SPLASH: G: Modular Educational Languages

Jesse Hoobergs
jesse.hoobergs@kuleuven.be, Department of Computer Science, KU Leuven

ABSTRACT

Effectively teaching programming to a general audience is challenging. Part of this challenge is caused by the friction between the programming language and pedagogical approach. We provide a new theoretical foundation that expresses this friction in terms of two novel concepts: the *program behavior gap* and *programming language misalignment*. We show preliminary results of an empirical study into the program behavior gap, and discuss the design of a system to build educational programming languages that reduce the gap. Our system uses a modular approach to allow teachers to compose programming languages specifically tailored to their pedagogical approach. We show the results of two case studies.

1 RESEARCH PROBLEM AND MOTIVATION

Programming is a skill that is becoming increasingly important in industry and various scientific domains, but many students struggle to acquire this skill [8, 22, 29]. This contrast has led to a global shortage of programmers [1, 3]. To address this problem, programming education is being introduced in secondary education around the world [4, 12]. However, teachers have difficulty in guiding an entire class in learning to program. This problem can stem from (1) the teacher missing the right expertise due to (qualified) teacher shortages, but (2) even teachers with the right expertise can struggle due to large class sizes and the nature of programming. The high failure and dropout rates in CS1 courses can attest to this second point [5].

The difficulties in learning to program are multifold. In this project, we look at (1) how the programming language hampers **learning**, (2) how the programming language hampers **teaching** and, (3) how we can remove as much programming-language-induced friction as possible. We use the wording *the* programming language and not *a* programming language to make it clear that for any chosen programming language, these issues exist.

Every programming course uses — explicitly or implicitly — a particular pedagogical approach. Building on the work of Duran et al. [11] we argue that a pedagogical approach entails at least four elements. The first element entails which programming language concepts students should learn about. The second element specifies what students should know about these concepts, or in other words, the mental model that should be formed about these concepts. The third element pertains to when each of these concepts and mental models should be formed. It is reflected by the order of concept introduction in the course. We would like to note that the targeted mental model for a programming language concept can change through time, as knowledge expands throughout the course. The fourth element is motivated by the context of the course and guides the development of the exercises and examples used in the course.

We argue that, regardless of which programming language is used, the programming language will hamper this pedagogical approach. The reason for this is a concept that we call the *Programming Language Misalignment*. A PL misalignment occurs when the language does not allow implementing the pedagogical approach exactly as wanted. It is often reflected by the fact that the teacher needs to introduce other concepts of the programming language to be able to introduce the concepts they want to introduce. These other concepts can be concepts that the teachers would like to introduce later, but can just as well be concepts that the teacher does not want to introduce at all.

One example of a PL misalignment is the *importing*-misalignment which occurs when the teacher wants the students to write programs using programming language features that can only be accessed after importing them from the standard library (e.g. mathematical functions) or other libraries (e.g. image library). The language imposes on the teacher to also introduce the concept of importing. A second example of a PL misalignment (the *floating-point*-misalignment) occurs when the teacher introduces floating-point values to express decimals. The programming language (often) imposes the teacher to discuss floating-point representations to explain why the result of `0.1 + 0.2` is `0.30000000000000004` instead of `0.3`.

PL misalignments can be handled in a couple of ways: (1) introduce the needed additional concepts together with the wanted concepts, (2) change the order of concept introduction, and (3) neglect the misalignment by not explaining the additional concepts. The first option will be a valid option in some cases, but will often make the cognitive load too high. The second option is a good idea, if it does not lead to a pedagogical approach that one would never consider if not imposed by the language. The third option is arguably the most used option, as one does not need to change their approach. The choice for this option is often reflected by teachers telling students to write *magic incantations* like `from math import sqrt`.

This third option leads to what we call the *Program Behavior Gap* as there now exists a gap between what a student should understand — the targeted mental model — and the behavior of the program when executed by the programming language. One example is the *error-messages*-gap which occurs when an error message uses concepts not yet known to the student. As determining whether they should be able to understand this error message or not is impossible for students, they will need to ask the teacher for help. Another example is the *expressions-as-statements*-gap which occurs when a student accidentally writes an expression instead of a statement. When a student writes `x == 20` instead of `x = 20` they might not be able to find their mistake. As the targeted model of program behavior does not entail using expressions as statements, students would expect to get a runtime error from `x == 20`.

The PL misalignment and PB Gap concepts form the theoretical foundation for our *Programming Education Runtime System* (PERS) amPERSand. This system makes it possible to combine programming constructs and features to create modular educational programming languages without PB gaps and PL misalignments for a particular pedagogical approach. These languages are modular as features can be combined to compose a programming language. The amPERSand system generates an interpreter for the designed education programming language that supports step-by-step and in-browser — through WebAssembly [19] — execution.

The next section discusses related work on computing education theory and educational programming languages. The third section reports on an empirical study into the PB gap, while the fourth section discusses the PERS. The final section presents the results of two case studies and future work.

2 BACKGROUND AND RELATED WORK

Computing education research

When a novice learns to program, they form a mental model of the programming language. One might wonder when a mental model is the correct one — the target model – for a novice to have. The semantics of a programming language constitute the ground truth of how programs behave in the language. Unfortunately, semantics are intricate and interdependent, which makes it infeasible to use them as target model. Duran et al. [11] call the target model for the mental model, the *model of program behavior*.

The reader might be familiar with the concept of a notional machine [9, 23, 15, 18] and wonder how it relates to the model of program behavior. Dickson et al. mention that a notional machine "can be conceived of as the ideal mental model we would like students to have (for present requirements)", which is exactly our definition of a model of program behavior. However, as Duran et al. [11] show that the term *Notional machine* has been used in literature with many definitions, some of which are incompatible with each other, we use the term *model of program behavior* (MPB). The right side of Figure 1 shows the relationship between a mental model, an MPB and misconceptions.

The mental model that students should have — the MPB — changes throughout the learning process. This means that a pedagogical approach does not entail one MPB, but a **progression** of MPBs [11]. In week 1 one might want students to have a particular conception about a concept — or no conception of it at all — and in week 3 one might want them to have a different, more elaborate conception about that concept. Figure 1 shows the relation between the mental model and the MPB throughout a progression. For each part of the progression, the MPB becomes larger and hopefully the mental model of the student becomes larger as well. We would like to note, that something that is a misconception in one part of the progression, might be a valid conception in another part. This is illustrated in Figure 1 with the position of the black dot in week 1 and week 3.

Misconceptions and our new PB gap concept are distinct. Figure 2 shows both a general overview and

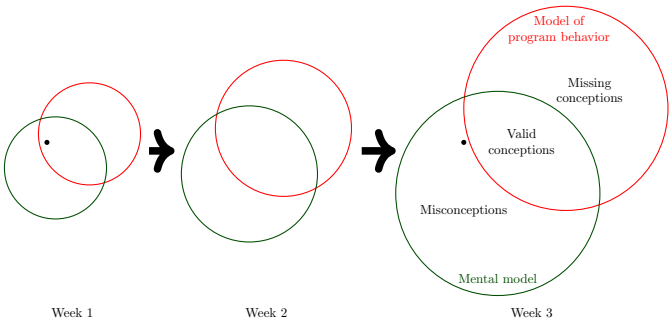

Figure 1: Overview of the relationship between a mental model and a model of program behavior throughout a progression. The dot symbolizes a conception that was valid in week 1, but became a misconception in week 3.

a more realistic overview of the relationship between programming language semantics, models of program behavior and mental models. The general overview highlights seven different areas. There are four distinct areas labelled as PB gap, two in the blue circle and two in the red circle. The first type of PB gap – in the blue circle – occurs when some parts of the language are not yet (fully) explained. This relates highly with the *coverage* evaluation criteria of the RPB framework of Duran et al. [11] who state that "[a]n RPB ruleset may be viable for only some of the programs that can be written in the learners' programming language. The more programs it is viable for, the higher its coverage." and that it is related to completeness in semantics. The second type of PB gap – in the red circle – occurs when parts of the model of program behavior are conflicting with the language semantics. This related highly with the *accuracy* evaluation criteria of the RPB framework of Duran et al. [11] who state that "[a]n RPB ruleset is accurate if what the rules say about the system effectively happens and the rules thus lead to correct predictions of the system's behavior." and that it is related to soundness in semantics.

The more realistic overview given in Figure 2 uses the size of the circles to highlight that an MPB often only explains a limited part of the programming language. The MPB is now placed almost completely within the programming language, as most rules are sound with respect to the semantics of the programming language.

Educational programming languages

The origin of educational programming languages can be traced back to the LOGO programming language [26]. Later, Pascal [31] came to the fore as an educational programming language and grew even further as an industrial programming language. Contemporary educational (text-based) programming languages are, for example, Grace [6], Hedy [20], Pyret [2], MuLE [10] and Quorum [30].

Looking at these programming languages from our theoretical framework, we see that they often try to reduce PL misalignments. Hedy [20], for example, has built-in turtle graphics commands to counter the *importing*-misalignment and has localized syntax for over 50 natural languages to counter the *english-keywords*-misalignment

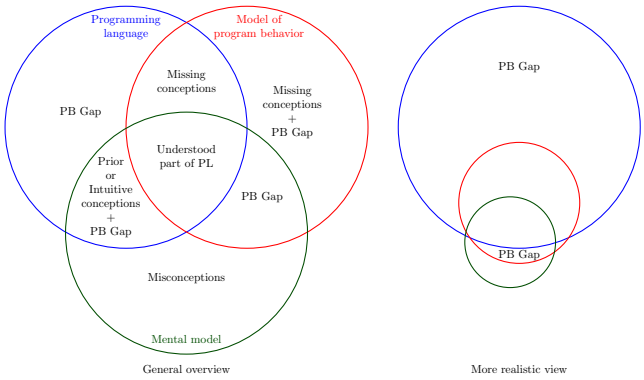

Figure 2: Overview of the relationship between programming language semantics, the model of program behavior and misconceptions

for non-native English-speaking novices.

A related type of languages are **sublanguages** like MiniJava [28] which are a subset of another programming language — in this case Java. As these sublanguages have fewer features, they will have fewer PB gaps. They might, however, have more PL misalignments for particular pedagogical approaches.

The sublanguage approach can also be used for progressions. Hedy is a *gradual* language consisting of 18 languages levels that gradually build up to a subset of Python where each level can be seen as a sublanguage of the next. Each of these levels closely matches a MPB in a progression of MPBs, resulting in few PB gaps throughout the progression. The syntax also evolves gradually throughout the levels, leading to fewer PL misalignments for their pedagogical approaches. A second example of such gradually changing languages with few PB gaps throughout the progression, are the teaching languages as used in *How to design programs* [13].

Unfortunately, almost all existing educational programming languages are monolithic programs. It is nearly impossible to adapt them ever so slightly. Each of these languages thus works for very specific pedagogical approaches, as they are **not adaptable to other approaches**. As an example, changing the level in which a certain feature is introduced in Hedy [20], is impossible without a deep dive into the source code of the programming language.

We found two notable exceptions to this. The first one is the 'Programmable programming language' Racket [14]. The teaching languages from *How to design programs* [13] are created as languages within the Racket language, allowing for relatively easy adaptation. The second one is the modular system of Cazzola and Olivares [7] which was used to create different languages to gradually learn JavaScript and allows changing the order of concept introduction.

3 EMPIRICAL STUDY OF PB GAPS

To investigate the idea of the PB gap, we conducted a study to answer the following research questions: (1) When students trace code with a PB gap, which result do they predict? (2) When students trace code with a PB gap, do they notice the gap? (3) Does the answer to the previous two questions differ for syntactic and non-syntactic PB gaps?

PB gaps are 'syntactic' if they make use of syntactic elements that are unknown to the novice. These gaps are visible in the source code, in contrast to non-syntactic gaps.

The preliminary results indicate that many students don't notice the PB gaps. As an example, we asked students to trace code in which the precedence of the logical operator `and` over `or` mattered. This precedence presented a non-syntactic PB gap as the precedence rules were not mentioned in class, while all syntactic elements were known to the students. Less than 40% of the students, noticed that the precedence rules were important for this problem, and most students evaluated the expressions with `and` and `or` from left to right.

For syntactic PB gaps, we noticed many students expressing that they did not know how to trace the code, due to the unknown syntactic element. We also noticed that this amount dropped significantly once these syntactic elements were introduced in class. However, most students then showed misconceptions.

As expected, students almost always predict the result that corresponds to the model of program behavior. The code in listing 1 shows the classical *aliasing* problem. We showed this code to students when they did not yet know about lists and references, and asked them what the value of `list1` was at the end of the code. They did know about indexing in the context of strings, but as strings in Python are immutable, references don't matter for them. The reference semantics of list thus give rise to a non-syntactic PB gap.

For this exercise, 14 out of 16 students answered according to the model of program behavior — assignment copies the value into a new variable — and mentioned that the "change to `list2` does not matter for the result of `list1`". Without the PB gap concept, one would say that all these students have a misconception as their mental model does not correspond to the real (reference) semantics. With the PB gap concept, one would say that 14 students have a model corresponding to the model of program behavior, 2 students have a model corresponding to the semantics of the language, and no students have a misconception.

```
list1 = [10, 20, 30]
list2 = list1
list2[1] = 77
list1[2] = 55
```

Listing 1: Code containing a PB gap due to reference semantics.

4 APPROACH AND UNIQUENESS

In this project, we use our PL misalignment and PB gap concepts as a theoretical foundation for a system that can be used by instructional designers to create a progression of educational programming languages which exactly matches the progression of MPBs in their pedagogical approach.

Our system is designed from an education-first perspective: all languages created with the system are meant to

be used in education and the system should be **useable by teachers** to create custom languages for their courses. In this regard, the system differs substantially from Racket [14], as Racket allows creating teaching languages just because they are programming languages. Creating a teaching language in Racket is not feasible for most teachers.

Composing languages

Figure 3 shows the composition of the languages `LangWeek1<E, S>` and `LangWeek2<E, S>` where the carriers `E` and `S` represent expressions and statements. The language for the first week is composed of features for variable assignment, variable reference, printing and literal numbers. The language for the second week contains all features of `LangWeek1` together with features for literal lists and list indexing.

Teachers thus start creating a language from the ground up by selecting the wanted (predefined) features for their languages. The selection of each feature consists of the following steps: (1) selecting the feature (2) choosing one of the predefined semantics for this feature (3) writing error messages for the possible runtime errors in the chosen semantics. (4) choosing one of the predefined syntaxes for this feature. In the fourth step, the teacher can make some choices like the keywords and symbols used. It is also possible to automatically generate definitions for Blockly [16] blocks to create a block-based language.

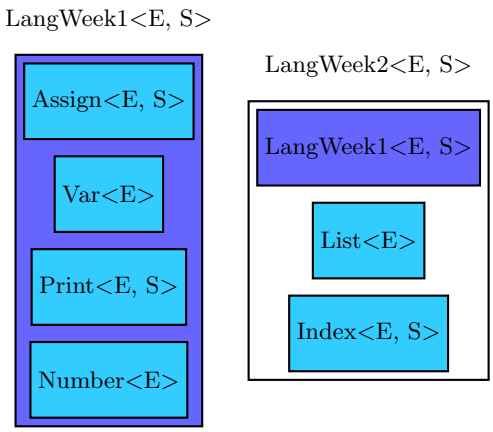

Figure 3: Specification of two languages

Implementation

To modularly compose the languages, the PERS uses object algebras [24]. The modularity of object algebras fits our purpose, but comes at a price: reduced efficiency [32] and boilerplate code [25]. Due to our education-first approach, efficiency is not a priority. The amount of boilerplate code is reduced by the use of the Rust programming language and its procedural macros.

As step-by-step execution is useful for educational programming languages, the used carriers of the object algebras implement a `step` method that utilizes the *trampolining* technique [17] to execute one step at a time. Each step evaluation returns the current evaluation result and the continuation for the next step to resume computation. On top of this, the system uses *algebraic effects and handlers* [27] to give back control to the controlling platform (e.g. a web platform) when an effect like reading input or drawing turtle graphics has to be fulfilled. The effect contains specific data for the effect (e.g. the length of the turtle movement) and a continuation to resume execution.

Parsing modular languages poses a significant challenge. We have chosen to use parser combinators as they have similar compositional properties as object algebras. For each feature, different possible parsers from which a teacher can choose, are implemented. Whenever a carrier (expression or statement) needs to be parsed, the selected parsers for all object algebras returning that carrier are tried to parse the expression. Doing this naively, leads to different parse results depending on the order of feature selection, which is unwanted. We therefore use a parse forest approach in which all possible object algebras are tried when parsing a carrier — instead of stopping when one parsing succeeds. This also allows us to let the parser of each feature give concrete feedback when parsing fails.

As we do want one parse tree instead of a forest, the teacher will need to specify how to filter the trees in the forest until one tree remains. They can do this in the form of rules stating that a node of feature X (e.g. addition) can never be contained within a node of feature Y (e.g. multiplication). As stating these rules is cumbersome, many of these rules could be generated from precedence-rules specified by the teacher, which are easier to determine.

5 RESULTS AND CONTRIBUTIONS

Case Study: Hedy

As a case study, the Hedy [21] programming language has been implemented with this PERS. Each Hedy language level was implemented as a language within the PERS by combining and reusing the needed features. As Hedy is a localized language, there actually exist more than 50 different *Hedy*s, one for each supported natural language. Each of these *Hedy*s has the same features and semantics at each level, but a different syntax and parsing. For this case study, the English and Dutch Hedy were implemented.

This reimplementation yields many benefits: (1) fewer PB gaps as the compilation to Python might accidentally allow unwanted features, (2) algebraic effects make the interaction with the Hedy platform explicit, (3) automatic debugging support without the need for source maps, (4) many object algebras are reused, (5) localization of error messages is easier, and (6) no need for a backend server as the parsing works in the browser.

Case Study: Python with Turtle graphics

In another research project we wanted to see whether debugging turtle graphics problems can be aided by allowing the user to hover over a drawn line to see which statement in the source code drew this line. Our PERS served as a good starting point to implement this tool, as each turtle command (draw, change color, turn...) leads to an effect that we can easily capture to keep track of which statement yielded which line. As the students already knew Python, we reimplemented a subset of Python with the PERS. This shows that the system can also be used to create educational versions of industrial languages.

Future Work

The empirical qualitative study on the PB gap concept yielded interesting results. We will do a similar study on the PL misalignment where we interview teachers to see whether they experience misalignments and how they deal with them.

One of the key limitations of the PERS is that it currently does not have a graphical user interface to design the modular languages. For now, the languages are described by using a set of custom Rust procedural macros to specify the composition of features and the linking of features with semantics and syntax. As a next step, a UI will be created which guides the teachers through the process of creating (a progression of) programming languages. The system with UI will then be tested with teachers.

ACKNOWLEDGMENTS

This research is partly funded by *KU Leuven Industrial Research Fund* (project 3E230056).

REFERENCES

[1] [n. d.]. Eurostat: Statistics explained. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=ICT_specialists_-_statistics_on_hard-to-fill_vacancies_in_enterprises. Accessed: 2023-02-07.

[2] [n. d.]. The Pyret Code; or A Rationale for the Pyret Programming Language. https://www.pyret.org/pyret-code/

[3] [n. d.]. Software developers, Quality Assurance Analysts, and testers : Occupational outlook handbook. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. Accessed: 2023-02-07.

[4] Erik Barendsen, Nataša Grgurina, and Jos Tolboom. 2016. A new informatics curriculum for secondary education in the Netherlands. In *ISSEP 2016*.

[5] Jens Bennedsen and Michael E Caspersen. 2019. Failure rates in introductory programming: 12 years later. *ACM inroads* 10, 2 (2019), 30–36.

[6] Andrew P Black et al. 2013. Seeking Grace: a new object-oriented language for novices. In *SIGCSE 2013*. 129–134.

[7] Walter Cazzola and Diego Mathias Olivares. 2015. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (2015), 404–415.

[8] Chin Soon Cheah. 2020. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology* (2020).

[9] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies* (1981). https://doi.org/10.1016/S0020-7373(81)80056-9

[10] Nikita Dümmel, Bernhard Westfechtel, and Matthias Ehmann. 2020. MuLE–a Multiparadigm Language for Education-The Procedural Sublanguage. In *EDUCON 2020*. IEEE, 392–401.

[11] Rodrigo Duran, Juha Sorva, and Otto Seppälä. 2021. Rules of Program Behavior. *ACM Trans. Comput. Educ.* 21, 4 (2021), 33:1–33:37. https://doi.org/10.1145/3469128

[12] Katrina Falkner, Sue Sentance, et al. 2019. An international comparison of k-12 computer science education intended and enacted curricula. In *Koli calling 2019*. 1–10.

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

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (2018), 62–71.

[15] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention *(ITiCSE-WGR '20)*. ACM, 21–50. https://doi.org/10.1145/3437800.3439202

[16] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE blocks and beyond workshop*. IEEE, 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000

[17] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. 1999. Trampolined style. In *ICFP99*. 18–27. https://doi.org/10.1145/317636.317779

[18] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019), 1–23. https://doi.org/10.4230/DAGREP.9.7.1

[19] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI 2017*. 185–200. https://doi.org/10.1145/3062341.3062363

[20] Felienne Hermans. 2020. Hedy: a gradual language for programming education. In *ICER 2020*.

[21] Felienne Hermans. 2020. Hedy: a gradual language for programming education. In *ICER 2020*. 259–270. https://doi.org/10.1145/3372782.3406262

[22] Tony Jenkins. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*.

[23] Shriram Krishnamurthi and Kathi Fisler. 2019. 13 Programming Paradigms and Beyond. *The Cambridge handbook of computing education research* (2019).

[24] Bruno C d S Oliveira and William R Cook. 2012. Extensibility for the masses: Practical extensibility with object algebras. In *ECOOP*. Springer, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

[25] Bruno C d S Oliveira, Tijs Van Der Storm, Alex Loh, and William R Cook. 2013. Feature-oriented programming with object algebras. In *ECOOP 2013*. Springer, 27–51. https://doi.org/10.1007/978-3-642-39038-8_2

[26] Seymour A Papert. [n. d.]. *Mindstorms: Children, computers, and powerful ideas*.

[27] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, ESOP 2009*. Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[28] Eric Roberts. 2001. An Overview of MiniJava. *SIGCSE Bull.* (feb 2001).

[29] Anthony V Robins. 2019. 12 Novice Programmers and Introductory Programming. *The Cambridge handbook of computing education research* (2019).

[30] Andreas Stefik and Richard Ladner. [n. d.]. The Quorum Programming Language (Abstract Only) *(SIGCSE '17)*.

[31] Niklaus Wirth. 1971. The programming language Pascal. *Acta informatica* 1 (1971), 35–63.

[32] Haoyuan Zhang, Huang Li, and Bruno C d S Oliveira. 2017. Type-safe modular parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 2–13. https://doi.org/10.1145/3136014.3136016