply by assigning the key of the property' slot to the `formEntryKey` attribute. In addition, since the user interface can be specified using arbitrary JSX code, forms can include other kinds of elements, such as text and images, let the user customise the style of the form with CSS using `style` attributes, etc.

These projections support a form of exploratory programming: by exposing a number of settings, they make it easy to discover, try and compare different graphical styles until the output of the code is satisfying. By combining them with a continuous evaluation of the code and a live output, they can be used to create a live programming environment with projections offering user interfaces similar to those of style inspectors in WYSIWYG systems.

Although forms are very common in graphical user interfaces of code editing environment, they can rarely be entirely customised by end-users. For instance, forms are used to configure Ivy's templates for Vega (McNutt and Chugh, 2021), but they cannot be customised nor adapted to other languages. Codelets and Visual syntax for Racket have both demonstrated that text could be projected onto customisable forms. However, neither is as general as LORGNETTE: the former only supports binding forms to regions of predefined code snippets, whereas the latter can only we used with the Racket language.

7.5 CONCLUSION

In this work, I demonstrated that standard text editors can be augmented so as to provide alternative projections of code that is usually represented only as text that end-users can create and modify. The formative study, in which I collected 62 projections sourced from the literature and a design workshop, resulted in two key observations: there exist a wide variety of projections of computer languages, which are used in very diverse contexts and for very different languages; and yet, almost two thirds of them (62%) reuse one of three kinds of user interfaces—a form, a graph or a grid. Since all current approaches to tailor a text editor with new projections of the text require reprogramming, i.e., inducing a high level of friction that is likely to discourage most users, including those with programming skills, I developed LORGNETTE, a framework for instrumenting text editors so as to facilitate the development of malleable code projections.

LORGNETTE is designed to decrease the friction for tailoring one's text editor: ready-made projections can be shared and installed to extend the editor; templates, user interfaces and renderers can be freely recombined to form new projections; and custom patterns, mappings and user interfaces can be created using scripts. I successfully used LORGNETTE to create projections that help understand and/or modify code in five different situations, each time in a few dozens of lines of code only. I demonstrated that LORGNETTE can be used to create and adapt simple projections such as colour pickers with very little work (low threshold), as well as craft more complex projections such as debugging probes and configuration forms with some scripting (high ceiling). Overall, the five use cases demonstrate LORGNETTE's capacity to map different types of data (text, syntax nodes, runtime data) onto different sorts of user interfaces, whose arbitrary recombinations yield very diverse projections (wide walls).

Yet, despite its successful use, LORGNETTE also suffers from a number of technical limitations. One such limitation is that LORGNETTE can only search for fragments in one file at a time. This prevents LORGNETTE from supporting composite code fragments formed by multiple code fragments split across several files, which are required, e.g., to modify a CSS file by interacting with a table described in a HTML file (R9) or to create and grade code assignments where questions and answers are located in separate files (D14, D16, R2, R3).

Another limitation is that LORGNETTE is more tailored for projections that are local, both regarding the code and the runtime data, making it ill-adapted to create projections that represent global information, such as displaying the call graph or the stack and the heap of a program, as in Reacher (LaToza and Myers, 2011) and the Online Python Tutor (Guo, 2013). I also did not address the challenge of creating a renderer able to embed projections in other documents, as *i*-LaTeX does with transitionals embedded within the PDF generated by the code, therefore preventing the creation of projections for manipulating images directly in the output (D13). This leaves room for future opportunities to link code with locations in other documents in a language-agnostic fashion, possibly inspired by existing solutions such as SyncTeX (Laurens, 2008) and Source Maps (@168).

Overall, each paradigm for authoring code editors with protean projections has advantages and weaknesses, including LORGNETTE. Yet, I argue that adding support for non-textual projections of the code in regular text editors in a malleable fashion, so as to give end-users the ability to tailor their editors without having to reprogram them entirely, is key to spreading the use of protean projection. Unlike other approaches, LORGNETTE achieves these goals in a somewhat conservative way, making it readily compatible with existing text editors and all of the most popular computer languages. Just like the success of open-source software and of the internet owes in great part to the freedom they offer, I believe this also holds true for code projections, which could greatly benefit from being more malleable.

8

Discussion

In the previous chapters, I successively motivated (chapter 2) and introduced (chapter 3) a new holistic model to frame and reflect on what being and interacting with a computer language means, leading to a new taxonomy of levels of interaction with computer languages (chapter 4) and to the key concept of projection (chapter 5). Inspired by this broader vision of what interaction with a computer language can be, I then designed, implemented and evaluated additional projections for computer languages that are primarily edited as text through my work on *i*-LaTeX (chapter 6) and LORGNETTE (chapter 7). Together, they constitute the main contributions presented in this thesis. Yet, a link has yet to be established between the theoretical and practical aspects of ny work. Further, as the choices I made were directed towards exploring and showcasing a specific subset of the design space supported by the theory, I also need to clarify what I purposely left apart, leaving various open questions and opportunities for future work.

This chapter contributes these two missing links. First, section 8.1 summarises and connects the theoretical contributions I made in chapters 2 through 5 and the applied contributions I made in chapters 6 and 7. It shows that thinking about interaction with computer languages in terms of projections yields a design space of interaction techniques that could not be captured by previous theories, yielding novel systems that I implemented and demonstrated. Then, section 8.2 presents five limitations of this work, ranging from questions I could not address in the user studies that I carried out to incomplete aspects of the theory, from which I derive a number of directions for future work.

8.1 CONTRIBUTIONS

Throughout this thesis, I introduced ideas, artefacts and studies of different natures. Although each was motivated by internal or external factors, such as a consequence of the chapter before or an observation found in the literature or in my own experience, they are loosely connected to each other. This section connects the dots between these contributions to draw the bigger picture and show how they indeed compose the two parts of my work, as announced in the introduction: introducing a new theory of interaction with computer languages, and applying the theory by extending text editors compatible with established computer languages.

8.1.1 A THEORY OF INTERACTION WITH COMPUTER LANGUAGES

Motivated by the lack of a theoretical frame for analysing and designing our interaction with computer languages, besides limited views such as the classic textual/visual distinction often found in HCI research, I started by presenting a more holistic model of what being a computer language means (Figure 3.1). According to this model, which I derived from the definition I gave in section 2.1 and a single axiom (we reason using concepts), every computer language can be decomposed into five aspects: a set of concepts that are part of the language itself, or expressed with the help of the language (conceptualisation); a specification including an alphabet of symbols and a set of rules for deciding how to combine and interpret symbols (formalisation); a scheme to code and decode the language so as to implement computer programs able to read and write the language (implementation); a collection of substrates, gathered in code editing environments, to let us perceive and act upon code encoded in a computer's memory (interaction); and relations with other conceptual and physical entities at both computational and sociocultural levels (contextualisation).

I then derived two consequences from this model of computer languages. First, I used this model to establish a new taxonomy of interaction with computer languages formed of four levels, each roughly giving us access to a different aspect of the language (Figure 4.1). It posits that substrates let us interact with code in four different ways: by interacting with the encoding itself (graphemic interaction); with the symbols and the structures of the code (morphosyntactic interaction); with the concepts we associate to the code (semantic interaction); and with other artefacts related to the code (pragmatic interaction). Given the central role given to substrates, I then focused on qualifying the action of pairing code (and other resources) with a representation, which I call a *projection*. I defined multiple properties of substrates (subsection 5.2) and reported on two competing strategies when projecting a computer language (subsection 5.3), using a single substrate (uniform projection) and using multiple substrates (protean projection).

Taken together, these three theoretical contributions—decomposing the notion of computer language into five aspects, deriving four levels of interaction and describing the central role of projection in our interaction with computer languages—form a new theory of interaction with computer languages. This theory is *descriptive*: it defines a series of interrelated concepts to reason about computer languages, and can be used to analyse and classify systems and techniques for interacting with computer languages—as demonstrated in chapter 4—without attempting to predict any metric nor prescribe any design choice or human behaviour. It also has a certain *generative* power, in the sense conveyed by Beaudouin-Lafon et al. (2021), as it can help identify alternative or additional modalities to interact with a computer language, and therefore help generate new design artefacts, as illustrated by the two concrete applications of the notion of projection that I developed.

8.1.2 A POSTMODERN APPLICATION OF PROTEAN PROJECTION

Contemporary research in protean projection of computer languages can be roughly divided into two categories: modern approaches and postmodern approaches. Modern approaches cut loose with existing technologies and advocate for their own instead, such as a specific runtime environment or language, which are often required to implement the ideas they present. Examples include model-based engineering tools and language workbenches such as Barista (Ko and Myers, 2006) and JetBrains MPS (Voelter and Lisson, 2014); siloed image-based systems such as Lively (@169) and the Glamourous Toolkit (@122); and niche programming languages and environments such as Sketch-n-Sketch (Hempel et al., 2019) and Livelits (Omar et al., 2021). On the contrary, postmodern approaches embrace established computer languages and widespread technologies and aim at being compatible—rather than at odds—with them, even though this makes the approach more restrictive than its modern counterpart in terms of, e.g., what resources can be projected and what type of substrate can be used. Examples include code editing environments centred around text that are extended with additional projections of the code, such as Graphite (Omar et al., 2012), Codelets (Oney and Brandt, 2012) and mage (Kery et al., 2020).

Given how popular established computer languages and text editors remain in the 2020s, even though alternatives have existed for over 40 years, I chose to apply the theory in a postmodern fashion, leading me to create code editing environments compatible with several computer languages that are massively used all over the world. To achieve this goal, I used user-centred design methodologies to ground my design process in knowledge gained from formative studies (interviews, design workshop) and evaluated the outcomes of my work in both quantitive and qualitative fashions (controlled experiment, longitudinal study). To that end, I focused on two user groups, each corresponding to one of the five major purposes computer languages are used for that I reported in section 2.2: users of LaTeX, a specific document description language, and users of diverse programming languages. This led me to work on two different projects, each complementing text editing with additional projections of the code in different ways: *i*-LaTeX and LORGNETTE.

Each of these systems is designed to complement the projection of code onto a textual substrate—which is a defining characteristic of all text editors—with additional projections mainly designed to help users interact with concepts expressed in the code, i.e., at a semantic level of interaction. The example projections I implemented and demonstrated with *i*-LaTeX and LORGNETTE share a number of properties: they are all local, mostly bidirectional, live and persistent. They also differ in terms of location, malleability and language agnosticism. *i*-LaTeX's transitionals are specific to LaTeX; they are embedded within a projection of the document generated by the code, not within the code itself; and they are not designed to be malleable at all. On the contrary, LORGNETTE's projections can either be language-agnostic or language-specific; they can be located within the textual substrate representing the code (inline or floating) and might be made standalone elsewhere in the user interface (with some extra work), but could hardly be embedded in another document at the time of writing; and they are purposely engineered to be customisable and recombinable. As such, each system demonstrates what can be done in two different regions of the design space of projections induced by the seven properties of projections presented in section 5.2, therefore serving as two complementary implementations of the protean projection strategy identified in subsection 5.3.2.

Overall, my work on *i*-LaTeX and LORGNETTE shows that thinking about interaction with computer languages in terms of projections has multiple advantages and yields systems that would hardly fit in the traditional textual/visual split that is predominantly used in HCI research on computer languages. By nature, protean projection helps going beyond text-centred interaction techniques without replacing it altogether, therefore avoiding issues faced by syntax-oriented editors which, even in the 2020s, suffer from engineering and usability issues that seem hard to tackle.[75] As a result, users of computer languages can still use text editors while benefiting from other projections in specific situations, which, in the case of *i*-LaTeX's evaluation, proved to help LaTeX users perform various tasks faster, with less compilations and with a lower workload, without restricting them from writing full-fledge LaTeX documents and alternate between different editors. Furthermore, LORGNETTE's use in five situations implying real-world computer languages highlights that creating and adapting projections on-the-fly is possible, even for established languages, making it a suitable approach to push the end-user tailorability of text editors further, in line with the goals of recent research on malleable software (Borowski et al., 2022; Grønbæk et al., 2023).

75. For example, understanding how to use the cursor appears to be far less obvious in syntax-directed editors than in regular text editors, as suggested by the discussion on this topic in Dimitar Asenov's thesis (2017, ch. 6), the complex operations associated with cursors in Forest (Voinov et al., 2022) and Tylr (Moon et al., 2022), and the extra engineering efforts put into addressing this issue in Sandblocks (Beckmann et al., 2023a), in which "*a modifed parser [...] informs our reconciliation process of how users' changes can be made compatible with the language grammar*" (§3.2, p. 4) to help users write code as text even when it spans across syntax nodes of different types. Yet, even with such efforts, Sandblocks' evaluation reveals that "*compared to conventional text editors, [...] participants only took on average 21% (JS), 34% (Clojure), and 95% (RegExp) longer*" (p. 1).

8.2 LIMITATIONS AND FUTURE WORK

Despite their demonstrated benefits, my work on *i*-LaTeX and LORGNETTE suffers from multiple limitations, beyond the technical ones already mentioned in the respective chapters. By choosing to focus on specific user groups, making specific design choices, and evaluating specific aspects of these systems, I naturally ignored other possibilities, some of which have already been studied in previous work, and some of which remain to be investigated by future research. In addition, the theory of interaction with computer languages itself shows limitations, and so does my own position on what a computer language is and what interacting with it means, as it evolved greatly since I started writing this thesis. As a result, this section presents five limitations that I identified in this work and shows possible directions to go beyond them in the future, both by extending my own work and by exploring alternative paths.

8.2.1 BEYOND COMPARTMENTALISED USES

The two evaluations of *i*-LaTeX's transitionals and the five case studies with LORGNETTE demonstrate how projections complementary to text can help users perform specific tasks, such as selecting a colour, editing a table and understanding a regular expression. Yet, all these contributions focus on the benefits of using a single projection in a specific context, leaving unclear if and how users would integrate them into larger workflows. Although studying this question was one of the main goals of *i*-LaTeX's longitudinal study, the prototypal nature of the editor given to the participants and the limited sample size did not lead to any statistically sound conclusion. A few examples, however, hint at potential synergies between textual projections and other projections. For example, one participant of *i*-LaTeX's controlled experiment was able to complete a task very efficiently by switching between the textual and grid projections of LaTeX code to sort the rows of a table by a certain column, as explained in subsection 6.5.2. Similarly, when using LORGNETTE in different situations, I often found myself switching between a colour picker or a form user interface to configure a single property and the text editor to copy and paste the value I just set in other locations. Such situations illustrate the complementary nature that two projections can have, although the situations in which this happens and the way it affects users remain unclear.

Future work may study how users use multiple projections together and over time, beyond the frame of a specific operation. This includes studying why users prefer one projection over another in a variety of situations; what is the cost of switching between two or more substrates to perform a single code understanding or editing task; and how to help users transition from one substrate to another, such as by suggesting another substrate when appropriate, highlighting different representations of the same piece of code in two separate substrates, and displaying—or even animating—the transformations that connect one to the other, akin to how Gliimpse (Dragicevic et al., 2011) animates how pieces of LaTeX code are mapped to visual elements in the generated PDF. Beyond better understanding how projections of computer languages are use in real situations, such research may benefit other domains in which users commonly deal with multiple representations of the same piece of data. For example, this appears to be common in data analysis and visualisation software such as Microsoft Excel, SAS JMP and Jupyter notebooks, in which users frequently use a mixture of data tables, plots and code in the form of queries or scripts to visualise and manipulate the same underlying pieces of data.

8.2.2 BEYOND LOCAL PROJECTIONS

i-LaTeX's transitionals and LORGNETTE's applications each showcase the use of various substrates, for various computer languages, in various situations. However, all of them are local projections: they only project restricted and continuous fragments of code, always located in a single file. While this proved to be already helpful for some tasks, such as previewing and modifying a colour encoded as a contiguous sequence of characters, it is not adapted situations in which multiple fragments of code (or other pieces of information, from other resources) have to be combined into a single projection.

LORGNETTE's formative study highlights a number of projections that were of interest to participants of the design workshop and/or implemented in other systems published in the literature, but could not be implemented nor studied using *i*-LaTeX and LORGNETTE—at least not in their current version. For example, I did not implement nor study projections for editing CSS properties located in a different file next to a piece of HTML code they apply to, which is standard practice in front-end web development, as needed by designs D11 and R9. Similarly, applications such as explaining an optimisation problem from the code that describes it (design D2) and visualising how functions or objects access each other over time (design D7 and D17; LaToza and Myers, 2011) require to project information that is global to a project, possibly spanning over hundreds of files, far from the restricted scope of the projections I studied with *i*-LaTeX and LORGNETTE.

Studies of how programmers use whiteboards and sketches (Cherubini et al., 2007; Mangano et al., 2015) further motivate the importance of providing tools to help programmers transfer analog practices into digital systems, so as to directly being able to refer to and transform code beyond fragments that are only a few lines long. Code thumbnails (DeLine et al., 2006), now available in text editors such as Sublime Text and Visual Studio Code, displays a textual substrate from afar to let users rely on their spatial memory to navigate files shown as text. CodeMap (Taniguchi and Masuhara, 2022) lets users create diagrams with nodes linked to the code and switch between the diagram and the text editor, but it only allows them to use the diagram to take notes, with no semantic meaning nor any way to edit the code from the diagram. Interfaces such as Code Bubbles (Bragdon et al., 2010) and the Debugger Canvas (DeLine et al., 2012) hint at possible ways to combine multiple projections to give a more global view of a code base, but they only demonstrated it by combining multiple textual substrates linked to each other. At an even larger scale, systems such as OverCode (Glassman et al., 2015), WEVL (Taniguchi et al., 2022) and VizProg (Zhang et al., 2023) explore how to visualise thousands of solutions to a programming problem submitted by students by proposing different ways to cluster and compare pieces of code before mapping them onto a substrate such as a 2D plot.

Future work may not only go beyond studying local projections, but also go beyond overly global ones too, by rather investigating the continuum that exists between them, and the different requirements that come with each scale, both in terms of implementation and interaction. Regarding implementation, this raises the question of getting the appropriate information when it is scattered across multiple files. For example, how could one scale up the malleability of LORGNETTE's custom search patterns for fragments, given that indexing and searching an entire code base may be prohibitively costly in terms of time and memory.[76] Regarding interaction, future work could address the challenge of working with combinations of substrates representing information at different scales, such as a text editor showing an extract from a single file next to a call graph showing how different pieces of code located in different files relate to each other during the execution of the code. User studies may help understand what is hard, and future design and implementation work may help identify how to address such difficulties. The challenge of working at a larger scale echoes, once again, those faced by data analysis and visualisation experts, who are facing increasingly large datasets and possibly left with no better solution than black-box statistical models such as neural networks to extract meaningful information from them.

76. Although the scale is not comparable, Clem and Thomson (2022) discuss the challenge of indexing and querying code bases in an efficient manner by relating their own experience with providing static analysis of repositories hosted on GitHub.

8.2.3 Beyond hand-crafted mappings

In both *i*-LaTeX and lorgnette, the mappings of the projections—that is, the instructions that tell the computer how to map a set of resources onto a representation, and conversely—were entirely written by hand. While this was the most straightforward choice, and sometimes an easy solution, such as when splitting an hexadecimal string into three parts and parsing each of them as a base-16 number, it quickly proved to be challenging. For example, implementing *i*-LaTeX's transitional for tables required me to deal with table formatting macros that must be ignored but preserved in the text, such as `\toprule` and `\hline`, as well as rules regarding whitespace and line breaks.[77] Similarly, in an experiment with lorgnette not reported in this thesis, mapping a few CSS properties onto a form user interface required me to write several hundred lines of code, in part to account for ambiguities that sometimes arise, such as when the user modifies the colour of a border using a form and the code includes both the atomic `border-color` property and the composite `border` property.

77. Although whitespace and line breaks may be regarded as purely secondary notation which may be discarded altogether or reinserted using a code formatting program rather than hand-craftd rules, I argue that they also embed a form of idiosyncratic knowledge about code, sometimes specific to a community or a language, which may be misunderstood by such programs and result in users disliking a projection because it messes up the formatting they adhere to.

As a result, writing mappings quickly appeared to me as one of the main challenges to broaden the development—let alone by end-users—of ad-hoc code projections when needed. The problem, which appears in many forms in the literature, ranging from the old-school *view-update* problem introduced in the database community to the more recent notion of *lens* in programming theory and functional programming (Foster et al., 2007), has no definite solution. Yet, a number of directions have already been suggested to ease the difficulty of writing mappings, forming a gradient of potential solutions. The most simple approach consists of forming bindings between ranges of text and properties of the other projection's model, either by manually labelling the ranges or using a regular expression to find them, as used in systems such as Codelets (Oney and Brandt, 2012) and mage (Kery et al., 2020). A more advanced approach consists of treating this task as an optimisation problem and using algorithms to explore the solution space. This is the approach used in systems such as SnipPy (Ferdowsifard et al., 2020) and Falx (Wang et al., 2021), which synthesise code using a generative grammar of (a subset of) the language, and in several variants of Sketch-n-Sketch (Mayer et al., 2018; Hempel et al., 2019), which can trace the provenance of the result of a computation to help an algorithm identify what must be modified to account for a change in the output. However, all these implementations have only tackled a highly-constrained search space and small-scale code synthesis, far from the hundreds of lines of code with a single input example and no output example that correspond to the situations I described above.

Future work may keep investigating how to go beyond crafting complex mappings by hand. One possible direction is to pursue the work on syntax- and trace-directed code synthesis. However, it may be hard to use such techniques in the wild rapidly, as they usually require to create new computer languages with specially crafted specifications. This approach faces the same difficulties of widespread adoption that I mentioned in section 5.4 to support my choice of making a postmodern application of protean projection in my work. To apply it to larger-scale problems *and* established computer languages, the recent surge in large language model-based code synthesis and the impressive results of models such as GPT-4—which has successfully generated entire programs such as a simple 3D game given a textual description and working LaTeX code given a partially formal description (Bubeck et al., 2023, §3)—seem to point towards the key role AI methods may soon play in this area. Ongoing advances in artificial intelligence techniques may therefore greatly decrease the cost of writing mappings on the fly, therefore helping users create their own tools to understand and write code, instead of merely letting a model write it without any human supervision.

This view for a future where code synthesis complements code understanding and writing techniques, rather than replacing them, echoes the positions of other researchers in this domain. Maneesh Agrawala (@170) argues, for instance, that black-box models at the core of state-of-the-art synthesis techniques in use today "*do not provide a predictive conceptual model*", therefore making them "*terrible interfaces*" for interaction. In a similar stand, Geoffrey Litt (@171) envisions large language models as companions for tailoring software to one's needs, capable of making local, on-demand changes to a program to account for a missing feature or provide a context-specific user interface. If they indeed succeed, artificial intelligence models such as LLMs may therefore not only help writing projections' mappings, but also benefit from the resulting projections themselves, either as tools to configure the synthesis itself or as tools to understand and modify the code they output.

8.2.4 BEYOND SEMANTIC INTERACTION

By decomposing the notion of computer language into five aspects, I highlighted that substrates let us interact with the four other aspects, yielding four levels of interaction: graphemic interaction, morphosyntactic interaction, semantic interaction and pragmatic interaction. Although, historically, code editors used to focus on a single level of interaction, such as graphemic interaction in text editors from the 1960s and morphosyntactic interaction in visual programming systems from the 1980s, they can also be combined into a single code editing environment, making our interaction with computer languages protean. By focusing on complementing text editors with other projections, *i*-LaTeX and LORGNETTE allowed me to focus on the combination of graphemic interaction with semantic interaction.[78] As a result, it leaves the other combinations permitted by protean interaction out of the scope of my work.

78. One could argue that I did not strictly focus on semantic interaction, as exemplified by the use of a typeset formula that looks like the output in *i*-LaTeX's transitional for mathematical formulae and LORGNETTE's concept of *runtime query*, which I used to trace a variable at runtime. Both allude to pragmatic interaction. Yet, the first example is only a reproduction of the output for lack of a better representation, and the second example uses the runtime as a source of the projection, but with the intention of displaying the sinusoidal behaviour of a variable within the code, rather than actually interacting with the output itself.

Besides the combination of graphemic and semantic interaction, which was also studied in a number of systems, such as Graphite (Omar et al., 2012), Codelets (Oney and Brandt, 2012), mage (Kery et al., 2020) and Livelits (Omar et al., 2021), a number of other combinations have been investigated in the literature. By mixing text editing and direct manipulation of pluggable blocks, hybrid block programming environments such as GP (Monig et al., 2015) and Pencil Code (Weintrop and Wilensky, 2017) combine graphemic and morphosyntactic interaction. By supporting domain-specific projections of syntax tree nodes, beyond text as a default, mbedrr (Voelter et al., 2019) and Sandblocks with visual replacements (Beckmann et al., 2023b) combine morphosyntactic with semantic interaction. By letting users view and edit code as text, as well as by directly manipulating its visual output, Sketch-n-

Sketch (Hempel et al., 2019) demonstrates the combination of graphemic and pragmatic interaction. Similarly, by supporting text editing and local evaluation of code snippets from example values, babylonian-style programming (Rauch et al., 2019) and projection boxes (Lerner, 2020b) achieve another kind of combination between graphemic and pragmatic interaction.

Future work may keep implementing and studying such two-way combinations between levels of interaction with computer languages, as well as explore new ones. For example replacing Sketch-n-Sketch's text editor with Deuce (Hempel et al., 2018) would yield a code editing environment that combines graphemic interaction (editing code as text), morphosyntactic interaction (refactoring code by direct manipulation of syntactic constructs), and pragmatic interaction (transforming code by transforming its visual output). Further adding projections of certain concepts, such as for configuring graphical properties using dedicated widgets,[79] would also let user interact with code semantically, effectively combining four levels of interaction with a single computer language in the same environment.

79. Sketch-n-Sketch already includes a colour picker widget, but it can only be used to modify a colour of a shape in the output, rather than a piece of code representing the concept of colour.

Besides attempting to combine levels of interaction in new ways, future work may also study which combinations work well together and which do not, and why. Interestingly, all the combinations mentioned above combine graphemic and/or morphosyntactic interaction with semantic and/or pragmatic interaction. It may be that, because the first two levels are the only ones that give access to an entire computer language as an abstract symbolic system, they are by nature complementary to semantic and pragmatic interaction, which are too focused on a specific concept or use of the language to let users exploit it all. It may also be, as I discuss below, that this observation itself shows a limitation of my theory of interaction with computer languages, at least in the way it is phrased in this thesis.

8.2.5 BEYOND COMPUTER LANGUAGES

In the limitations presented so far, I propose directions to explore alternatives to the choices I made in my own application of the theory of interaction with computer languages introduced in this work. As such, they all stay within the boundaries of that theory, and they are all compatible with its glossary of concepts. Yet, the theory itself is subject to limitations, some of which I encountered myself, therefore calling for extension of the theoretical reasoning I conducted.

Through the notion of projection, motivated in chapter 3 and presented more in depth in chapter 5, I argue that any piece of code written in some computer language is not tied to a single representation by essence, and can be projected on any substrate as long as the appropriate resources and an appropriate mapping are provided. Yet, because of how prevalent textual representations have been in my own education and experience of computing and programming, the knowledge and habits I have are inevitably rooted in a view of the world in which text is the default. As such, the way I presented my theory of interaction with computer languages in this thesis is nuanced with a view that is, if not text-centric, at least text-first. For example, the four levels of interaction with computer languages I envision start from the one closest to the language's encoding, for which sequences of characters representing sequences of bytes are the most common representation.

Looking at this theory from the other way around yields a different approach to computer languages. By starting from the pragmatic level of interaction, in which computer languages are used in context, possibly very far and independently from how they are encoded, a surprisingly large set of user interfaces appear to become candidate environments for working with computer languages. If the vector-graphic editing panel of Sketch-n-Sketch (Hempel et al., 2019) can be considered as a substrate for interacting with the code that, when interpreted, yields the shapes shown on the screen and manipulated by the user, even if that user never writes code as text, why would software such as Microsoft Word, Apple Numbers or Adobe Photoshop not qualify as well?[80] Since graphemic and morphosyntactic interaction are only two out of four levels of interaction, code written in a computer language could therefore very well be edited using projections that are primarily designed to support semantic or pragmatic interaction, as do most traditional GUIs (such as those of the three systems cited in the last sentence). In this sense, the term *code* as I used it in this work appears to become interchangeable with *model* (as already hinted in subsection 5.3.1) or *data*: anything that is somehow encoded in a computer can qualify to be written in a computer language, even if it is never represented and edited as a piece of text or a graph of nodes—the two notations traditionally respectively associated with "textual" and "visual" languages in the literature.

80. The scripting capability of some of these pieces of software actually demonstrates that concepts first meant to be represented at a semantic or pragmatic level only can eventually become exposed as text and edited at the graphemic level, taking the opposite path to the one I explored.

Future work may explore what this conclusion means for the theoretical contributions presented in this thesis, which may be pushed beyond the mind-constraining notion of computer language to, perhaps, become a *linguistic theory of interaction* instead. This, interestingly, relates to one of the points made by Jakubovic et al. (2023) in their work on programming systems: seeing the activity of programming and the artefact of code as linguistic is a design choice we make—or, rather, someone else's decision we have to deal with—rather than a necessity. Without going as far as authors such as Chomsky, who argue that our human ability for language is hard-coded in our brains, I still hypothesise that the fact that we stick to language as an interaction metaphor, sixty year after the introduction of graphical user interfaces capable of displaying more than text, is not merely a coincidence or a technical or design debt.

In their work on generative theories of interactions, Beaudouin-Lafon et al. (2021) describe such theoretical constructs as follows:

> *We define a Generative Theory of Interaction as a construct that is: (1) grounded in a theory of human activity and behavior with technology; (2) amenable to analytical, critical and constructive interpretation; and (3) actionable through the theory's concepts and generative principles.*
>
> — Beaudouin-Lafon et al. (2021, §2.1)

In this thesis, I focused on demonstrating the second property of the theory I presented, by giving conceptual tools to decompose what interacting with a computer language means and applying them to analyse and categorise a variety of interactive systems. I only hinted at possible grounds for the two other properties by briefly reviewing work on computer languages in other disciplines (section 2.3) and by applying the concept of projection in some specific situations (chapters 6 and 7). Working towards a *generative linguistic theory of interaction* would require to expand on these two properties. As such, future work may help ground this theory in other theories of how and why humans deal with languages, helped with findings from fields such as psycholinguistics and cognitive science, as well as devise generative principles that can be used to design new interaction techniques and systems as I informally did myself, perhaps by suggesting to either start from graphemic or morphosyntactic interaction and go up towards semantic and pragmatic interaction, or the opposite, depending on the situation and the existing practices and software to design for.

9

Conclusion

In the beginning of this thesis, I made the following claim:

> *No computer language is inherently bound to a single representation, neither theoretically nor technically, and diversifying our interaction with computer code can help users understand and modify it.*

Eight chapters later, I now return to it with the knowledge I learnt by building up the theoretical, empirical and technical contributions I made in this thesis. I argue that both the theory of interaction with computer languages introduced in this work and the two concrete applications of protean projection I devised support this claim.

According to the theory of interaction with computer languages, every computer language must be projected onto a substrate for us to interact with it—otherwise, code remains data in the computer's memory, hidden from us. Since projecting a language onto one substrate rather than another is a purely arbitrary choice, there is no single nor best representation for a computer language, only habits and technical constraints—such as using text because it is very common in a Unix-centred world, easy to implement, fit for keyboards, and so on. Besides not being bound to a single representation, computer languages are not bound to a single level of interaction either: code can be edited at multiple levels in parallel, as demonstrated in many systems published in the literature, as well as those I developed myself, in which graphemic and semantic interaction can be used side by side. The theoretical claims made in this thesis are therefore strongly in favour of decoupling the concepts of a computer language, its formal specification, its encoding, and its context of use, from how we interact with it, which is nothing but a design choice we make when we develop and use a system for editing code written in this language.

This distinction is also supported by my work on *i*-LaTeX and LORGNETTE, two systems I created to complement text editors with additional projections. By offering transitional representations for certain fragments of code, *i*-LaTeX provides LaTeX users with alternative representations of mathematical formulae, tables, images and grid layouts, which can all be perceived and edited either as text or as another representation chosen to be more semantically meaningful and appropriate. They have shown benefits in two user studies by allowing participants to achieve a number of tasks faster and with a lower workload and by helping LaTeX users work on specific parts of their own documents in a non-controlled setting. Similarly, by letting users extend text editors with additional projections for specific patterns of code, LORGNETTE helps tailor one's editor to their needs and preferences beyond traditional end-user customisation practices. I demonstrated how LORGNETTE can be used in five different situations, all including projections of an established computer language that is otherwise edited as text only, such as JavaScript and Python. As such, *i*-LaTeX and LORGNETTE demonstrate that computer languages can not only benefit from multiple projections at the same time in practice, but that this approach has proven benefits and can be readily used with some of the most popular computer languages while remaining compatible with existing encodings, editors and workflows.

This claim has consequences on how we might use and interact with computer languages in the future. Moreover, the postmodern approach I followed in my applied work demonstrates that combining multiple representations and interaction levels can readily benefit computer languages that are being massively used today. Unlike other visions and prototypes found in the literature, this can be achieved without requiring any significant technological shift, as only slight adjustments to the most used text editors are required to let their users benefit from additional projections of the code they read and write.

In education, where computers occupy an increasing space, students may benefit from a multiplicity of projections when working with anything encoded as a computer language, similar to the benefits of observing the same piece of information from different perspectives when learning, and in line with the complementary uses of text, schemas, and even analog and digital artefacts by teachers. This obviously applies to teaching computer science and programming, which may benefit from going beyond morphosyntactic interaction; but it also concerns any discipline taught with computers, by encouraging developers of educational software to expose the languages of what they build so as to let students appropriate it at different conceptual levels, which may, in turn, help them better understand the underlying concepts.

In industry, which extensively develops and relies on software, developers may benefit from new ways of perceiving and transforming code, possibly increasing their efficiency while decreasing their workload; and users of said software may benefit from larger toolsets, possibly helping them become more expert and tailor their software to their needs or automate some of their work. In addition, exposing more and more mechanisms to support third-party projections of data manipulated by software—such as computer languages, but not only—might lead to the emergence of a new market, in which companies create and customise projections adapted to specific domains and personal preferences in an ecosystem ruled by common protocols. Eventually, everyone could *bring their own client* without risking to be isolated from others,[81] for "*one application is not enough*" and "*users should be able to easily select, fine-tune and appropriate their tools*" (Beaudouin-Lafon, 2017, §1).

81. The formula originates from related work from Geoffrey Litt (@151).

More generally, at the scale of society, I believe that thinking and talking about our interaction with computer languages in terms of projections may help making these languages, and, by extension, computing literacy, less obscure and frightening, and more accessible to all. My belief is akin to those who popularise mathematics and other scientific fields by showing what abstract concepts and cryptic formulae *mean*, beyond sequences of symbols we usually denote them with.[82] Given the role computers are taking in our societies, and how much education and regulation lag behind the fast pace of technological evolutions, I therefore see a democratic potential in this vision of computer languages, as a means to equip people with intellectual and technical tools to become actors of the computer-led world we now live in.

82. Popular examples include Nicky Case, the creator of a few popular explorable explanations (@138), and Grant Sanderson, the creator of the 3Blue1Brown (@172) website and YouTube channel.

Bibliography

Acar et al., 2013 DOI: `10.3233/JCS-130487`
Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2013. A Core Calculus for Provenance. *Journal of Computer Security* 21.6, pp. 919–969.

Aho et al., 2006 ISBN: `978-0-321-48681-3`
Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools*. 2nd edition. Addison-Wesley.

Ainsworth, 2020
Thomas Ainsworth. 2020. Form vs. Matter. *The Stanford Encyclopedia of Philosophy*. Summer 2020. Metaphysics Research Lab, Stanford University.

Almeida et al., 2022 DOI: `10.1109/VL/HCC53370.2022.9833121`
Marcelo Almeida, Grant Cole, Ke Du, Gongming Luo, Shulin Pan, Yu Pan, Kai Qiu, Vishnu Reddy, Haochen Zhang, Yingying Zhu, and Cyrus Omar. 2022. RustViz: Interactively Visualizing Ownership and Borrowing. *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10.

Alur et al., 2018 DOI: `10.1145/3208071`
Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Communications of the ACM* 61.12, pp. 84–93.

Andersen et al., 2020 DOI: `10.1145/3428290`
Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proceedings of the ACM on Programming Languages*. Vol. 4, pp. 1–28.

Andersen, 1992
Peter Andersen. 1992. Computer Semiotics. *Scandinavian Journal of Information Systems* 4.1.

Appel and Haken, 1977 DOI: `10.1038/scientificamerican1077-108`
Kenneth Appel and Wolfgang Haken. 1977. The Solution of the Four-Color-Map Problem. *Scientific American* 237.4, pp. 108–121.

Appert and Beaudouin-Lafon, 2006 DOI: `10.1145/1166253.1166302`
Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: Adding State Machines to the Swing Toolkit. *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*. UIST '06. ACM, pp. 319–322.

Arawjo, 2020 DOI: `10.1145/3313831.3376731`
Ian Arawjo. 2020. To Write Code: The Cultural Fabrication of Programming Notation and Practice. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. ACM, pp. 1–15.

Arawjo et al., 2022 DOI: 10.1145/3526113.3545619
Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan Parikh. 2022. Notational Programming for Notebook Environments: A Case Study with Quantum Circuits. *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST '22. ACM, pp. 1–20.

Asenov, 2017
Dimitar Asenov. 2017. Envision: Reinventing the Integrated Development Environment. Ph.D. thesis. ETH Zurich.

Asenov et al., 2016 DOI: 10.1145/2858036.2858372
Dimitar Asenov, Otmar Hilliges, and Peter Müller. 2016. The Effect of Richer Visualizations on Code Comprehension. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 5040–5045.

Atkinson and Shiffrin, 1968 DOI: 10.1016/S0079-7421(08)60422-3
Richard C. Atkinson and Richard M. Shiffrin. 1968. Human Memory: A Proposed System and Its Control Processes. *Psychology of Learning and Motivation*. Vol. 2. Academic Press, pp. 89–195.

Avrahami et al., 1989 DOI: 10.1145/74334.74347
Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. 1989. A Two-View Approach to Constructing User Interfaces. *ACM SIGGRAPH Computer Graphics* 23.3, pp. 137–146.

Backus, 1978 DOI: 10.1145/960118.808380
John Backus. 1978. The History of FORTRAN I, II, and III. *ACM SIGPLAN Notices* 13.8, pp. 165–180.

Baddeley and Hitch, 1974 DOI: 10.1016/S0079-7421(08)60452-1
Alan D. Baddeley and Graham Hitch. 1974. Working Memory. *Psychology of Learning and Motivation*. Vol. 8. Academic Press, pp. 47–89.

Banken et al., 2018 DOI: 10.1145/3180155.3180156
Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. *Proceedings of the 40th International Conference on Software Engineering*. ACM, pp. 752–763.

Bau et al., 2015 DOI: 10.1145/2771839.2771875
David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children - IDC '15*. ACM, pp. 445–448.

Bau et al., 2017 DOI: 10.1145/3015455
David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Communications of the ACM* 60.6, pp. 72–80.

Baudel, 1995
Thomas Baudel. 1995. Aspects Morphologiques de l'interaction Humain-Ordinateur : Étude de Modèles d'interaction Gestuels. Ph.D. thesis. Paris 11 University.

Beaudouin-Lafon, 2000 DOI: 10.1145/332040.332473
Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP

User Interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '00. ACM, pp. 446–453.

Beaudouin-Lafon, 2017 DOI: 10.1145/3125571.3125602
Michel Beaudouin-Lafon. 2017. Towards Unified Principles of Interaction. *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*. ACM, pp. 1–2.

Beaudouin-Lafon, 2023 DOI: 10.1145/3583961.3583968
Michel Beaudouin-Lafon. 2023. Au-delà des applications : Substrats et instruments d'interaction. *Proceedings of the 34th Conference on l'Interaction Humain-Machine*. IHM '23. ACM, pp. 1–15.

Beaudouin-Lafon et al., 2021 DOI: 10.1145/3468505
Michel Beaudouin-Lafon, Susanne Bødker, and Wendy E. Mackay. 2021. Generative Theories of Interaction. *ACM Transactions on Computer-Human Interaction* 28.6, pp. 1–54.

Beaudouin-Lafon and Mackay, 2000 DOI: 10.1145/345513.345267
Michel Beaudouin-Lafon and Wendy E. Mackay. 2000. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '00. ACM, pp. 102–109.

Beck et al., 2014 DOI: 10.1145/2591062.2591111
Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. 2014. RegViz: Visual Debugging of Regular Expressions. *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 504–507.

Beckmann et al., 2023a DOI: 10.1145/3544548.3580785
Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–16.

Beckmann et al., 2023b DOI: 10.1145/3623504.3623569
Tom Beckmann, Daniel Stachnik, Jens Lincke, and Robert Hirschfeld. 2023. Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. PAINT 2023. ACM, pp. 25–35.

Benbouzid, 2019 DOI: 10.1177/2053951719861703
Bilel Benbouzid. 2019. To Predict and to Manage. Predictive Policing in the United States. *Big Data & Society* 6.1, pp. 1–13.

Bergel et al., 2013
Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. *Deep into Pharo*. Square Bracket Associates.

Bergström and Blackwell, 2016 DOI: 10.1109/VLHCC.2016.7739684
Ilias Bergström and Alan F. Blackwell. 2016. The Practices of Programming. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 190–198.

Bertin, 1967 ISBN: 978-1-58948-261-6
Jacques Bertin. 1967. *Sémiologie Graphique : Les Diagrammes, Les Réseaux, Les Cartes*. Mouton; Gauthier-Villars.

Bertot and Castéran, 2004 ISBN: 978-3-540-20854-9
Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer.

Blackwell, 2018
Alan F. Blackwell. 2018. A Craft Practice of Programming Language Research. *Proceedings of the 29th Annual Workshop of the Psychology of Programming Interest Group*. PPIG 2018, pp. 1–9.

Blackwell et al., 2001 DOI: 10.1007/3-540-44617-6_31
Alan F. Blackwell, Carol Britton, Anna Cox, Thomas R. G. Green, Corin Gurr, Gada Kadoda, Maria S. Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Christopher Roast, Cristophe Roe, A. Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. *Cognitive Technology: Instruments of Mind*. Lecture Notes in Computer Science. Springer, pp. 325–341.

Blackwell et al., 2022 DOI: 10.7551/mitpress/13770.001.0001
Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual*. The MIT Press.

Blackwell et al., 2019 DOI: 10.1016/j.ijhcs.2019.06.009
Alan F. Blackwell, Marian Petre, and Luke Church. 2019. Fifty Years of the Psychology of Programming. *International Journal of Human-Computer Studies*. 50 Years of the International Journal of Human-Computer Studies. Reflections on the Past, Present and Future of Human-Centred Technologies 131, pp. 52–63.

Boey and Adams, 2022
Bernard Boey and Michael D. Adams. 2022. HenBlocks: Structured Editing for Coq. *FLoC2022: The 8th Federated Logic Conference*. The Coq Workshop 2022.

Borning, 1979
Alan Borning. 1979. *ThingLab—A Constraint-Oriented Simulation Laboratory*. Tech. rep. SSL-79-3. Xerox PARC.

Borowski et al., 2022 DOI: 10.1145/3491102.3502064
Marcel Borowski, Luke Murray, Rolf Bagge, Janus B. Kristensen, Arvind Satyanarayan, and Clemens Nylands N. Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. *CHI Conference on Human Factors in Computing Systems*. CHI '22. ACM, pp. 1–20.

Bower and McIver, 2011 DOI: 10.1145/1999747.1999809
Matt Bower and Annabelle McIver. 2011. Continual and Explicit Comparison to Promote Proactive Facilitation during Second Computer Language Learning. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. ACM, pp. 218–222.

Bragdon et al., 2010 DOI: 10.1145/1753326.1753706
Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, pp. 2503–2512.

Braun and Clarke, 2019 DOI: `10.1080/2159676X.2019.1628806`
Virginia Braun and Victoria Clarke. 2019. Reflecting on Reflexive Thematic Analysis. *Qualitative Research in Sport, Exercise and Health* 11.4, pp. 589–597.

Breckel and Tichy, 2016
Alexander Breckel and Matthias Tichy. 2016. Live Programming with Code Portals. *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP'16)*, pp. 1–9.

Brooks, 1977 DOI: `10.1016/S0020-7373(77)80039-4`
Ruven Brooks. 1977. Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Man-Machine Studies* 9.6, pp. 737–751.

Bubeck et al., 2023 DOI: `10.48550/arXiv.2303.12712`
Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. *Sparks of Artificial General Intelligence: Early Experiments with GPT-4*.

Burks, 1947 DOI: `10.1109/JRPROC.1947.234265`
Alan W. Burks. 1947. Electronic Computing Circuits of the ENIAC. *Proceedings of the IRE* 35.8, pp. 756–767.

Cascaval et al., 2022 DOI: `10.1111/cgf.14476`
Dab Cascaval, Mira Shalah, Philip Quinn, Rastislav Bodik, Maneesh Agrawala, and Adriana Schulz. 2022. Differentiable 3D CAD Programs for Bidirectional Editing. *Computer Graphics Forum* 41.2, pp. 309–323.

Chang and Myers, 2014 DOI: `10.1145/2642918.2647371`
Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. UIST '14. ACM, pp. 87–96.

Chasins et al., 2021 DOI: `10.1145/3469279`
Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: Better Together. *Communications of the ACM* 64.8, pp. 98–106.

Chen et al., 2021 DOI: `10.48550/arXiv.2107.03374`
Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. *Evaluating Large Language Models Trained on Code*.

Cheney et al., 2009 DOI: `10.1561/1900000006`
James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1.4, pp. 379–474.

Cherubini et al., 2007 DOI: `10.1145/1240624.1240714`
Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. LaTeX2Solver: a Hierarchical Semantic Parsing of LaTeX Document into Code for an Assistive Optimization Modeling Application, pp. 557–566.

Chomsky, 1959 DOI: `10.1016/S0019-9958(59)90362-6`
Noam Chomsky. 1959. On Certain Formal Properties of Grammars. *Information and Control* 2.2, pp. 137–167.

Chugh, 2016 DOI: `10.1145/2889160.2889210`
Ravi Chugh. 2016. Prodirect Manipulation: Bidirectional Programming for the Masses. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. ACM, pp. 781–784.

Chugh et al., 2016 DOI: `10.1145/2980983.2908103`
Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. *ACM SIGPLAN Notices* 51.6, pp. 341–354.

Churchill et al., 2019 DOI: `10.48550/arXiv.1904.09828`
Alex Churchill, Stella Biderman, and Austin Herrick. 2019. *Magic: The Gathering Is Turing Complete.*

Clem and Thomson, 2022 DOI: `10.1145/3486594`
Timothy Clem and Patrick Thomson. 2022. Static Analysis at GitHub. *Communications of the ACM* 65.2, pp. 44–51.

Coblenz et al., 2021 DOI: `10.1145/3452379`
Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process That Integrates User-Centered Methods into Programming Language Design. *ACM Transactions on Computer-Human Interaction* 28.4, 28:1–28:53.

Coblenz et al., 2022 DOI: `10.1145/3510003.3510107`
Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. ACM, pp. 1021–1032.

Conlen and Heer, 2018 DOI: `10.1145/3242587.3242600`
Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST '18. ACM, pp. 977–989.

Conlen et al., 2021 DOI: `10.1145/3472749.3474731`
Matthew Conlen, Megan Vo, Alan Tan, and Jeffrey Heer. 2021. Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles. *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. ACM, pp. 1–12.

Conner, 1984
Richard L. Conner. 1984. Cobol, Your Age Is Showing. *Computerworld* 18.20, pp. ID/7–ID/18.

Connolly and Cooke, 2004 DOI: `10.1515/semi.2004.065`
John H. Connolly and D. John Cooke. 2004. The Pragmatics of Programming Languages. *Semiotica* 2004.151, pp. 149–161.

Conversy, 2014 DOI: `10.1145/2661136.2661138`
Stéphane Conversy. 2014. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. ACM, pp. 201–212.

Cooper, 2010 DOI: `10.1145/1868358.1868362`
Stephen Cooper. 2010. The Design of Alice. *ACM Transactions on Computing Education* 10.4, pp. 1–16.

Copeland, 2023
B. Jack Copeland. 2023. The Church-Turing Thesis. *The Stanford Encyclopedia of Philosophy*. Winter 2023. Metaphysics Research Lab, Stanford University.

Cordy, 2006 DOI: `10.1016/j.scico.2006.04.002`
James R. Cordy. 2006. The TXL Source Transformation Language. *Science of Computer Programming*. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04) 61.3, pp. 190–210.

Dale, 2021 DOI: `10.1017/S1351324920000601`
Robert Dale. 2021. GPT-3: What's It Good for? *Natural Language Engineering* 27.1, pp. 113–118.

de Moura et al., 2015 DOI: `10.1007/978-3-319-21401-6_26`
Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). *Automated Deduction - CADE-25*. Lecture Notes in Computer Science. Springer International Publishing, pp. 378–388.

de Souza, 1993 DOI: `10.1006/imms.1993.1082`
Clarisse S. de Souza. 1993. The Semiotic Engineering of User Interface Languages. *International Journal of Man-Machine Studies* 39.5, pp. 753–773.

DeFanti, 1980
Thomas DeFanti. 1980. Language Control Structures for Easy Electronic Visualization. *Byte* 5.11, pp. 90–104.

DeLine et al., 2012 DOI: `10.1109/ICSE.2012.6227113`
Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. 2012. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm. *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 1064–1073.

DeLine et al., 2006 DOI: `10.1109/VLHCC.2006.14`
Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. 2006. Code Thumbnails: Using Spatial Memory to Navigate Source Code. *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pp. 11–18.

DeLine, 2021 DOI: `10.1145/3411764.3445267`
Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-Specific Language. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1–11.

Descheemaeker et al., 2021 DOI: 10.1145/3486605.3486785
Cloé Descheemaeker, Sam Van den Vonder, Thierry Renaux, and Wolfgang De Meuter. 2021. Poker: Visual Instrumentation of Reactive Programs with Programmable Probes. *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. ACM, pp. 14–26.

Dietrich, 1986 DOI: 10.2307/1578284
Frank Dietrich. 1986. Visual Intelligence: The First Decade of Computer Art (1965–1975). *Leonardo* 19.2, pp. 159–169.

Dijkstra, 1968 DOI: 10.1145/362929.362947
Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Communications of the ACM* 11.3, pp. 147–148.

Dijkstra, 1977
Edsger W. Dijkstra. 1977. Programming : From Craft to Scientific Discipline. *Proceedings of the 5th International Computing Symposium*, pp. 23–30.

Dijkstra, 1982 DOI: 10.1145/947923.947924
Edsger W. Dijkstra. 1982. How Do We Tell Truths That Might Hurt? *ACM SIGPLAN Notices* 17.5, pp. 13–15.

diSessa and Abelson, 1986 DOI: 10.1145/6592.6595
Andrea A. diSessa and Harold Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Communications of the ACM* 29.9, pp. 859–868.

Dix, 2007
Alan Dix. 2007. Designing for Appropriation. *Proceedings of HCI 2007 The 21st British HCI Group Annual Conference University of Lancaster*. 21, pp. 1–4.

Donzeau-Gouge et al., 1980
Véronique Donzeau-Gouge, Gérard Huet, Bernard Lang, and Gilles Kahn. 1980. *Programming Environments Based on Structured Editors : The Mentor Experience*. Research Report RR-0026. Inria.

Dragicevic et al., 2011 DOI: 10.1145/2047196.2047229
Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from Markup Code to Rendered Documents and Vice Versa. *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. ACM, pp. 257–262.

Dragicevic et al., 2019 DOI: 10.1145/3290605.3300295
Pierre Dragicevic, Yvonne Jansen, Abhraneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the Transparency of Research Papers with Explorable Multiverse Analyses. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. ACM, pp. 1–15.

Dreher, 2020 ISBN: 978-1-71685-581-8
Thomas Dreher. 2020. *History of Computer Art*. Lulu.

Drosos et al., 2020 DOI: 10.1145/3313831.3376442
Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-

by-Example Interaction for Synthesizing Readable Code for Data Scientists. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. ACM, pp. 1–12.

Edwards, 2005 DOI: `10.1145/1094811.1094851`
Jonathan Edwards. 2005. Subtext: Uncovering the Simplicity of Programming. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. ACM, pp. 505–518.

Ehtesham-Ul-Haque et al., 2022 DOI: `10.1145/3526113.3545620`
Md Ehtesham-Ul-Haque, Syed Mostofa Monsur, and Syed Masum Billah. 2022. Grid-Coding: An Accessible, Efficient, and Structured Coding Paradigm for Blind and Low-Vision Programmers. *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST '22. ACM, pp. 1–21.

Eiselmayer et al., 2019 DOI: `10.1145/3290605.3300447`
Alexander Eiselmayer, Chat Wacharamanotham, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2019. Touchstone2: An Interactive Environment for Exploring Trade-Offs in HCI Experiment Design. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. ACM, pp. 1–11.

Eisenberg and Kiczales, 2007 DOI: `10.1145/1218563.1218573`
Andrew D. Eisenberg and Gregor Kiczales. 2007. Expressive Programs through Presentation Extension. *Proceedings of the 6th International Conference on Aspect-Oriented Software Development - AOSD '07*. ACM, p. 73.

Ellis et al., 2018
Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc.

Engelbart, 1962
Douglas C. Engelbart. 1962. *Augmenting Human Intellect: A Conceptual Framework*. Tech. rep. Stanford Research Institute.

Engelbart and English, 1968 DOI: `10.1145/1476589.1476645`
Douglas C. Engelbart and William K. English. 1968. A Research Center for Augmenting Human Intellect. *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, Part I). ACM, pp. 395–410.

Ens et al., 2017
Barrett Ens, Fraser Anderson, Tovi Grossman, Michelle Annett, Pourang Irani, and George Fitzmaurice. 2017. Ivy: Exploring Spatially Situated Visual Programming for Authoring and Understanding Intelligent Environments. *Proceedings of the 43rd Graphics Interface Conference*. GI '17. Canadian Human-Computer Communications Society, pp. 156–162.

Ensmenger, 2010 ISBN: `978-0-262-05093-7`
Nathan Ensmenger. 2010. *The Computer Boys Take over: Computers, Programmers, and the Politics of Technical Expertise*. History of Computing. MIT Press.

Erdweg and Ostermann, 2011 DOI: `10.1007/978-3-642-19440-5_26`
Sebastian T. Erdweg and Klaus Ostermann. 2011. Featherweight TeX and Parser Correctness. *Software Language Engineering*. Springer, pp. 397–416.

Erwig and Meyer, 1995 DOI: `10.1109/VL.1995.520825`
Martin Erwig and Bernd Meyer. 1995. Heterogeneous Visual Languages-Integrating Visual and Textual Programming. *1995 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Comput. Soc. Press, pp. 318–325.

Esser et al., 2021 DOI: `10.1109/CVPR46437.2021.01268`
Patrick Esser, Robin Rombach, and Björn Ommer. 2021. Taming Transformers for High-Resolution Image Synthesis. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 12868–12878.

Ferdowsifard et al., 2021 DOI: `10.1145/3485530`
Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages* 5.OOPSLA, 153:1–153:29.

Ferdowsifard et al., 2020 DOI: `10.1145/3379337.3415869`
Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 614–626.

Fix and Wiedenbeck, 1996 DOI: `10.1016/0360-1315(96)00022-X`
Vikki Fix and Susan Wiedenbeck. 1996. An Intelligent Tool to Aid Students in Learning Second and Subsequent Programming Languages. *Computers & Education* 27.2, pp. 71–83.

Flener, 2002 DOI: `10.1007/3-540-45628-7_13`
Pierre Flener. 2002. Achievements and Prospects of Program Synthesis. *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I*. Lecture Notes in Computer Science. Springer, pp. 310–346.

Fog and Klokmose, 2019
Bjarke V. Fog and Clemens N. Klokmose. 2019. Mapping the Landscape of Literate Computing. *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group*. PPIG 2019, pp. 1–10.

Forment and Armitage, 2023 DOI: `10.5281/zenodo.7843817`
Raphaël Maurice Forment and Jack Armitage. 2023. Sardine: A Modular Python Live Coding Environment. *Proceedings of the 7th International Conference on Live Coding*. ICLC 2023, pp. 1–12.

Foster et al., 2007 DOI: `10.1145/1232420.1232424`
Nate (Nathan) Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems* 29.3, pp. 1–65.

Fowler, 2019 ISBN: `978-0-13-475759-9`
Martin Fowler. 2019. *Refactoring: Improving the Design of Existing Code*. 2nd edition. Addison-Wesley.

Françoise et al., 2022 DOI: 10.1145/3491102.3501916
Jules Françoise, Sarah Fdili Alaoui, and Yves Candau. 2022. CO/DA: Live-Coding Movement-Sound Interactions for Dance Improvisation. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. ACM, pp. 1–13.

French, 2000 DOI: 10.1016/S1364-6613(00)01453-4
Robert M. French. 2000. The Turing Test: The First 50 Years. *Trends in Cognitive Sciences* 4.3, pp. 115–122.

Fuggetta, 1993 DOI: 10.1109/2.247645
Alfonso Fuggetta. 1993. A Classification of CASE Technology. *Computer* 26.12, pp. 25–38.

Fulton et al., 2021
Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. *17th Symposium on Usable Privacy and Security (SOUPS 2021)*, pp. 597–616.

Gamma et al., 1995 ISBN: 978-0-201-63361-0
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Ge et al., 2012 DOI: 10.1109/ICSE.2012.6227192
Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. *2012 34th International Conference on Software Engineering (ICSE)*, pp. 211–221.

Gil and Maman, 2005 DOI: 10.1145/1094811.1094819
Joseph (Yossi) Gil and Itay Maman. 2005. Micro Patterns in Java Code. *Proceedings of the 20th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA '05*. ACM, pp. 97–116.

Gil et al., 2019 DOI: 10.1016/j.cola.2019.100905
Joseph (Yossi) Gil, Ori Marcovitch, and Matteo Orrú. 2019. A Nano-Pattern Language for Java. *Journal of Computer Languages* 54, pp. 1–25.

Gilsing et al., 2022 DOI: 10.1016/j.cola.2022.101158
Marleen Gilsing, Jesús Pelay, and Felienne Hermans. 2022. Design, Implementation and Evaluation of the Hedy Programming Language. *Journal of Computer Languages* 73, pp. 1–17.

Glassman et al., 2015 DOI: 10.1145/2699751
Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction* 22.2, 7:1–7:35.

Gobert and Beaudouin-Lafon, 2021 DOI: 10.1145/3450522.3451325
Camille Gobert and Michel Beaudouin-Lafon. 2021. Représentations Intermédiaires Interactives Pour La Manipulation de Code LaTeX. *32e Conférence Francophone Sur l'Interaction Homme-Machine*. IHM '21. ACM, pp. 1–11.

Gobert and Beaudouin-Lafon, 2022 DOI: 10.1145/3491102.3517494
Camille Gobert and Michel Beaudouin-Lafon. 2022. I-LaTeX: Manipulating Transitional Representations between

LaTeX Code and Generated Documents. *CHI Conference on Human Factors in Computing Systems*. CHI ’22. ACM, pp. 1–16.

Gobert and Beaudouin-Lafon, 2023 DOI: `10.1145/3586183.3606817`
Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST ’23. ACM, pp. 1–16.

Goldberg and Robson, 1983 ISBN: `978-0-201-11371-6`
Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

Goldenson et al., 1992 DOI: `10.1145/142386.1055544`
Dennis R. Goldenson, Ravinder P. Chandhok, David H. Garlan, Glenn Meter, Philip L. Miller, John Pane, Jacobo Carrasquel, James A. Roberts, and Edward J. Skwarecki. 1992. GENIE: Developing and Assessing State-of-the-Art Integrated Programming Environments. *ACM SIGCHI Bulletin* 24.2, pp. 39–40.

Gonthier, 2008
Georges Gonthier. 2008. Formal Proof—The Four-Color Theorem. *Notices of the AMS* 55.11, pp. 1382–1393.

Gonzalez et al., 2023 DOI: `10.1145/3607822.3614521`
Johann Felipe Gonzalez, Danny Kieken, Thomas Pietrzak, Audrey Girouard, and Géry Casiez. 2023. Introducing Bidirectional Programming in Constructive Solid Geometry-Based CAD. *Proceedings of the 2023 ACM Symposium on Spatial User Interaction*. SUI ’23. ACM, pp. 1–12.

Green, 1989
Thomas R. G. Green. 1989. Cognitive Dimensions of Notations. *People and Computers V*, pp. 443–460.

Green and Petre, 1996 DOI: `10.1006/jvlc.1996.0009`
Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages & Computing* 7.2, pp. 131–174.

Grønbæk et al., 2023 DOI: `10.1145/3586183.3606767`
Jens Emil S. Grønbæk, Marcel Borowski, Eve Hoggan, Wendy E. Mackay, Michel Beaudouin-Lafon, and Clemens N. Klokmose. 2023. Mirrorverse: Live Tailoring of Video Conferencing Interfaces. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST ’23. ACM, pp. 1–14.

Grossman and Balakrishnan, 2005 DOI: `10.1145/1054972.1055012`
Tovi Grossman and Ravin Balakrishnan. 2005. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor’s Activation Area. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’05. ACM, pp. 281–290.

Gulwani, 2011 DOI: `10.1145/1925844.1926423`
Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *ACM SIGPLAN Notices* 46.1, pp. 317–330.

Guo, 2013 DOI: `10.1145/2445196.2445368`
Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE ’13. ACM, pp. 579–584.

Guo, 2018 DOI: 10.1145/3173574.3173970
Philip J. Guo. 2018. Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. ACM, pp. 1–14.

Han et al., 2020 DOI: 10.1145/3313831.3376804
Han L. Han, Miguel A. Renom, Wendy E. Mackay, and Michel Beaudouin-Lafon. 2020. Textlets: Supporting Constraints and Consistency in Text Documents. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. ACM, pp. 1–13.

Hanna, 2002 DOI: 10.1145/581478.581493
Keith Hanna. 2002. Interactive Visual Functional Programming. *Proceedings of the 7th International Conference on Functional Programming - ICFP '02*. ACM, pp. 145–156.

Hansen, 1972 DOI: 10.1145/1479064.1479159
Wilfred J. Hansen. 1972. User Engineering Principles for Interactive Systems. *Proceedings of the November 16-18, 1971, Fall Joint Computer Conference*. AFIPS '71 (Fall). ACM, pp. 523–532.

Hao and Glassman, 2020 DOI: 10.4230/OASIcs.PLATEAU.2019.1
Rebecca L. Hao and Elena L. Glassman. 2020. Approaching Polyglot Programming: What Can We Learn from Bilingualism Studies? *10th Workshop on Evaluation and Usability of Programming Languages and Tools*. Vol. 76. PLATEAU 2019, pp. 1–7.

Hart, 2006 DOI: 10.1177/154193120605000909
Sandra G. Hart. 2006. Nasa-Task Load Index (NASA-TLX); 20 Years Later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50.9, pp. 904–908.

Hartmann et al., 2008 DOI: 10.1145/1449715.1449732
Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*. UIST '08. ACM, pp. 91–100.

Hayatpur et al., 2023 DOI: 10.1145/3544548.3581390
Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. 2023. CrossCode: Multi-Level Visualization of Program Execution. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–13.

Head et al., 2021 DOI: 10.1145/3411764.3445648
Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S. Weld, and Marti A. Hearst. 2021. Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1–18.

Hempel and Chugh, 2020 DOI: 10.1109/VL/HCC50065.2020.9127256
Brian Hempel and Ravi Chugh. 2020. Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions). *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 1–5.

Hempel and Chugh, 2022 DOI: 10.4230/LIPIcs.ECOOP.2022.16
Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Vol. 222, 16:1–16:29.

Hempel et al., 2019 DOI: 10.1145/3332165.3347925
Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. ACM, pp. 281–292.

Hempel et al., 2018 DOI: 10.1145/3180155.3180165
Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM, pp. 654–664.

Hermans, 2020 DOI: 10.1145/3372782.3406262
Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ICER '20. ACM, pp. 259–270.

Hoffswell et al., 2018 DOI: 10.1145/3173574.3174106
Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. ACM, pp. 1–12.

Homer and Noble, 2017 DOI: 10.18293/VLSS2017-007
Michael Homer and James Noble. 2017. Lessons in Combining Block-Based and Textual Programming. *Journal of Visual Languages and Sentient Systems* 3.1, pp. 22–39.

Horowitz and Heer, 2023 DOI: 10.48550/arXiv.2303.06777
Joshua Horowitz and Jeffrey Heer. 2023. Live, Rich, and Composable: Qualities for Programming Beyond Static Text. *13th Annual Workshop at the Intersection of PL and HCI*. PLATEAU 2023. arXiv.

Hui and Kromberg, 2020 DOI: 10.1145/3386319
Roger K. W. Hui and Morten J. Kromberg. 2020. APL since 1978. *Proceedings of the ACM on Programming Languages* 4.HOPL, 69:1–69:108.

Ivanova et al., 2020 DOI: 10.7554/eLife.58906
Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. 2020. Comprehension of Computer Code Relies Primarily on Domain-General Executive Brain Regions. *eLife* 9, pp. 1–24.

Jabi, 2013 ISBN: 978-1-78067-314-1
Wassim Jabi. 2013. *Parametric Design for Architecture*. Laurence King Publishing.

Jablonski and Hou, 2007 DOI: 10.1145/1328279.1328283
Patricia Jablonski and Daqing Hou. 2007. CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE. *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse'07*. ACM, pp. 16–20.

Jacobs et al., 2018 DOI: `10.1145/3173574.3174164`
Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. ACM, pp. 1–13.

Jakobson, 1960
Roman Jakobson. 1960. Linguistics and Poetics. *Style in Language*. MIT Press, pp. 350–377.

Jakubovic et al., 2023 DOI: `10.22152/programming-journal.org/2023/7/13`
Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7.3, pp. 1–59.

Jiang et al., 2023 DOI: `10.1145/3544548.3581403`
Peiling Jiang, Fuling Sun, and Haijun Xia. 2023. Log-It: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–16.

Joharizadeh et al., 2020 DOI: `10.1145/3334480.3382806`
Nima Joharizadeh, Advait Sarkar, Andrew D. Gordon, and Jack Williams. 2020. Gridlets: Reusing Spreadsheet Grids. *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA '20. ACM, pp. 1–7.

Johnson and Bui, 2015 DOI: `10.1109/BLOCKS.2015.7369007`
Chris Johnson and Peter Bui. 2015. Blocks in, Blocks out: A Language for 3D Models. *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pp. 77–82.

Johnson, 2020 DOI: `10.1080/03071847.2020.1752026`
James Johnson. 2020. Artificial Intelligence, Drone Swarming and Escalation Risks in Future Warfare. *The RUSI Journal* 165.2, pp. 26–36.

Kang and Guo, 2017 DOI: `10.1145/3126594.3126632`
Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. ACM, pp. 737–745.

Kell, 2013 DOI: `10.1145/2525528.2525534`
Stephen Kell. 2013. The Operating System: Should There Be One? *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*. PLOS '13. ACM, pp. 1–7.

Kery et al., 2020 DOI: `10.1145/3379337.3415842`
Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 140–151.

Klokmose et al., 2015 DOI: `10.1145/2807442.2807446`
Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. ACM, pp. 280–290.

Klokmose et al., 2019 DOI: `10.1145/3332165.3347912`
Clemens N. Klokmose, Christian Remy, Janus B. Kristensen, Rolf Bagge, Michel Beaudouin-Lafon, and Wendy Mackay. 2019. Videostrates: Collaborative, Distributed and Programmable Video Manipulation. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. ACM, pp. 233–247.

Knauff and Nejasmic, 2014 DOI: `10.1371/journal.pone.0115069`
Markus Knauff and Jelica Nejasmic. 2014. An Efficiency Comparison of Document Preparation Systems Used in Academic Research and Development. *PLOS ONE* 9.12, pp. 1–12.

Knuth, 1974 DOI: `10.1145/361604.361612`
Donald E. Knuth. 1974. Computer Programming as an Art. *Communications of the ACM* 17.12, pp. 667–673.

Knuth, 1984a DOI: `10.1093/comjnl/27.2.97`
Donald E. Knuth. 1984. Literate Programming. *The Computer Journal* 27.2, pp. 97–111.

Knuth, 1984b ISBN: `978-0-201-13448-3`
Donald E. Knuth. 1984. *The TeXbook*. Computers & Typesetting. Addison-Wesley.

Ko, 2016 DOI: `10.1145/3001878.3001880`
Amy J. Ko. 2016. What Is a Programming Language, Really? *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2016. ACM, pp. 32–33.

Ko and Myers, 2006 DOI: `10.1145/1124772.1124831`
Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. ACM, pp. 387–396.

Ko and Myers, 2008 DOI: `10.1145/1368088.1368130`
Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. ACM, pp. 301–310.

Kodosky, 2020 DOI: `10.1145/3386328`
Jeffrey Kodosky. 2020. LabVIEW. *Proceedings of the ACM on Programming Languages* 4.HOPL, 78:1–78:54.

Kolata, 1982 DOI: `10.1126/science.217.4566.1237`
Gina Kolata. 1982. How Can Computers Get Common Sense? *Science* 217.4566, pp. 1237–1238.

Kölling, 2010 DOI: `10.1145/1868358.1868361`
Michael Kölling. 2010. The Greenfoot Programming Environment. *ACM Transactions on Computing Education* 10.4, 14:1–14:21.

Kölling et al., 2017 DOI: `10.18293/VLSS2017-009`
Michael Kölling, Neil Brown, and Amjad Altadmri. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems* 3.1, pp. 40–67.

Kuhn, 1989
D. L. Kuhn. 1989. Selecting and Effectively Using a Computer Aided Software Engineering Tool. *Annual Westinghouse Computer Symposium*, pp. 1–13.

Kurtz, 1978 DOI: `10.1145/800025.1198404`
Thomas E. Kurtz. 1978. BASIC. *History of Programming Languages*. ACM, pp. 515–537.

Lafontant, 2022
Louis-Edouard Lafontant. 2022. Gentleman: A Lightweight Web-Based Projectional Editor. M.Sc. Thesis. Montreal University.

Lamport, 1994 ISBN: `978-0-201-52983-8`
Leslie Lamport. 1994. *LaTeX: A Document Preparation System: User's Guide and Reference Manual*. Addison-Wesley.

LaToza and Myers, 2011 DOI: `10.1109/VLHCC.2011.6070388`
Thomas D. LaToza and Brad A. Myers. 2011. Visualizing Call Graphs. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 117–124.

Laurens, 2007
Jérôme Laurens. 2007. Will TeX Ever Be WYSIWYG or the PDF Synchronization Story. *The PracTeX Journal* 3.3, pp. 1–8.

Laurens, 2008
Jérôme Laurens. 2008. Direct and Reverse Synchronization with SyncTeX. *TUGBoat* 29.3, pp. 365–371.

LeCun et al., 1989 DOI: `10.1162/neco.1989.1.4.541`
Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1.4, pp. 541–551.

Lerner, 2020a DOI: `10.1145/3379337.3415834`
Sorin Lerner. 2020. Focused Live Programming with Loop Seeds. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 607–613.

Lerner, 2020b DOI: `10.1145/3313831.3376494`
Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. ACM, pp. 1–7.

Leroy, 2020
Xavier Leroy. 2020. *Software, between Mind and Matter*. Inaugural Lecture at Collège de France. Collège de France.

Leroy et al., 2016
Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert — A Formally Verified Optimizing Compiler. *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, pp. 1–8.

Li et al., 2021 DOI: `10.1145/3411764.3445682`
Jingyi Li, Sonia Hashim, and Jennifer Jacobs. 2021. What We Can Learn From Visual Artists About Software Development. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. ACM, pp. 1–14.

Li et al., 2022 DOI: `10.1145/3550469.3555395`
Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2022. HeartDown: Document Processor for Executable Linear Algebra Papers. *SIGGRAPH Asia 2022 Conference Papers*. SA '22. ACM, pp. 1–8.

Lieber et al., 2014 DOI: `10.1145/2556288.2557409`
Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. ACM, pp. 2481–2490.

Lieberman, 2001 ISBN: `978-1-55860-688-3`
Henry Lieberman. 2001. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann Publishers.

Lilis and Savidis, 2019 DOI: `10.1145/3354584`
Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Computing Surveys* 52.6, 113:1–113:39.

Lin et al., 2021 DOI: `10.1145/3472749.3474804`
Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. ACM, pp. 1039–1049.

Litt and Jackson, 2020 DOI: `10.1145/3397537.3397541`
Geoffrey Litt and Daniel Jackson. 2020. Wildcard: Spreadsheet-Driven Customization of Web Applications. *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*. Programming '20. ACM, pp. 126–135.

Liu et al., 2020 DOI: `10.7554/eLife.59340`
Yun-Fei Liu, Judy Kim, Colin Wilson, and Marina Bedny. 2020. Computer Code Comprehension Shares Neural Resources with Formal Logical Inference in the Fronto-Parietal Network. *eLife* 9, pp. 1–22.

Long and Rinard, 2016 DOI: `10.1145/2837614.2837617`
Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, pp. 298–312.

Mangano et al., 2015 DOI: `10.1109/TSE.2014.2362924`
Nicolas Mangano, Thomas D. LaToza, Marian Petre, and Andre van der Hoek. 2015. How Software Designers Interact with Sketches at the Whiteboard. *IEEE Transactions on Software Engineering* 41.2, pp. 135–156.

Martin et al., 2022 DOI: `10.1145/3524610.3527885`
Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. 2022. Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor. *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC '22. ACM, pp. 241–252.

Martin et al., 2023 DOI: `10.1007/s11023-023-09624-2`
Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. 2023. Computers as Interactive Machines: Can We Build an Explanatory Abstraction? *Minds and Machines* 33.1, pp. 83–112.

Masson et al., 2023a DOI: 10.1145/3544548.3581091
Damien Masson, Sylvain Malacria, Géry Casiez, and Daniel Vogel. 2023. Charagraph: Interactive Generation of Charts for Realtime Annotation of Data-Rich Paragraphs. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–18.

Masson et al., 2023b DOI: 10.1145/3586183.3606762
Damien Masson, Sylvain Malacria, Géry Casiez, and Daniel Vogel. 2023. Statslator: Interactive Translation of NHST and Estimation Statistics Reporting Styles in Scientific Documents. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST '23. ACM, pp. 1–14.

Masson et al., 2020 DOI: 10.1145/3313831.3376559
Damien Masson, Sylvain Malacria, Edward Lank, and Géry Casiez. 2020. Chameleon: Bringing Interactivity to Static Digital Documents. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. ACM, pp. 1–13.

Mathur et al., 2020 DOI: 10.1111/cgf.14046
Aman Mathur, Marcus Pirron, and Damien Zufferey. 2020. Interactive Programming for Parametric CAD. *Computer Graphics Forum* 39.6, pp. 408–425.

Mayer et al., 2018 DOI: 10.1145/3276497
Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages* 2.OOPSLA, pp. 1–28.

McCarthy et al., 1967 DOI: 10.1145/1465482.1465582
John McCarthy, Dow Brian, Gary Feldman, and John Allen. 1967. THOR: A Display Based Time Sharing System. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). ACM, pp. 623–633.

McCarthy and Silver, 1960
John McCarthy and Roland Silver. 1960. *Colossal Typewriter Program*. Memorandum to PDP-1 Users. MIT.

McCartney, 2002 DOI: 10.1162/014892602320991383
James McCartney. 2002. Rethinking the Computer Music Language: Super Collider. *Computer Music Journal* 26.4, pp. 61–68.

McGrenere, 1998
Joanna McGrenere. 1998. *Learning to Use Complex Computer Technology: The Importance of User Interface Design*. CSRG Technical Report 403. Department of Computer Science, University of Toronto, pp. 1–67.

McGuffin and Fuhrman, 2020 DOI: 10.1145/3399715.3399821
Michael J. McGuffin and Christopher P. Fuhrman. 2020. Categories and Completeness of Visual Programming and Direct Manipulation. *Proceedings of the ACM International Conference on Advanced Visual Interfaces - AVI'20*. ACM, pp. 1–8.

McLean, 2011
Alex McLean. 2011. Artist-Programmers and Programming Languages for the Arts. Ph.D. thesis. Department of Computing, Goldsmiths, University of London.

McLean, 2014 DOI: `10.1145/2633638.2633647`
Alex McLean. 2014. Making Programming Languages to Dance to: Live Coding with Tidal. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design - FARM '14*. ACM Press, pp. 63–70.

McLean and Wiggins, 2010
Alex McLean and Geraint Wiggins. 2010. Tidal — Pattern Language for the Live Coding of Music. *Proceedings of the 7th Sound and Music Computing Conference*, pp. 264–269.

McNutt and Chugh, 2021 DOI: `10.1145/3411764.3445356`
Andrew McNutt and Ravi Chugh. 2021. Integrated Visualization Editing via Parameterized Declarative Templates. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. ACM, pp. 1–14.

McNutt and Chugh, 2023 DOI: `10.1109/VL-HCC57772.2023.00015`
Andrew McNutt and Ravi Chugh. 2023. Projectional Editors for JSON-Based DSLs. *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 60–70.

Mcnutt et al., 2023 DOI: `10.1145/3544548.3580683`
Andrew Mcnutt, Anton Outkine, and Ravi Chugh. 2023. A Study of Editor Features in a Creative Coding Classroom. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–15.

Mealin and Murphy-Hill, 2012 DOI: `10.1109/VLHCC.2012.6344485`
Sean Mealin and Emerson Murphy-Hill. 2012. An Exploratory Study of Blind Software Developers. *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 71–74.

Mellor and Balcer, 2002 ISBN: `978-0-201-74804-8`
Stephen J. Mellor and Marc J. Balcer. 2002. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley.

Memmott, 2011 DOI: `10.1386/jwcp.4.1.93_1`
Talan Memmott. 2011. Codework: Phenomenology of an anti-genre. *Journal of Writing in Creative Practice* 4.1, pp. 93–105.

Merigoux et al., 2021 DOI: `10.1145/3473582`
Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021. Catala: A Programming Language for the Law. *Proceedings of the ACM on Programming Languages* 5.ICFP, 77:1–77:29.

Miara et al., 1983 DOI: `10.1145/182.358437`
Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. 1983. Program Indentation and Comprehensibility. *Communications of the ACM* 26.11, pp. 861–867.

Michael et al., 2019 DOI: `10.1109/ASE.2019.00047`
Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 415–426.

Michel and Boubekeur, 2021 DOI: `10.1145/3450626.3459823`
Élie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. *ACM Transactions on Graphics* 40.4, 173:1–173:14.

Miller, 1956 DOI: `10.1037/h0043158`
George A. Miller. 1956. The Magical Number Seven, plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review* 63.2, pp. 81–97.

Miller et al., 1994 DOI: `10.1080/1049482940040202`
Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments* 4.2, pp. 140–158.

Miller and Myers, 2001
Robert C. Miller and Brad A. Myers. 2001. Interactive Simultaneous Editing of Multiple Text Regions. *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, pp. 161–174.

Minör, 1992 DOI: `10.1016/0020-7373(92)90002-3`
Sten Minör. 1992. Interacting with Structure-Oriented Editors. *International Journal of Man-Machine Studies*. Structure-Based Editors and Environments 37.4, pp. 399–418.

Mogk et al., 2018
Ragnar Mogk, Pascal Weisenburger, Julian Haas, David Richter, Guido Salvaneschi, and Mira Mezini. 2018. From Debugging Towards Live Tuning of Reactive Applications. *SPLASH '18: Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. LIVE'18, pp. 1–6.

Monig et al., 2015 DOI: `10.1109/BLOCKS.2015.7369001`
Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line between Blocks and Text in GP. *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, pp. 51–53.

Moody, 2009a DOI: `10.1109/TSE.2009.67`
Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35.6, pp. 756–779.

Moody, 2009b DOI: `10.1109/VLHCC.2009.5295275`
Daniel Moody. 2009. Theory Development in Visual Language Research: Beyond the Cognitive Dimensions of Notations. *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 151–154.

Moon et al., 2022 DOI: `10.1145/3546196.3550164`
David Moon, Andrew Blinn, and Cyrus Omar. 2022. Tylr: A Tiny Tile-Based Structure Editor. *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2022. ACM, pp. 28–37.

Moon et al., 2023 DOI: `10.1109/VL-HCC57772.2023.00016`
David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 71–81.

Morales et al., 2019 DOI: `10.1109/ACCESS.2019.2920124`
Jenny Morales, Cristian Rusu, Federico Botella, and Daniela Quinones. 2019. Programmer eXperience: A Systematic Literature Review. *IEEE Access* 7, pp. 71079–71094.

Mørch, 1997 DOI: `10.7551/mitpress/1966.003.0004`
Anders Mørch. 1997. Three Levels of End-User Tailoring: Customization, Integration, and Extension. *Computers and Design in Context*. The MIT Press.

Morris, 1938 ISBN: `978-0-226-57577-3`
Charles William Morris. 1938. Foundations of the Theory of Signs. *International Encyclopedia of Unified Science*. Chicago University Press, pp. 1–59.

Moskal et al., 2017 DOI: `10.1145/3105726.3106170`
Adon Christian Michael Moskal, Joy Gasson, and Dale Parsons. 2017. The 'Art' of Programming: Exploring Student Conceptions of Programming through the Use of Drawing Methodology. *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, pp. 39–46.

Mossienko, 2004
Maxim Mossienko. 2004. Structural Search and Replace: What, Why, and How-To. *OnBoard Magazine*.

Murphy-Hill and Black, 2010 DOI: `10.1145/1879211.1879216`
Emerson Murphy-Hill and Andrew P. Black. 2010. An Interactive Ambient Visualization for Code Smells. *Proceedings of the 5th International Symposium on Software Visualization - SOFTVIS '10*. ACM.

Myers et al., 2000 DOI: `10.1145/344949.344959`
Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* 7.1, pp. 3–28.

Myers, 1983 DOI: `10.1145/964967.801140`
Brad A. Myers. 1983. INCENSE: A System for Displaying Data Structures. *ACM SIGGRAPH Computer Graphics* 17.3, pp. 115–125.

Myers, 1990 DOI: `10.1016/S1045-926X(05)80036-9`
Brad A. Myers. 1990. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* 1.1, pp. 97–123.

Myers et al., 2004 DOI: `10.1145/1015864.1015888`
Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Communications of the ACM* 47.9, pp. 47–52.

Nadin, 1988 DOI: `10.1515/semi.1988.69.3-4.269`
Mihai Nadin. 1988. Interface Design: A Semiotic Paradigm. 69.3-4, pp. 269–302.

Naik et al., 2021 DOI: `10.1145/3472749.3474737`
Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code Using Query-by-Example. *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. ACM, pp. 84–99.

Neff and Stark, 2002 DOI: 10.7916/D8G44X47
Gina Neff and David C. Stark. 2002. Permanently Beta: Responsive Organization in the Internet Era. *Institute for Social and Economic Research and Policy Working Papers*.

Newell, 1994 ISBN: 978-0-674-92101-6
Allen Newell. 1994. *Unified Theories of Cognition*. Harvard University Press.

Newell and Simon, 1972 ISBN: 978-0-13-445403-0
Allen Newell and Herbert A. Simon. 1972. *Human Problem Solving*. Prentice-Hall.

Ni et al., 2021 DOI: 10.1145/3472749.3474748
Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. reCode : A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. ACM, pp. 258–269.

Nielsen, 1986 DOI: 10.1016/S0020-7373(86)80028-1
Jakob Nielsen. 1986. A Virtual Protocol Model for Computer-Human Interaction. *International Journal of Man-Machine Studies* 24.3, pp. 301–312.

Nipkow et al., 2002 DOI: 10.1007/3-540-45949-9
Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, eds. 2002. *Isabelle/HOL*. Vol. 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Noble and Biddle, 2002
James Noble and Robert Biddle. 2002. Notes on Postmodern Programming. *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*.

Norman, 2002 ISBN: 978-0-465-06710-7
Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books.

Olsen, 2007 DOI: 10.1145/1294211.1294256
Dan R. Olsen. 2007. Evaluating User Interface Systems Research. *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. UIST '07. ACM, pp. 251–258.

Omar et al., 2021 DOI: 10.1145/3453483.3454059
Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, pp. 511–525.

Omar et al., 2017 DOI: 10.1145/3009837.3009900
Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*. ACM Press, pp. 86–99.

Omar et al., 2012 DOI: 10.1109/ICSE.2012.6227133
Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 859–869.

Oney and Brandt, 2012 DOI: `10.1145/2207676.2208664`
Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. ACM, pp. 2697–2706.

Orwell, 1949
George Orwell. 1949. *Nineteen Eighty-Four*. Secker & Warburg.

Papert, 1982 ISBN: `978-0-465-04627-0`
Seymour Papert. 1982. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.

Peirce, 1976
Charles S. Peirce. 1976. *The New Elements of Mathematics*. Vol. 4 — Mathematical philosophy. Mouton Publishers.

Perera et al., 2022 DOI: `10.1145/3498668`
Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proceedings of the ACM on Programming Languages* 6.POPL, pp. 1–29.

Perkel, 2023 DOI: `10.1038/d41586-023-01833-0`
Jeffrey M. Perkel. 2023. Six Tips for Better Coding with ChatGPT. *Nature* 618.7964, pp. 422–423.

Petricek, 2016
Tomas Petricek. 2016. Programming Language Theory: Thinking the Unthinkable. *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group*. PPIG 2016, pp. 1–5.

Petricek, 2020 DOI: `10.22152/programming-journal.org/2020/4/8`
Tomas Petricek. 2020. Foundations of a Live Data Exploration Environment. *The Art, Science, and Engineering of Programming* 4.3, pp. 1–37.

Piner, 1972
Stephen D. Piner. 1972. *Expensive Typewriter*. Tech. rep. PDP-22. MIT.

Pit-Claudel, 2020 DOI: `10/ghs5sn`
Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, pp. 155–174.

Psallidas and Wu, 2018 DOI: `10.1145/3209900.3209904`
Fotis Psallidas and Eugene Wu. 2018. Provenance for Interactive Visualizations. *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA'18. ACM, pp. 1–8.

Puckette, 2002 DOI: `10.1162/014892602320991356`
Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal* 26.4, pp. 31–43.

Rädle et al., 2017 DOI: `10.1145/3126594.3126642`
Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. ACM, pp. 715–725.

Rambally, 1986 DOI: 10.1145/5600.5702
Gerard K. Rambally. 1986. The Influence of Color on Program Readability and Comprehensibility. *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '86. ACM, pp. 173–181.

Rauch et al., 2019 DOI: 10.22152/programming-journal.org/2019/3/9
David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming: Design and Implementation of an Integration of Live Examples into General-Purpose Source Code. *The Art, Science, and Engineering of Programming* 3.3, pp. 1–39.

Reid, 1980 DOI: 10.1145/567446.567449
Brian K. Reid. 1980. A High-Level Approach to Computer Document Formatting. *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '80. ACM, pp. 24–31.

Rein et al., 2018 DOI: 10.22152/programming-journal.org/2019/3/1
Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3.1, pp. 1–33.

Reis et al., 2021 DOI: 10.1145/3411763.3443455
Paulo Reis, John D Lees-Miller, and Sven Laqua. 2021. Merging SaaS Products In A User-Centered Way — A Case Study of Overleaf and ShareLaTeX. *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1–8.

Renom et al., 2022 DOI: 10.1145/3491102.3501877
Miguel A. Renom, Baptiste Caramiaux, and Michel Beaudouin-Lafon. 2022. Exploring Technical Reasoning in Digital Tool Use. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. ACM, pp. 1–17.

Renom et al., 2023 DOI: 10.1145/3544548.3581246
Miguel A. Renom, Baptiste Caramiaux, and Michel Beaudouin-Lafon. 2023. Interaction Knowledge: Understanding the 'Mechanics' of Digital Tools. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–14.

Resnick et al., 2009 DOI: 10.1145/1592761.1592779
Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Communications of the ACM* 52.11, pp. 60–67.

Resnick et al., 2005
Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. *NSF Workshop Report on Creativity Support Tools*.

Rickless, 2020
Samuel Rickless. 2020. Plato's Parmenides. *The Stanford Encyclopedia of Philosophy*. Spring 2020. Metaphysics Research Lab, Stanford University.

Roberts et al., 2015
Charles Roberts, Matthew Wright, and JoAnn Kuchera-Morin. 2015. Beyond Editing: Extended Interaction with

Textual Code Fragments. *Proceedings of the International Conference on New Interfaces for Musical Expression*. NIME 2015, pp. 126–131.

Roberts and Kuchera-Morin, 2012
Charlie Roberts and JoAnn Kuchera-Morin. 2012. Gibber: Live Coding Audio in the Browser. *Non-Cochlear Sound: Proceedings of the 38th International Computer Music Conference*. ICMC 2012. Michigan Publishing, pp. 64–69.

Rock and Palmer, 1990 DOI: `10.1038/scientificamerican1290-84`
Irvin Rock and Stephen Palmer. 1990. The Legacy of Gestalt Psychology. *Scientific American* 263.6, pp. 84–91.

Romat et al., 2019 DOI: `10.1145/3332165.3347934`
Hugo Romat, Emmanuel Pietriga, Nathalie Henry-Riche, Ken Hinckley, and Caroline Appert. 2019. SpaceInk: Making Space for In-Context Annotations. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. ACM, pp. 871–882.

Rorty, 2007 ISBN: `978-0-521-69835-1`
Richard Rorty. 2007. *Philosophy as Cultural Politics: Volume 4: Philosophical Papers*. Cambridge University Press.

Sarkar et al., 2022 DOI: `10.1109/VL/HCC53370.2022.9833131`
Advait Sarkar, Sruti Srinivasa Ragavan, Jack Williams, and Andrew D. Gordon. 2022. End-User Encounters with Lambda Abstraction in Spreadsheets: Apollo's Bow or Achilles' Heel? *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–11.

Satyanarayan et al., 2017 DOI: `10.1109/TVCG.2016.2599030`
Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23.1, pp. 341–350.

Satyanarayan et al., 2016 DOI: `10.1109/TVCG.2015.2467091`
Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22.1, pp. 659–668.

Saussure, 1916
Ferdinand de Saussure. 1916. *Cours de Linguistique Générale*. Payot.

Scholtz and Wiedenbeck, 2009 DOI: `10.1080/10447319009525970`
Jean Scholtz and Susan Wiedenbeck. 2009. Learning Second and Subsequent Programming Languages: A Problem of Transfer. *International Journal of Human-Computer Interaction*, pp. 51–72.

Schreiber et al., 2017
Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. 2017. *Transmorphic: Mapping direct manipulation to source code transformations*. Tech. rep. 110. Hasso Plattner Institute, University of Potsdam.

Schroeder and Saltzer, 1972 DOI: `10.1145/361268.361275`
Michael D. Schroeder and Jerome H. Saltzer. 1972. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM* 15.3, pp. 157–170.

Shaw, 2022 DOI: 10.1145/3480947
Mary Shaw. 2022. Myths and Mythconceptions: What Does It Mean to Be a Programming Language, Anyhow? *Proceedings of the ACM on Programming Languages* 4.HOPL, pp. 1–44.

Shieber, 1984 DOI: 10.3115/980491.980566
Stuart M. Shieber. 1984. The Design of a Computer Language for Linguistic Information. *Proceedings of the 10th International Conference on Computational Linguistics and 22nd Annual Meeting on Association for Computational Linguistics*. ACL '84/COLING '84. ACM, pp. 362–366.

Shneiderman, 1983 DOI: 10.1109/MC.1983.1654471
Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16.8, pp. 57–69.

Shneiderman and Mayer, 1979 DOI: 10.1007/BF00977789
Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences* 8.3, pp. 219–238.

Shrestha et al., 2018 DOI: 10.1109/VLHCC.2018.8506508
Nischal Shrestha, Titus Barik, and Chris Parnin. 2018. It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge. *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 177–185.

Shrestha et al., 2020 DOI: 10.1145/3377811.3380352
Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. ACM, pp. 691–701.

Sime et al., 1973 DOI: 10.1016/S0020-7373(73)80011-2
Max E. Sime, Thomas R. G. Green, and D. J. Guest. 1973. Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. *International Journal of Man-Machine Studies* 5.1, pp. 105–113.

Simonyi, 1995
Charles Simonyi. 1995. *The Death Of Computer Languages, The Birth of Intentional Programming*. Tech. rep. MSR-TR-95-52. Microsoft Research.

Simonyi et al., 2006 DOI: 10.1145/1167473.1167511
Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. ACM, pp. 451–464.

Solmi, 2005
Riccardo Solmi. 2005. Whole Platform. Ph.D. thesis. University of Bologna.

Sulír et al., 2018 DOI: 10.1016/j.jvlc.2018.10.001
Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual Augmentation of Source Code Editors: A Systematic Mapping Study. *Journal of Visual Languages & Computing* 49, pp. 46–59.

Sundermeyer et al., 2015 DOI: 10.1109/TASLP.2015.2400218
Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. 2015. From Feedforward to Recurrent LSTM Neural

Networks for Language Modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23.3, pp. 517–529.

Swamy et al., 2016 DOI: `10.1145/2837614.2837655`
Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, pp. 256–270.

Tanaka-Ishii, 2006 DOI: `10.1515/SEM.2006.006`
Kumiko Tanaka-Ishii. 2006. Dyadic versus Triadic Sign Models in Functional and Object-Oriented Computer Programming Paradigms. *Semiotica* 2006.158, pp. 213–231.

Taniguchi and Masuhara, 2022 DOI: `10.1145/3532512.3535225`
Rikito Taniguchi and Hidehiko Masuhara. 2022. CodeMap: A Graphical Note-Taking Tool Cooperating with an Integrated Development Environment. *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*. Programming '22. ACM, pp. 54–59.

Taniguchi et al., 2022 DOI: `10.3390/su14138084`
Yuta Taniguchi, Tsubasa Minematsu, Fumiya Okubo, and Atsushi Shimada. 2022. Visualizing Source-Code Evolution for Understanding Class-Wide Programming Processes. *Sustainability* 14.13, pp. 1–17.

Tanimoto, 1990 DOI: `10.1016/S1045-926X(05)80012-6`
Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages & Computing* 1.2, pp. 127–139.

Tanimoto, 2013 DOI: `10.1109/LIVE.2013.6617346`
Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, pp. 31–34.

Tanner et al., 2019 DOI: `10.1145/3290605.3300758`
Kesler Tanner, Naomi Johnson, and James A. Landay. 2019. Poirot: A Web Inspector for Designers. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1–12.

Tchernavskij, 2019
Philip Tchernavskij. 2019. Designing and Programming Malleable Software. Ph.D. thesis. Université Paris-Saclay.

Teitelbaum and Reps, 1981 DOI: `10.1145/358746.358755`
Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM* 24.9, pp. 563–573.

Tiazzoldi, 2016
Caterina Tiazzoldi. 2016. Combinatorial Architecture Methods for the Creation of Ambiance in Public Space. *Ambiances, Tomorrow. Proceedings of 3rd International Congress on Ambiances*. Vol. 2. International Network Ambiances, pp. 865–872.

Toomim et al., 2004 DOI: `10.1109/VLHCC.2004.35`
Michael Toomim, Andrew Begel, and Susan L. Graham. 2004. Managing Duplicated Code with Linked Editing. *2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 173–180.

Turing, 1950 DOI: `10.1093/mind/LIX.236.433`
Alan M. Turing. 1950. Computing Machinery and Intelligence. *Mind; a quarterly review of psychology and philosophy* LIX.236, pp. 433–460.

Utting et al., 2010 DOI: `10.1145/1868358.1868364`
Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. 2010. Alice, Greenfoot, and Scratch – A Discussion. *ACM Transactions on Computing Education* 10.4, pp. 1–11.

Vasek, 2012
Marie Vasek. 2012. Representing Expressive Types in Blocks Programming Languages. B.Sc. Thesis.

Vincur et al., 2017 DOI: `10.1145/3139131.3141785`
Juraj Vincur, Martin Konopka, Jozef Tvarozek, Martin Hoang, and Pavol Navrat. 2017. Cubely: Virtual Reality Block-Based Programming Environment. *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology*. VRST '17. ACM, pp. 1–2.

Voelter et al., 2019 DOI: `10.1007/s10270-016-0575-4`
Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. 2019. Lessons Learned from Developing Mbeddr: A Case Study in Language Engineering with MPS. *Software & Systems Modeling* 18.1, pp. 585–630.

Voelter and Lisson, 2014
Markus Voelter and Sascha Lisson. 2014. Supporting Diverse Notations in MPS' Projectional Editor. *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages*, pp. 7–16.

Voinov et al., 2022 DOI: `10.1145/3563835.3567663`
Philippe Voinov, Manuel Rigger, and Zhendong Su. 2022. Forest: Structural Code Editing with Multiple Cursors. *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2022. ACM, pp. 137–152.

Wang et al., 2021 DOI: `10.1145/3411764.3445249`
Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1–15.

Weintrop and Wilensky, 2017 DOI: `10.1145/3078072.3079715`
David Weintrop and Uri Wilensky. 2017. Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments. *Proceedings of the 16th International Conference on Interaction Design and Children - IDC'17*. ACM, pp. 183–192.

Weizenbaum, 1966 DOI: `10.1145/365153.365168`
Joseph Weizenbaum. 1966. ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine. *Communications of the ACM* 9.1, pp. 36–45.

Whatley et al., 2021 DOI: 10.1109/VL/HCC51201.2021.9576201
Daniel Whatley, Max Goldman, and Robert C. Miller. 2021. Snapdown: A Text-Based Snapshot Diagram Language for Programming Education. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 1–9.

Wiedenbeck, 1986 DOI: 10.1016/S0020-7373(86)80083-9
Susan Wiedenbeck. 1986. Beacons in Computer Program Comprehension. *International Journal of Man-Machine Studies* 25.6, pp. 697–709.

Williams and Gordon, 2021 DOI: 10.1109/VL/HCC51201.2021.9576272
Jack Williams and Andrew D. Gordon. 2021. Where-Provenance for Bidirectional Editing in Spreadsheets. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10.

Wilson, 2004 DOI: 10.1145/1039511.1039534
Gregory V. Wilson. 2004. Extensible Programming for the 21st Century: Is an Open, More Flexible Programming Environment Just around the Corner? *Queue* 2.9, pp. 48–57.

Wu et al., 2020 DOI: 10.1145/3379337.3415851
Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 152–165.

Wu et al., 2023 DOI: 10.1145/3586183.3606731
Zhiyuan Wu, Jiening Li, Kevin Ma, Hita Kambhamettu, and Andrew Head. 2023. FFL: A Language and Live Runtime for Styling and Labeling Typeset Math Formulas. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST '23. ACM, pp. 1–16.

Xiong et al., 2021 DOI: 10.48550/arXiv.2104.00682
Bo Xiong, Haoqi Fan, Kristen Grauman, and Christoph Feichtenhofer. 2021. Multiview Pseudo-Labeling for Semi-Supervised Learning from Video.

Yu et al., 2020 DOI: 10.1145/3379337.3415890
Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 126–139.

Zemanek, 1966 DOI: 10.1145/365230.365249
Heinz Zemanek. 1966. Semiotics and Programming Languages. *Communications of the ACM* 9.3, pp. 139–143.

Zhang et al., 2023 DOI: 10.1145/3544548.3581516
Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. VizProg: Identifying Misunderstandings By Visualizing Students' Coding Progress. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. ACM, pp. 1–16.

Zhang and Oney, 2020 DOI: 10.1145/3379337.3415824
Lei Zhang and Steve Oney. 2020. FlowMatic: An Immersive Authoring Tool for Creating Interactive Scenes in Virtual Reality. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 342–353.

Zhang et al., 2020 DOI: `10.1145/3379337.3415900`
Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. ACM, pp. 627–648.

Zhou et al., 2021 DOI: `10.48550/arXiv.2107.02053`
Kaiyang Zhou, Yongxin Yang, Yu Qiao, and Tao Xiang. 2021. MixStyle Neural Networks for Domain Generalization and Adaptation.

Zinenko et al., 2015
Oleksandr Zinenko, Cédric Bastoul, and Stéphane Huot. 2015. Manipulating Visualization, Not Codes. *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pp. 1–8.

Links

1. *FACT SHEET: President Biden Issues Executive Order on Safe, Secure, and Trustworthy Artificial Intelligence*. Oct. 30, 2023. The White House. `https://www.whitehouse.gov/briefing-room/statements-releases/2023/10/30/fact-sheet-president-biden-issues-executive-order-on-safe-secure-and-trustworthy-artificial-intelligence/`.
2. *Artificial Intelligence Act: deal on comprehensive rules for trustworthy AI*. Dec. 9, 2023. European Parliament. `https://www.europarl.europa.eu/news/en/press-room/20231206IPR15699/artificial-intelligence-act-deal-on-comprehensive-rules-for-trustworthy-ai`.
3. *Future of Coding*. `https://futureofcoding.org/`.
4. *LIVE workshop*. ACM SPLASH 2023. `https://2023.splashcon.org/home/live-2023`.
5. *PAINT workshop*. 2023. ACM SPLASH 2023. `https://2023.splashcon.org/home/paint-2023`.
6. *asm.js Working Draft*. `http://asmjs.org/spec/latest/%5C#annotations`.
7. Lucas Pluvinage. May 5, 2023. *Implementing value speculation in OCaml*. `https://www.lortex.org/articles/value-speculation-ocaml/`.
8. Francesco Mazzoli. July 20, 2021. *Beating the L1 cache with value speculation*. `https://mazzo.li/posts/value-speculation.html`.
9. *Full Grammar specification*. Python 3.12.2 documentation. `https://docs.python.org/3/reference/grammar.html`.
10. *Yacc*. Wikipedia. `https://wikipedia.org/wiki/Yacc`.
11. *GNU Bison*. `https://www.gnu.org/software/bison/`.
12. *ANTLR*. `https://www.antlr.org/`.
13. *Package naive-ebnf*. CTAN. `https://ctan.org/pkg/naive-ebnf`.
14. *COBOL blues*. Reuters Graphics. `http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html`.
15. Alicia Lee. Apr. 8, 2020. *Wanted urgently: People who know a half century-old computer language so states can process unemployment claims*. CNN Business. `https://edition.cnn.com/2020/04/08/business/coronavirus-cobol-programmers-new-jersey-trnd/index.html`.
16. *Jupyter*. `https://jupyter.org/`.

17. *CoqIDE*. Coq reference manual. https://coq.inria.fr/refman/practical-tools/coqide.html.

18. *TKGate 2.0*. https://bnoordhuis.github.io/tkgate/.

19. *Vega Editor*. https://vega.github.io/editor.

20. *BLAS (Basic Linear Algebra Subprograms)*. https://www.netlib.org/blas/.

21. *LAPACK — Linear Algebra PACKage*. https://www.netlib.org/lapack/.

22. *SageMath*. https://www.sagemath.org/.

23. *NumPy*. https://numpy.org/.

24. *Data Scientists*. Sept. 6, 2023. U.S. Bureau of Labor Statistics. https://www.bls.gov/ooh/math/data-scientists.htm.

25. *Pandas*. https://pandas.pydata.org/.

26. *Matplotlib*. https://matplotlib.org/.

27. *Overleaf*. https://www.overleaf.com.

28. *Frescobaldi*. https://www.frescobaldi.org/.

29. *LilyPond*. http://lilypond.org/.

30. *Edotor*. https://edotor.net/.

31. *Graphviz*. https://graphviz.org/.

32. *troff — The Text Processor for Typesetters*. https://troff.org/.

33. *GNU roff (groff)*. https://www.gnu.org/software/groff/.

34. *PEP 287 — reStructuredText Docstring Format*. https://peps.python.org/pep-0287/.

35. *Documentation*. Rust By Example. https://doc.rust-lang.org/rust-by-example/meta/doc.html.

36. Will Crichton. 2021. *A New Medium for Communicating Research on Programming Languages*. https://willcrichton.net/nota/.

37. *Observable*. https://observablehq.com/.

38. *Adobe PostScript*. https://www.adobe.com/products/postscript.html.

39. *Package Metafont*. CTAN. https://ctan.org/pkg/metafont.

40. *The TikZ and PGF Packages*. https://tikz.dev/.

41. *Mermaid*. https://mermaid.js.org/.

42. *ultimate-guitar/Tabdown: Tabdown is an open mark-up language for text tabs & chords*. GitHub. https://github.com/ultimate-guitar/Tabdown.

43. *Alda*. https://alda.io/.

44. *Hour of Code*. https://hourofcode.com/.

45. Megan Smith. Jan. 30, 2016. *Computer Science For All*. The White House. https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all/.

46. *Programme d'enseignement de spécialité de numérique et sciences informatiques de la classe de première de la voie générale*. Jan. 22, 2019. Bulletin officiel de l'éducation nationale, de la jeunesse et des sports. https://www.education.gouv.fr/bo/19/Special1/MENE1901633A.htm.

47. *Programme de l'enseignement de spécialité de numérique et sciences informatiques de la classe terminale de la voie générale*. July 25, 2019. Bulletin officiel de l'éducation nationale, de la jeunesse et des sports. https://www.education.gouv.fr/bo/19/Special8/MENE1921247A.htm.

48. Harry McCracken. Apr. 29, 2014. *Fifty Years of BASIC, the Programming Language That Made Computers Personal*. Time. https://time.com/69316/basic/.

49. *Logo History*. Logo Foundation. https://el.media.mit.edu/logo-foundation/what_is_logo/history.html.

50. *Scratch's Annual Report 2022*. Scratch Foundation. https://www.scratchfoundation.org/annualreport.

51. *A Logo Primer*. Logo Foundation. https://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html.

52. *Blockly*. Google for Developers. https://developers.google.com/blockly.

53. *Orca*. https://hundredrabbits.itch.io/orca.

54. *Shadertoy*. https://www.shadertoy.com.

55. Flexi. Jan. 6, 2016. *Expansive reaction-diffusion*. Shadertoy. https://www.shadertoy.com/view/4dcGW2.

56. *Submarine.ijs*. Code Poetry. https://code-poetry.com/submarine.

57. Benj Edwards. Jan. 24, 2013. *The Never-Before-Told Story of the World's First Computer Art (It's a Sexy Dame)*. The Atlantic. https://www.theatlantic.com/technology/archive/2013/01/the-never-before-told-story-of-the-worlds-first-computer-art-its-a-sexy-dame/267439/.

58. *TOPLAP*. https://toplap.org/.

59. *Blender Python API*. https://docs.blender.org/api/current/.

60. *Rhino.Python Guides*. Rhino developer. https://developer.rhino3d.com/guides/rhinopython/.

61. Frank J. Swetz. *Mathematical Treasure: Ratdolt's Euclid's Elements*. Mathematical Association of America. https://www.maa.org/press/periodicals/convergence/mathematical-treasure-ratdolts-euclids-elements.

62. *Erhard Ratdolt's edition of Euclid's Elements*. The Folger Shakespeare Library's digital image collection. `https://luna.folger.edu/luna/servlet/s/e7b5y0`.

63. *The first six books of the Elements of Euclid*. Internet Archive. `https://archive.org/details/firstsixbooksofe00byrn/page/n33/mode/2up`.

64. *Code Poetry*. `https://code-poetry.com/`.

65. *What is the best comment in source code you have ever encountered?* Stack Overflow. `https://stackoverflow.com/q/184618`.

66. *code/game/q_math.c file of Quake III Arena's source code (lines 549–572)*. Software Heritage. `https://archive.softwareheritage.org/browse/content/sha1_git:bb0faf6919fc60636b2696f32ec9b3c2adb247fe/#L549-L572`.

67. *Psychology of Programming Interest Group*. `https://www.ppig.org/`.

68. Tony Hoare. Mar. 2009. *Null References: The Billion Dollar Mistake*. QCON London. `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`.

69. *The neural networks behind Google Voice transcription*. Google Research Blog. `https://ai.googleblog.com/2015/08/the-neural-networks-behind-google-voice.html`.

70. Alex Hern. Mar. 29, 2023. *Elon Musk joins call for pause in creation of giant AI 'digital minds'*. The Guardian. `https://www.theguardian.com/technology/2023/mar/29/elon-musk-joins-call-for-pause-in-creation-of-giant-ai-digital-minds`.

71. *ACM Policy on Authorship*. Apr. 20, 2023. ACM. `https://www.acm.org/publications/policies/new-acm-policy-on-authorship`.

72. *Copilot*. GitHub. `https://github.com/features/copilot`.

73. Bret Victor. Sept. 2012. *Learnable programming*. `http://worrydream.com/LearnableProgramming`.

74. *Experiences with the New Java 5 Language Features*. `https://www.oracle.com/technical-resources/articles/java/java-5-features.html`.

75. *ECMAScript 2020 Language Specification*. `https://262.ecma-international.org/11.0/`.

76. Thomas Schoch. 2006. *"Hello, world!" in Piet*. `https://retas.de/thomas/computer/programs/useless/piet/explain.html`.

77. *Piet*. Esolang. `https://esolangs.org/wiki/Piet`.

78. *Brainfuck*. Esolang. `https://esolangs.org/wiki/Brainfuck`.

79. *Typing glyphs*. APL Wiki. `https://aplwiki.com/wiki/Typing_glyphs`.

80. David Naccache. 2009. *Quel est le QI de Dionaea Muscipula ?* Les Ernest. `https://savoirs.ens.fr/expose.php?id=3197`.

81. *PdFileFormat*. Pure Data. `http://puredata.info/docs/developer/PdFileFormat`.

82. *The Jupyter Notebook Format*. IPython. https://ipython.org/ipython-doc/3/notebook/nbformat.html.

83. *Konrad Zuse Internet Archive*. http://zuse.zib.de/.

84. *Dynamicland*. https://dynamicland.org/.

85. Geoffrey Litt. *Experiments in Dynamicland*. https://www.geoffreylitt.com/projects/dynamicland.

86. *GDB: The GNU Project Debugger*. https://www.sourceware.org/gdb/.

87. *Language Server Protocol*. https://microsoft.github.io/language-server-protocol/.

88. *Debug Adapter Protocol*. https://microsoft.github.io/debug-adapter-protocol//.

89. *Chrome DevTools Protocol*. https://chromedevtools.github.io/devtools-protocol/.

90. *Light Table*. http://lighttable.com/.

91. *Collaborate with Live Share*. Visual Studio Code. https://code.visualstudio.com/learn/collaboration/live-share.

92. *Code With Me*. JetBrains. https://www.jetbrains.com/code-with-me/.

93. *Regulex*. https://jex.im/regulex.

94. *RegExr*. https://regexr.com/.

95. *Commanding the editor*. JetBrains' MPS Documentation. https://www.jetbrains.com/help/mps/commanding-the-editor.html.

96. *Introduction to Nodes*. Blender 2.79 Manual. https://docs.blender.org/manual/en/2.79/render/blender_render/materials/nodes/introduction.html.

97. *Blueprint Visual Scripting*. Unreal Engine Documentation. https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/.

98. *Nodes*. https://nodes.io/.

99. *Enso*. https://enso.org.

100. *P5.js*. https://p5js.org/.

101. *Create a UI by using XAML Designer*. https://learn.microsoft.com/en-us/visualstudio/xaml-tools/creating-a-ui-by-using-xaml-designer-in-visual-studio.

102. *Develop a UI with Views*. Android Developers. https://developer.android.com/studio/write/layout-editor.

103. *Creating your app's interface with SwiftUI*. Apple Developer Documentation. https://developer.apple.com/documentation/xcode/creating-your-app-s-interface-with-swiftui.

104. *Adobe Dreamweaver*. https://www.adobe.com/fr/products/dreamweaver.html.

105. *Compositor*. https://compositorapp.com/.

106. Bret Victor. Jan. 20, 2012. *Inventing on Principle*. CUSEC conference. http://worrydream.com/InventingOnPrinciple.

107. *About GitHub Copilot Chat*. GitHub Docs. https://docs.github.com/en/copilot/github-copilot-chat/about-github-copilot-chat.

108. *Cursor — The AI-first Code Editor*. https://cursor.sh/.

109. Geoffrey Litt. July 2023. *Codifying a ChatGPT workflow into a malleable GUI*. https://www.geoffreylitt.com/2023/07/25/building-personal-tools-on-the-fly-with-llms.

110. Philip Guo. July 25, 2023. *Real-Real-World Programming with ChatGPT*. https://www.oreilly.com/radar/real-real-world-programming-with-chatgpt/.

111. *TextEditorDecorationType*. VS Code API. https://code.visualstudio.com/api/references/vscode-api/%5C#TextEditorDecorationType.

112. *Example: Decorations*. CodeMirror. https://codemirror.net/examples/decoration/.

113. *Natto*. https://natto.dev/.

114. *Hydra — Live coding video synth*. https://hydra.ojack.xyz.

115. Brian Harvey and Jens Mönig. 2020. *Snap! Reference Manual*. https://snap.berkeley.edu/snap/help/SnapManual.pdf.

116. *Features and Benefits*. Eclipse Capella. https://eclipse.dev/capella/features.html.

117. *Ed — a line-oriented text editor*. https://www.gnu.org/software/ed/.

118. *vi*. Linux manual page. https://man7.org/linux/man-pages/man1/vi.1p.html.

119. *Semantics of a Foundational Subset for Executable UML Models*. Object Management Group. https://www.omg.org/spec/FUML/1.5/About-FUML.

120. *Eclipse Sirius*. https://eclipse.dev/sirius/.

121. *Eclipse Capella*. https://eclipse.dev/capella/.

122. *Glamorous Toolkit*. https://gtoolkit.com/.

123. *feenk*. https://feenk.com/.

124. *Tables Generator*. https://tablesgenerator.com.

125. *CSS Gradient*. https://cssgradient.io/.

126. Patrick Dubroy. 2014. *Future Programming workshop*. ACM SPLASH 2014. https://harc.github.io/moonchild/.

127. *The top programming languages*. GitHub Octoverse 2022. https://octoverse.github.com/2022/top-programming-languages.

128. *The Developer Survey 2023*. Stack Overflow. https://survey.stackoverflow.co/2023/.

129. *pdfTeX*. https://www.tug.org/applications/pdftex/.

130. *XƎTEX*. https://tug.org/xetex/.

131. *LuaTEX*. https://www.luatex.org/.

132. *SILE*. https://sile-typesetter.org/.

133. *Pollen: the book is a program*. https://docs.racket-lang.org/pollen/.

134. *AUCTeX*. https://www.gnu.org/software/auctex/.

135. *LyX — The Document Processor*. https://www.lyx.org/.

136. *pandas.DataFrame.to_latex*. Pandas 2.2.1 documentation. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_latex.html.

137. *LaTeXTools: A LaTeX Plugin for Sublime Text 2 and 3*. https://latextools.readthedocs.io/en/latest/.

138. *Explorable Explanations*. https://explorabl.es/.

139. *MyST Markdown — Tools for the future of technical communication*. https://mystmd.org/.

140. *Typst*. https://typst.app/.

141. *Potluck — Dynamic documents as personal software*. Ink & Switch. https://www.inkandswitch.com/potluck/.

142. *Texifier*. https://www.texifier.com/.

143. *Texifier — The Story So Far*. https://www.texifier.com/blog/texpadtex-story-so-far.

144. *Package amsmath*. CTAN. https://ctan.org/pkg/amsmath.

145. *Package graphicx*. CTAN. https://ctan.org/pkg/graphicx.

146. *Visual Studio Code*. https://code.visualstudio.com/.

147. *exsitu-projects/ilatex: i-LATEX is a prototypal LATEX editor with interactive code representations we call transitionals*. GitHub. https://github.com/exsitu-projects/ilatex.

148. *Is there a BNF grammar of the TeX language?* TEX StackEchange. https://tex.stackexchange.com/q/4201.

149. *mozilla/pdf.js: PDF Reader in JavaScript*. GitHub. https://github.com/mozilla/pdf.js/.

150. Michel Beaudouin-Lafon. Oct. 22, 2019. *A World Without Apps*. ACM UIST 2019 Visions. https://www.youtube.com/watch?v=ntaudUum06E.

151. Geoffrey Litt. Mar. 2021. *Bring Your Own Client*. https://www.geoffreylitt.com/2021/03/05/bring-your-own-client.

152. *ELPA: GNU Emacs Lisp Package Archive*. https://elpa.gnu.org/.

153. *Atom: The hackable text editor*. GitHub. https://github.com/atom/atom.

154. Geoffrey Litt. May 23, 2023. *Dynamic documents as personal software*. Causal Islands 2023. https://www.youtube.com/watch?v=bJ3i4K3hefI.

155. *CodeQL*. GitHub. https://codeql.github.com/.

156. *Tree-sitter*. https://tree-sitter.github.io/tree-sitter/.

157. *React*. https://react.dev/.

158. *exsitu-projects/lorgnette: LORGNETTE is a framework to create code editors with malleable projections of pieces of code*. GitHub. https://github.com/exsitu-projects/lorgnette.

159. *Monaco Editor*. https://microsoft.github.io/monaco-editor/.

160. *syntax-tree/mdast-util-from-markdown: mdast utility to parse markdown*. GitHub. https://github.com/syntax-tree/mdast-util-from-markdown.

161. *syntax-tree/mdast-util-to-markdown: Markdown Abstract Syntax Tree format*. GitHub. https://github.com/syntax-tree/mdast.

162. *syntax-tree/mdast-util-to-markdown: mdast utility to serialize markdown*. GitHub. https://github.com/syntax-tree/mdast-util-to-markdown.

163. *Layering Rolling Averages over Raw Values*. Vega-Lite. https://vega.github.io/vega-lite/examples/layer_line_rolling_mean_point_raw.html.

164. *seaborn.barplot*. Seaborn 0.13.2 documentation. https://seaborn.pydata.org/generated/seaborn.barplot.html.

165. *matplotlib.patches.Rectangle*. Matplotlib 0.13.3 documentation. https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.Rectangle.html#matplotlib.patches.Rectangle.

166. *Seaborn: statistical data visualization*. https://seaborn.pydata.org/.

167. *Writing Markup with JSX*. https://react.dev/learn/writing-markup-with-jsx.

168. *What are source maps?* https://web.dev/articles/source-maps.

169. *Lively*. https://lively-next.org/.

170. Maneesh Agrawala. Mar. 30, 2023. *Unpredictable Black Boxes are Terrible Interfaces*. Maneesh's Substack. https://magrawala.substack.com/p/unpredictable-black-boxes-are-terrible.

171. Geoffrey Litt. Mar. 2023. *Malleable software in the age of LLMs*. https://www.geoffreylitt.com/2023/03/25/llm-end-user-programming.

172. *3Blue1Brown*. https://www.3blue1brown.com.