Online Polyglot Programming Education with LFT (Lingua Franca Transformer)

Sokratis Karkalas[1], Filothei Chalvatza[1] and Manolis Mavrikis[2]
[1]*Simple, Farkadona, Greece*
[2]*UCL Knowledge Lab, University of London, London, U.K.*

Keywords: Authoring Tools, Programming Education, Exploratory Learning, Web Learning Components.

Abstract: This paper presents a novel approach to improve reusability and augment the educational value of web components through a polyglot environment. The idea is to enable communication with web components in a language neutral context by provisioning, along with the instructions, the grammar specification of the language used for those instructions and thus make the system agnostic of the language being used. This ability promotes reusability in the sense that learning designers are able to utilise learning materials using the language they feel more comfortable with or the language that seems to be more suitable for the task. Another benefit is that learners can make better use of the same learning environments they are accustomed to using through different languages. This allows learners to experiment with different programming paradigms, use more expressive or specialised languages and combine them with the concepts available in the learning environment of preference. In the context of this project we developed an authoring environment that allows the specification of any language and the automatic generation of parsers that can be used to dynamically transpile code into JavaScript. Preliminary testing confirmed that the idea is feasible and gave us positive feedback for future development.

1 INTRODUCTION

An important aspect of teaching programming to novices is the choice of language. The language used to encode concepts and techniques developed during learning should be expressive enough to carry them throughout the learning process. Different languages vary in how specialized they are, how close they are to the physical architecture of the machine, and other factors. Not all languages may be suitable for teaching programming, depending on the educational course, approach, and other parameters.

The Lingua Franca Transformer (LFT) is a system intended for transformations from any language to JavaScript. LFT is a tool that can be used to develop the definition of any language and generate the respective transpiler to JavaScript. The tool can be used to experiment with new languages designed for educational or other purposes. LFT makes it possible to use existing learning environments with different languages, and new programming languages may also be defined with the same ease as existing ones.

2 RELATED WORK

The question of which language to use for introductory programming education is almost as old as CS education itself (Becker and Quille, 2019; Sobral, 2019) and it is an important one (Laakso et al., 2008; Minor and Gewali, 2004). Comparisons between different languages and programming paradigms with respect to the educational value they carry in CS1 courses are a constant and recurring theme in the academic literature (Smith and Rickman, 1976; Wexelblat, 1979; Tharp, 1982; Duke et al., 2000; Mannila and de Raadt, 2006; Azad and Smith, 2014; Goosen, 2008; Parker et al., 2006; Sobral, 2019; Irimia, 2001; Wainer and Xavier, 2018; Lewis et al., 2016). Nevertheless it seems that there is still no definitive answer as to which should be the most preferable language to convey the necessary concepts to newcomers in the discipline. The fact that the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) have never given clear curriculum recommendations for this subject is indicative of the situation (Sobral, 2020). Naturally decisions about this are being made on an ad hoc ba-

Karkalas, S., Chalvatza, F. and Mavrikis, M.
Online Polyglot Programming Education with LFT (Lingua Franca Transformer).
DOI: 10.5220/0011981400003470
In *Proceedings of the 15th International Conference on Computer Supported Education (CSEDU 2023) - Volume 1*, pages 305-312
ISBN: 978-989-758-641-5; ISSN: 2184-5026
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

sis by course directors and teachers based on criteria that vary. An extensive list of typical criteria used are given in (Ezenwoye, 2018; Lindoo, 2020; Naveed et al., 2018; Sobral, 2019; Irimia, 2001; Wainer and Xavier, 2018; Sobral, 2021). A mixed approache that proposes multiple languages for CS1 courses is given in (Lindoo, 2020) where the use of a multi-programming language approach reduced the drop-out rates and increased pass - success rates significantly.

Programming is by nature a craft and as such it is learnt better through experimentation and exploration in a constructivist context. Learning in this context is accomplished through personal inquiry, divergent thinking and self-directed learning (Hannafin et al., 1999). Naturally, learning environments that promote that type of learning are virtual worlds (microworlds). There have been numerous attempts in academia and the industry to address this need through projects like (Solomon and Papert, 1976; PATTIS, 1981; Bergin et al., 1996; Bergin et al., 2005; Becker, 2001; Kahn, 1996; Cooper et al., 2000; Van Haaster and Hagan, 2004; Henriksen and Kölling, 2004; Kölling and Henriksen, 2005; Zschaler et al., 2014; Kynigos and Latsi, 2007; Maloney et al., 2008). In all those cases, the environments developed are tied to a specific language though. The language used in Turtle Graphics (Solomon and Papert, 1976; Papert, 1980) and Malt+ (Kynigos and Latsi, 2007) is LOGO and the one used in Greenfoot (Henriksen and Kölling, 2004) is Java. Following the discussion above about the flexibility to allow different languages for introductory programming education we consider this a potentially limiting factor that imposes considerable constraints when it comes to utilising those environments and exploiting the full potential of them. It would be really beneficial to have the ability to use the same environments with other languages like C# and Python.

Another challenge that needs to be addressed is that the de facto platform for development and deployment of educational software nowadays is the web (Wen et al., 2020). Naturally applications developed in that context are supposed to be executed within a web browser. This offers a lot of flexibility as it allows for easier development, deployment, maintenance, upgrading and administration but at the same time it introduces barriers in terms of what can actually be processed in the context of a browser. The most obvious concerns are the limitations of memory, processing resources and dependencies on certain functionalities. The not so obvious concern but a crucial one is that the low-level machine language of the browser is JavaScript. That realisation reveals an additional concern that regardless of the language being used to express things at a high level the common denominator is always JavaScript as execution takes place in the underlying JavaScript runtime engine supported by the browser. The alternative would be to utilise a server at the back end to actually process requests in native runtime environments and then send back to the browser the results (Serrano et al., 2006; Balat et al., 2009). A roundtrip to a server for every request is not a scalable and therefore viable solution for web deployments, therefore we are not considering that option at all. Executing everything locally in the browser, in a disconnected fashion is both good and bad. The advantage is that it is guaranteed that standard Javascript will always be the same in any platform and that means that no recompilation will be necessary for different machine and platform architectures. This guarantee of uniformity gives independence, flexibility and reduces the cost of developing solutions. On the flip side though, the limitations and the somewhat distinct idiosyncrasy of JavaScript will always be a limiting factor to what can be expressed and executed in the context of a browser. It is literally impossible to keep the exact same semantics between any other language and Javascript and it is also impossible to rely on certain operating system abstractions like multi-threading, file systems and so on. Thus, the chances are very high that there will be certain compromises when expressing and executing foreign code in the browser but these are compromises that we should be willing to accept since the benefit outweighs the cost. Another approach that has been proposed by (Canou et al., 2012) is to perform a compilation of code in its native environment and port the target bytecode to a virtual machine running in Javascript on the browser. The idea is simple, efficient and effective if it is known in advance what is required to be transported to the browser but in the context of online exploratory learning environments the expectation is that the code is given in real time by the learners and the aim is for this code to be directly executable in real time so that the user can manipulate constructs in those environments. This requires a JiT (Just in Time) transpilation of code from any language to Javascript directly in the browser.

Finally, the most recent development in this area is the deployment of fully fledged virtual machines in the browser that are able to execute absolutely unmodified source code written in different languages as if they are in their native environment (Wen et al., 2020). This is a solution that entails the execution of an OS via a virtual hypervisor running in the browser. The prospect of having the ability to operate like that in a browser exceeds even the most ambitious expectations and this is something definitely worth invest-

ing in future research projects. Nevertheless, the obvious obstacles to using this approach currently is that it is not mature enough to provide well defined interfacing options so that the functionality of the code being run in the VM can be externalised and used directly to manipulate objects in the browser and also the cost of running a heavyweight component like the VM itself may pose considerable resource and performance deficiencies in such a limited platform like the browser.

The work in this paper is inspired by (Ford, 2004; Matsumura and Kuramitsu, 2016). The aim is to have the ability to manipulate any object that exposes an API in the context of a browser with instructions given in any language. The idea is simple. Instead of just giving the code intended to perform some action, we provide both the code and the language specification that the code adheres to. The language specification is used to generate a parser able to understand the code and another process generates its equivalent in Javascript. The Javascript code is then executed in the usual way and all that happens in an automated fashion. Parsers can either be generated on the fly, or generated only once and reused thereafter if performance issues arise. This simple mechanism allows any number of diverse web widgets in the browser ecosystem to operate in a truly polyglot environment with minimal dependencies and performance overheads since everything takes place locally.

3 DESIGN CONSIDERATIONS - PARSERS

The main objective in this part is to identify the optimal solution to generate parsers for input languages. One option is to design an authoring environment to enable efficient development of custom parsers by hand. Building a custom parser by hand is a big endeavour and there has to be a good reason to justify the effort. Normally we consider this option only when there are specific requirements that cannot be satisfied by using a typical parser generator. This is typically the case when the input language is very specialised or there are particular requirements in terms of performance or integration that cannot be met. A more flexible approach that covers a wide range of cases and promises shorter development time is to use a parser generator. Parser generators may offer a cost effective alternative to building custom parsers but they are not trivial to use themselves. They come with constraints and limitations in terms of what input grammars they support and what types of target languages they are suitable for. Not all parser generators are compatible with all types of grammars. This consideration is needed because it shows the available options along with potential constraints and limitations with existing tools and technologies. The first and foremost constraint in our case is the fact that we expect the parser generator to be able to operate in the browser. Therefore the expectation is to find a tool that is itself developed in JavaScript. This criterion narrows down significantly the available range of options.

For that part a domain analysis is done to see what the available tools and technologies are in the market and what the technical specifications for those tools are.At the time that this part of the research was done the major players were Jison, PEG.js, ANTLR and Chevrotrain. The main two rivals in terms of popularity (npm trends) have always been PEG.js and Jison. These are the oldest tools in the market. PEG.js has always been by far the most popular tool in the market. This trend has only recently changed in favour of chevrotain that superseded PEG.js in 2022. Popularity (No of downloads per unit of time) as well as support and maintenance (frequency of updates) are important criteria in our decision because they relate to the size of the target user segment we aim for. Another consideration is performance benchmarks that show how performant the tools are when parsing text in JSON notation (typical test). Unfortunately, these reports proved not to be very helpful because they show ambiguous and sometimes contradicting results favouring one or the other tool. A possible explanation for that might be the fact that the results are heavily influenced by the range of use cases considered.

Jison was written in 2009 as an assisting technology for a compilers course. It creates bottom-up parsers that recognize context-free languages expressed in LALR(1), LR(0), SLR(1) grammars. Jison is an JavaScript implementation of Bison. Bison is a parser generator written in the 1990s to convert context-free grammars into deterministic parsers. PEG.js was written in 2010 to generate top-down parsers in JavaScript. The format and syntax of the grammar used is similar to the one used in Jison. PEG.js recognises a form of grammars that is alternative to context-free grammars and uses the packrat parsing technique to support any language defined by an LL(k) or LR(k) grammars (Bouilliez, 2014).

4 THE PARSER GENERATOR

The tool we chose to work with in the first prototype is PEG.js. The tight integration of the tool with JavaScript, the concise syntax of the supported input language for the grammar, the quality and clarity of

the documentation available as well as its huge popularity in the market contributed to the decision. The tool follows the Parsing Expression Grammar (PEG) formalism. PEG (Ford, 2004) formalises a language in terms of an analytic grammar and that means that syntax rules correspond more directly to the structure and the semantics of the parser generated for that language. PEG is designed specifically to accommodate the needs of programming language and compiler writing and is considered more powerful than traditional LL and LR formalisms. The PEG.js implementation accepts syntax rules in PEG and allows inline statements expressed in JavaScript. That means that syntax rules in JavaScript can be embedded in the grammar document in order to direct the generated parser to reshape the resulting AST and transform it to whatever output language we desire. Tools that lack this functionality have to resort to separate components to process the resulting trees and make them conformant to the desired specification. There is no separate lexer component for the identification of tokens as this functionality is integrated with the tool.

The following is a very simple example of a language specification given in PEG:

```
text = words:(w:word space? {return w;})*
{return words;}
word = letter+
letter = [a-zA-Z0-9]
space = " "
```

Parsing starts with the rule given first. The initial rule is called text and references two other rules named word and space respectively. These rules need to be defined as well. The rule word is defined as one or more letters. Letter is defined as any alphabetical or numerical character (English alphabet). Space is defined as the whitespace character. Going upwards, text is defined as any sequence of word tokens optionally followed by a space. The JavaScript snippets give instructions to the parser to return only the matched word tokens, not the spaces.

5 THE ARCHITECTURE

LFT is primarily intended to enable dynamic transformations of new and existing languages into JavaScript. This is based on the premise that a polyglot environment in the browser can add educational value to existing learning materials. Learning materials in that context take the form of web components (widgets) that encapsulate some functionality exposed via a GUI and/or an API. Learning designers or learning application developers communicate with these widgets programmatically to initialise them with content, to monitor learner activity and progress, to assess learner achievements and so on. Learners on the other hand sometimes communicate with them programmatically via the GUI to create and manipulate content and learn new concepts in the process. In both cases we have a widget that requires instructions expressed in JavaScript to operate.

If the activity logic needs to be expressed in an alternative language, we need a mechanism to transform this code dynamically into JavaScript before we send it off to the widget.

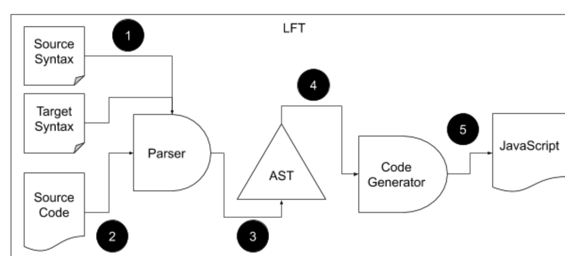

Figure 1: LFT Architecture.

LFT accepts the formal specification of the input language as well as the syntax rules for the output language in the same text and generates a parser for that language. The parser is then used to process the source code in the target language and produce an equivalent representation in the intended form. This form in most of the cases is expected to be an Abstract Syntax Tree (AST) that conforms to the ESTree specification, formerly known as SpiderMonkey AST. This is not mandatory but rather a convention used for convenience if the target language is JavaScript. AST is a language neutral, generic tree-like representation of the syntactic structure and the language constructs found in the source code. It is abstract in the sense that it does not depict every little detail found in the syntax or the semantics of the code but only the structure. The ESTree specification is assumed because it is a standard format for JavaScript. In LFT the ESTree AST serves as the common language denominator because its specification is much simpler than that of JavaScript itself and it allows automated conversion into JavaScript via compatible code generators. The final step is to dynamically evaluate the JavaScript code and execute the instruction in the component.

6 THE TOOL

The tool is designed to facilitate the authoring process of syntax rules for new and existing languages in PEG. The assumed target language is the ESTree

specification. The interface is split in panels organised as tabs. The first tab shows the editor that can be used to author the syntax rules for the input language - On the right there is another editor that can be used to test these rules with some text expressed in the language being specified.

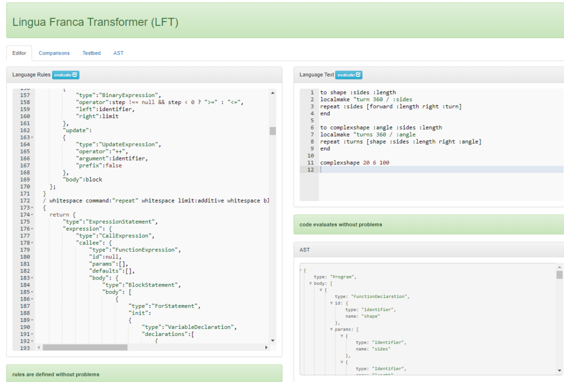

Figure 2: LFT Editor.

If the rules are well formed and the input is valid the resulting AST is displayed in another frame. This part of the interface can be used for the specification of any output language and ESTree output is not assumed. If the target language is ESTree then a more careful inspection of the output is needed. For that there is an additional tab that allows the author to perform a comparative analysis of the output between the statements given in the new language and JavaScript.

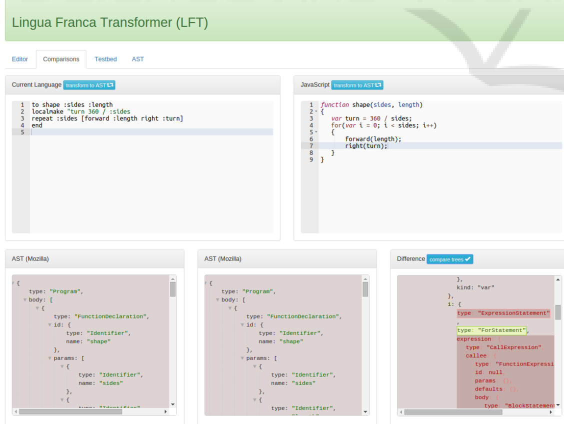

Figure 3: Comparison Tab.

The comparisons tab gives two editors, one for each language. The author is supposed to provide equivalent statements in both languages to express the same operation and examine carefully the generated ASTs.

If the ASTs don't match up then there is a mistake in the PEG syntax. An automatic comparison is performed and differences are highlighted in a third frame. The next tab is called Testbed and is designed to give an idea of how the newly defined language could be used to manipulate a turtle in a Microworld.

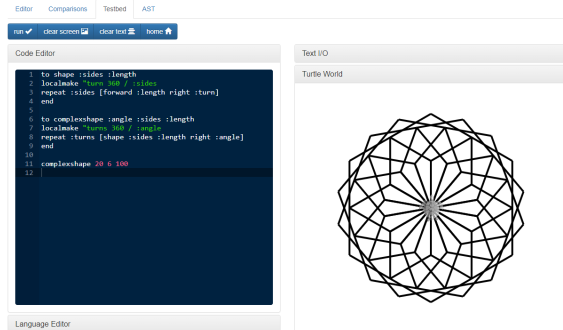

Figure 4: LFT Testbed.

7 IMPLEMENTATION DETAILS

In the context of LFT the parser is a JavaScript object that gets generated dynamically by a third party component found in the PEG.js[1] library. This component is a parser generator that conforms to the PEG formalism as presented earlier. Once the parser is created it can be used to parse code expressed in the input language and generate code expressed in the output language dynamically. LFT was designed to allow transformations between any two languages but its primary goal is to allow the creation of transpilers to JavaScript. As stated previously the purpose for this is to enable the use of different languages in the context of web browsers. Therefore, the expected target language is typically an Abstract Syntax Tree (AST) that conforms to the ESTree3 specification. This specification is the most common and standard format for JavaScript AST. Once the parser is created the next step is to enter a sample text in the input language and let the parser generate the respective AST for inspection. The AST is formatted appropriately and placed in a graphical control to enable visual inspection. A third-party library called JSON Viewer4[2] is used for the formatting. Further tests are done in the 'Comparisons' tab and that entails the automatic comparison and visual inspection of ASTs that are supposed to be identical. The test at that stage is to give two equivalent statements, one in the input language and the other in JavaScript and check if the resulting ASTs are exactly the same. Parsing the JavaScript code is done by a tool named Esprima5[3]. Automatic

[1] https://pegjs.org/
[2] https://github.com/abodelot/jquery.json-viewer
[3] https://esprima.org/

comparison is performed by a third party component called objectDiff6[4]. This checks both ASTs, detects discrepancies in the structures and displays visual indicators of the differences. This allows for a much more succinct testing of the parser conformity with the ESTree specification[5]. The final and ultimate test is to evaluate the parser in the challenging context of a microworld. This entails writing code in the input language to control and manipulate a turtle in a turtle-world. This environment is an own component built specifically for this project. The environment was developed in HTML5, JavaScript and Raphael7[6]. The API exposed by this microworld expects to be used in JavaScript.

Therefore, the ESTree AST generated by the parser must be transformed to Javascript before it gets executed. For this a third-party component named Escodegen8[7] is used. The visualisation that shows the interactive AST for the code given in the testbed is done with the third-party library named D3[8]. Finally, all the text editors used in this project are implemented using the third-party library called Ace10[9].

8 EVALUATION METHODOLOGY

The evaluation component was a two-fold process. The first part involved three software engineers that were presented the tool and after a short familiarisation session they were given the task to develop grammars for simple mathematical expressions. There was also a second round of more challenging projects that took place involving the development of a significant portion of Java 7, scheme, lisp and logo. During the activities the participants were asked to reflect on the process and the usability of the tool using free text. A general qualitative outcome of this feedback is that the tool at that level is relatively easy to use, it speeds up the process, and it is effective in producing adequate results.

The second part of the evaluation was a workshop involving two other groups of stakeholders, ICT teachers and learning designers. A non-random sampling method was used for the selection of the participants as there was a specific criterion needed to be met. The participants had to be knowledgable in learning technologies, familiar with this area of expertise, and relatively skilled in IT. The participants selected were three learning designers and two ICT teachers. The workshop was delivered in three parts. Initially the participants were given a general presentation of the tool. Then, they were then given a short presentation of the outputs generated by the developers in the first part of the evaluation. Having a first-hand experience of tangible outputs from that part the participants were finally asked to ideate on potential use cases of the tool. This was a focus group discussion facilitated by a moderator. This discussion generated very interesting outcomes and prospects of future work presented in the following section.

9 SUGGESTED USE CASES

The feedback received by the software engineers that did the usability testing as well as the workshop in the previous section gave us ideas and suggestions for potential use cases. The main type of users/stakeholders identified were software developers, learning designers, ICT Teachers, programming teachers and compilers teachers. Starting from the latter, the most conspicuous use of the tool would be as an assistive technology in University courses on compilers. It would also benefit programming education by enhancing web-based IDEs (Integrated Development Environments) with the ability to offer multi-language options. Additionally programming teachers could benefit by designing specialised languages for smoothening the learning curve on algorithmic thinking. These could be more high-level languages given as an intermediate step to make learning easier for beginners and less skilled developers. Another idea is to promote self-regulated learning to students learning a new language. Students knowledgable in one language could verify the correctness of their code by writing the same code in the language they know well and compare the transpilation outpouts to the new language. In this case the technology could be used as a meta-cognitive tool.

A different area of interest is web-based learning environments. In this category we have environments that natively support a language possibly exposed through a GUI component (Malt+) and environments that are not designed to teach programming but offer an API in JavaScript such as Geogebra (Carvalho et al., 2021). Geogebra is an application designed to teach geometry, algebra, statistics and calculus through a dynamic and interactive user interface. If an environment encapsulates concepts that may be difficult to express with the language na-

[4]https://github.com/NV/objectDiff.js
[5]https://github.com/estree/estree
[6]http://raphaeljs.com/
[7]https://github.com/estools/escodegen
[8]https://d3js.org/
[9]https://ace.c9.io/

tively supported, then it becomes apparent that the option to use a more expressive language would enhance the usefulness and thus the educational value of it. That combined with the fact that there would be no cognitive overhead for existing learners makes the idea even more appealing. Equally appealing is the idea to reuse learning environments like Geogebra in different ways and offer even programming education through them. That would increase significantly the usefulness and value of those environments as it would offer the opportunity to educators to take advantage of the strengths of different languages to convey different concepts using well tested and familiar to students interfaces. Respectively, it would also increase reusability of those environments allowing access to different communities of people and thus increasing the educational value of existing investments. Combining the strengths of different languages with different conceptual abstractions like objects, emotions, metaphors, and abstract processes supported by different environments is like using the best of both worlds to synthesise and convey compelling learning scenarios to prospective learners.

10 CONCLUSIONS AND FUTURE WORK

This paper presents an authoring environment designed to assist learning designers and developers to define grammars for new and existing languages for the web. This work took place in the context of the EU funded project ExtenDT2[10] and the development took place on the AWS (Amazon Web Services) platform. The benefit of using this tool is to optimise the process of defining input languages and make the output easily transformed into JavaScript. The intention of this work is to allow communication with web functionality in any language and thus take advantage of language-specific features combined with existing functionality to enhance learning and operational benefits. Preliminary usability testing gave us positive feedback. Furthermore, focus group discussions gave us directions on potential future uses. We envisage in future iterations to develop the tool further and combine it with many different learning or other JavaScript components in order to exploit its full potential and benefit. We also intend to carry out more elaborate evaluation cycles to assist with design improvements and the ergonomics of the interface.

[10]https://extendt2.eu/

REFERENCES

Azad, A. and Smith, D. T. (2014). Teaching an introductory programming language in a general education course. *Journal of Information Technology Education. Innovations in Practice*, 13:57.

Balat, V., Vouillon, J., and Yakobowski, B. (2009). Experience report: Ocsigen, a web programming framework. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 311–316.

Becker, B. A. and Quille, K. (2019). 50 years of cs1 at sigcse: A review of the evolution of introductory programming education research. In *Proceedings of the 50th acm technical symposium on computer science education*, pages 338–344.

Becker, B. W. (2001). Teaching cs1 with karel the robot in java. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 50–54.

Bergin, J., Roberts, J., Pattis, R., and Stehlik, M. (1996). *Karel++ A gentle introduction to the art of object-oriented programming*. John Wiley & Sons, Inc.

Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. (2005). A gentle introduction to the art of object-oriented programming in java. cafepress. com. *Google Scholar Google Scholar Digital Library Digital Library*.

Bouilliez, P. (2014). Glass cat–a tool for interactive visualization of the execution of oz programs in the pythia platform. *Université Catholique de Louvain*.

Canou, B., Chailloux, E., Vouillon, J., et al. (2012). How to run your favorite language in web browsers. *WWW2012 dev track proceedings*.

Carvalho, C. V. D. A., De Medeiros, L. G. F., De Medeiros, A. P. M., and Santos, R. M. (2021). Papert's microworld and geogebra: A proposal for enhancing functional teaching. *Modern Perspectives in Language, Literature and Education Vol. 5*, pages 1–12.

Cooper, S., Dann, W., and Pausch, R. (2000). Alice: a 3-d tool for introductory programming concepts. *Journal of computing sciences in colleges*, 15(5):107–116.

Duke, R., Salzman, E., Burmeister, J., Poon, J., and Murray, L. (2000). Teaching programming to beginners-choosing the language is just the first step. In *Proceedings of the Australasian conference on Computing education*, pages 79–86.

Ezenwoye, O. (2018). What language?-the choice of an introductory programming language. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122.

Goosen, L. (2008). A brief history of choosing first programming languages. In *IFIP International Conference on the History of Computing*, pages 167–170. Springer.

Hannafin, M., Land, S., and Oliver, K. (1999). Open learning environments: Foundations, methods, and models. *Instructional-design theories and models: A new paradigm of instructional theory*, 2:115–140.

Henriksen, P. and Kölling, M. (2004). Greenfoot: combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 73–82.

Irimia, A. (2001). Enhancing the introductory computer science curriculum: C++ or java? *Journal of Computing Sciences in Colleges*, 17(2):159–166.

Kahn, K. (1996). Toontalktm—an animated programming environment for children. *Journal of Visual Languages & Computing*, 7(2):197–217.

Kölling, M. and Henriksen, P. (2005). Game programming in introductory courses with direct state manipulation. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 59–63.

Kynigos, C. and Latsi, M. (2007). Turtle's navigation and manipulation of geometrical figures constructed by variable processes in a 3d simulated space. *Informatics Educ.*, 6(2):359–372.

Laakso, M.-J., Kaila, E., Rajala, T., and Salakoski, T. (2008). Define and visualize your first programming language. In *2008 Eighth IEEE International Conference on Advanced Learning Technologies*, pages 324–326. IEEE.

Lewis, M. C., Blank, D., Bruce, K., and Osera, P.-M. (2016). Uncommon teaching languages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 492–493.

Lindoo, E. (2020). Results of using a multi-programming language approach to decrease drop-fail rates in cs1. *Journal of Computing Sciences in Colleges*, 36(2):31–41.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., and Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371.

Mannila, L. and de Raadt, M. (2006). An objective comparison of languages for teaching introductory programming. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 32–37.

Matsumura, T. and Kuramitsu, K. (2016). A declarative extension of parsing expression grammars for recognizing most programming languages. *Journal of Information Processing*, 24(2):256–264.

Minor, J. T. and Gewali, L. P. (2004). Pedagogical issues in programming languages. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, volume 1, pages 562–565. IEEE.

Naveed, S., Sarim, M., and Nadeem, A. (2018). C in cs1: Snags and viable solution. *Mehran University Research Journal of Engineering & Technology*, 37(1):1–14.

Papert, S. (1980). Mindstonns vol. 607. *New York: Basic Rooks*.

Parker, K. R., Chao, J. T., Ottaway, T. A., and Chang, J. (2006). A formal language selection process for introductory programming courses. *Journal of Information Technology Education: Research*, 5(1):133–151.

PATTIS, R. (1981). Karel the robot john wiley & sons. *New York*.

Serrano, M., Gallesio, E., and Loitsch, F. (2006). Hop: a language for programming the web 2. 0. In *OOPSLA companion*, pages 975–985.

Smith, C. and Rickman, J. (1976). Selecting languages for pedagogical tools in the computer science curriculum. *ACM SIGCSE Bulletin*, 8(3):39–47.

Sobral, S. R. (2019). Cs1: C, java or python? tips for a conscious choice.

Sobral, S. R. (2020). The first programming language and freshman year in computer science: characterization and tips for better decision making. In *World Conference on Information Systems and Technologies*, pages 162–174. Springer.

Sobral, S. R. (2021). The old question: which programming language should we choose to teach to program? In *International Conference on Advances in Digital Science*, pages 351–364. Springer.

Solomon, C. J. and Papert, S. (1976). A case study of a young child doing turtle graphics in logo. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 1049–1056.

Tharp, A. L. (1982). Selecting the "right" programming language. *ACM SIGCSE Bulletin*, 14(1):151–155.

Van Haaster, K. and Hagan, D. (2004). Teaching and learning with bluej: an evaluation of a pedagogical tool. *Issues in Informing Science & Information Technology*, 1.

Wainer, J. and Xavier, E. C. (2018). A controlled experiment on python vs c for an introductory programming course: Students' outcomes. *ACM Transactions on Computing Education (TOCE)*, 18(3):1–16.

Wen, E., Warren, J., and Weber, G. (2020). Browservm: Running unmodified operating systems and applications in browsers. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 473–480. IEEE.

Wexelblat, R. L. (1979). First programming language: Consequences (panel discussion). In *ACM Annual Conference*, page 259.

Zschaler, S., Demuth, B., and Schmitz, L. (2014). Salespoint: A java framework for teaching object-oriented software development. *Science of Computer Programming*, 79:189–203.