Gradually Learning Programming Supported by a Growable Programming Language

Walter Cazzola and Diego Mathias Olivares

***Abstract*—Learning programming is a difficult task. The learning process is particularly disorienting when you are approaching programming for the first time. As a student you are exposed to several new concepts (control flow, variable, etc. but also coding, compiling etc.) and new ways to think (algorithms). Teachers try to expose the students gradually to the new concepts by presenting them one by one but the tools at student's disposal do not help: they provide support, suggestion and documentation for the full programming language of choice hampering the teacher's efforts. On the other side, students need to learn real languages and not didactic languages. In this work we propose an approach to gradually teaching programming supported by a programming language that grows—together with its implementation—along with the number of concepts presented to the students. The proposed approach can be applied to the teaching of any programming language and some experiments with Javascript are reported.**

***Index Terms*—Teaching of programming, gradual learning, modular development of programming languages, modularity.**

I. Introduction

Learning programming is a difficult task. It is widely accepted that to turn a novice into an expert it takes more or less ten years [1], [2]. Some studies [3]–[6] have been done about why to learn programming is so complex and basically it has turned out that the chosen programming language and the exposure to the single programming concept are part of the problem. Schneider [7] pointed out ten principles—still up-to-date—about teaching programming that can be summarized in: *learning programming is not learning a programming language* and *students must learn real programming languages and tools*. The survey reported in [8] and the TIOBE index[1] show that imperative programming languages such as Java, C and C++ are the most popular both in enterprise and in educational sectors. While their potential and versatility is mostly undisputed, their innate complexity and the complexity of their tools could risk to divert students' attention from the key point of learning appropriate programming techniques in general. Students that approach learning programming tend to focus on learning the programming language since it looks more practicable and it needs a certain and—in students eyes—quantifiable amount of time to learn how to use it, its syntax and its tools. The prominent issue lies within the core components of these languages: each programming language consists of many different features—such as control flow, variable definition, functions, and so on—, each bound to a different construct or group of constructs of the language. Due to their strict intertwining, the students are forced to keep up with a considerable quantity of unknown constructs and concepts since the beginning. This issue aggravates if we consider that many of these constructs are not meant to be fully explained until much further in the course, or worse, never. Let us consider the Java version of the `HelloWorld` program:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

This code snippet is typically used in the very first lesson to explain students a couple of basic concepts—such as strings and standard output—but they are also accidentally exposed to many more concepts that cannot be hidden: class declaration, methods (declaration and use), access modifiers, packages, static members, arrays, arguments and types. Typically, the teacher skips such concepts asking the students to learn them by heart promising a proper explanation in due time. Such an approach contributes to create a sense of incompleteness and disorientation and the students get the whole picture and have a full comprehension of what they do only at the end. Look at [9] and [10] to have a full comprehension of the problem of anticipating advanced concepts at the beginning of the learning. Also support tools, as IDEs, do not help the teacher in the compartmentalization of the programming concepts [11] since they provide the students with code skeletons to fill—e.g., look at the effect of the Eclipse's «`new`↪`class`» menu entry—without considering any language restriction.

Bruce *et al.* [12] presented a phenomenographic study about how first-year university students perceive the first programming course experience. As a study result, they categorized the replies upon what the students focus on and expect from an introductory programming course. From Bruce *et al.*'s study it is fairly evident that *the more a student refines his idea about what being a programmer and solving problems implies, the less he recognizes the language syntax as a fundamental basis for the course.* The need for a precise syntactic pattern to smooth the learning curve is directly proportional to the desire of just "passing" the course without any genuine interest in it. On the opposite a student aware and committed to his role as a programmer and confident in the rest of the programming community manages to see a language as a mere tool to solve problems.

Teaching basic and advanced programming courses taught

This work was partially supported by the MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society

Walter Cazzola is with the Computer Science Department at Università degli Studi di Milano. He can be contacted at `cazzola@di.unimi.it`.

[1]www.tiobe.com

us that students tend to confuse the *learning of programming* with the *learning of the programming language*. The students try to master the programming language by learning any tiny little detail of it; in the hope that such details hide an easy solution to the proposed assignments: *to master the language to rule over all the problems*. Evidently, this approach promotes the wrong belief that to learn programming means to learn the programming language instead of the correct thinking that *programming is a matter of being able to solve problems independently of the used programming language*. As Bret Victor [13] said «*Learning about "for" loops is not learning to program, any more than learning about pencils is learning to draw.*»

Problem solving is typically taught by examples [14] and this approach seconds the misbelief the students have about what programming is. The solution to a problem is often presented as an *already cooked* algorithm written in the programming language of choice. Hardly the teacher has the time during his/her lesson to present the *mind process* to get to such a solution, since he/she is already busy in describing the syntactic constructs used in the program, what they do, and all the possible variants the language proposes. Students that still have to tune up their programming skills are disoriented by this approach: they understand what the program does but they cannot either cook their own solution or adapt the proposed solution to a different context (see [4] for a study that supports this observation). More pre-cooked solutions they get and more difficult for them becomes to separate the programming language from the problem solving. At the very end, students rarely get acquainted to the full-extent of the programming language they use, and they feel disoriented when facing problems that slightly divert from those presented in the lectures. Even smarter students maintain a solving approach based upon few constructs that they master at the best, which often is not the most effective approach: it is really hard to eradicate bad habits when students are focused on learning the language rather than the problem to solve. The first programming language learned marks the programmer indelibly [15] and this also applies to the language constructs the programmer learns first. This is a more evident side-effect when the learner is exposed to the whole language since the beginning and without any constraints on what construct to use in solving a given problem. As Mason and Cooper [16] found out through their experiment most of the problems students have with learning how to program is bound to the *cognitive resources* the students put at disposal and the excessive *cognitive load* to learn a full programming language demands them.

Training students to problem solving means to let them fully explore what they have at their hands, sometimes addressing them to different solution strategies but if the tool is too complex, the exploration requires too much time that the students steal from thinking about the problem. Sometimes the only way to get students to experiment something is to force them in that direction by limiting the choice to a particular group of language constructs. Some extreme positions include to delay the coding and thus the learning of the language[2] and to substitute a programming language—rich of syntactic constraints—with a formalism independent of any syntatic aspect as either a modelling language as in [17] or pseudo-code as in [18]. In our opinion, this could help the students to focus on the problem solving aspect of programming but coding and all the other aspects related to it (e.g., compiling, debugging) are part of the learning process and cannot be avoided. However, we agree that if the students could have access to only a portion of the language and adequate support tools at any stage of their learning process they could focus more on the problem solving aspect and gradually master both the language and how to use it to solve problems. This view is perfectly in line with Kirschner *et al.* [19] experiment that reports a failure in teaching programming without a deep and gradual guidance as accidentally happens when the students are exposed to full language since the beginning. In the remainder of the paper we present our approach to teach how to program that adopts a programming language customizable by the teacher to his/her needs.

Paper outline. In Sect. II we describe a general idea and model about how programming can be gradually taught to students with the support of an incremental development framework. In Sect. III, we instantiate the proposed teaching model to use the Neverlang development framework to teach Javascript and summarize our first experience. Finally in Sect. IV and Sect. V we discuss some related work and draw our conclusion respectively.

II. Gradual Learning of Programming

Summarizing our experience as teachers and those reported in the literature (see Sect. I), we could conclude that the students have to:

- focus on problem solving;
- learn mainstream programming languages; and
- be gradually exposed to new concepts

in order to render more effective the learning of programming. Therefore the teaching strategy that accomplishes such requirements should be the one where the students are guided to learn how to program with a language through a gradual and tool supported disclosing of language features.

A. Gradual learning: The idea.

In other words, students should interact with a growing subset of the language along the whole course to master one concept at a time.

Basically, this is what every teacher does by compartmentalizing the arguments but this is normally hampered by support tools that consider the language as a whole [11]. To really turn on, this idea we need a (mainstream) programming language whose charateristics and implementation can modularly grow together with the number of concepts the students have already learned. Unfortunately, all the mainstream programming languages miss this last characteristic. Their compilers/interpreters are monolithic, the languages are designed as a whole,

[2]http://www.edutopia.org/blog/radically-transforming-teaching-programming-1-ajit-jaokar

and although students could be guided with some formal restrictions, the support tools ignore such restrictions. The development of *ad hoc* support tools as ProfessorJ [11] can support the teacher with language restrictions only as long as the students will not directly interact with the language interpreter/compiler. Consequently, the teacher should rely on a programming language whose implementation grows at the pace of what the students are learning and whose growth can be conveyed according to the teaching needs.

One possibility could be that the teacher partitions the language of choice in subsets according to the desired teaching order and then implements the compilers/interpreters corresponding to each subset. This solution subsumes that the teacher has a good knowledge about language grammars and compiler development, and this would also be a highly time-consuming task and a teacher could be disheartened by such a burden. In alternative, the teacher could exploit one of the frameworks that already supports the partitioning of the language to teach (see Sect. IV). The problem with such frameworks is their low degree of freedom: the proposed partitions are predefined and can rarely be customized by the teacher; both teacher and students are subdued to the framework's developers decisions about the right learning/teaching pace.

Nowadays however new instruments allow the language designer to build compilers and interpreters in a different and non-monolithic way. The concept of modularity has spread from basic software engineering to programming language development as well: with the introduction of tools like LISA [20], Silver/Copper [21], Spoofax [22], and Neverlang [23]–[25] the development of programming languages has taken a new turn towards language extensibility and reusability. As we will show, these tools can be used to develop a complete implementation of the language of choice, that can be restricted and extended following the teacher decision with little or no effort. This will smoothly support gradual learning and will enable the teacher to define his subdivision of the language at any time and towards any direction.

Clearly, a modular language design framework is not sufficient by itself to accomplish our idea. In the first place a complete modular implementation of the language of our choice is necessary. This will be the foundation for the whole compartmentalization used in the course; besides, the teacher should be enabled to easily navigate, select and compose the available set of language features to form the necessary support tools. Tools as FeatureHouse/FeatureIDE [26], [27] for Spoofax and AiDE [28]–[30] for Neverlang can support the teacher on such a task.

B. Gradual learning: the model.

In the proposed model, the teacher initially has to prepare the course by partitioning the programming language of choice in a set of usable sub-languages according to the programming concepts he wants to present and in which order. Such sub-languages must be sortable in a sequence where—apart from the first sub-language—every sub-language adds some programming concepts to the previous one in the sequence and the last one corresponds to the original language. Given such a sequence, the learning process passes through three reiterated steps (depicted in Fig. 1):

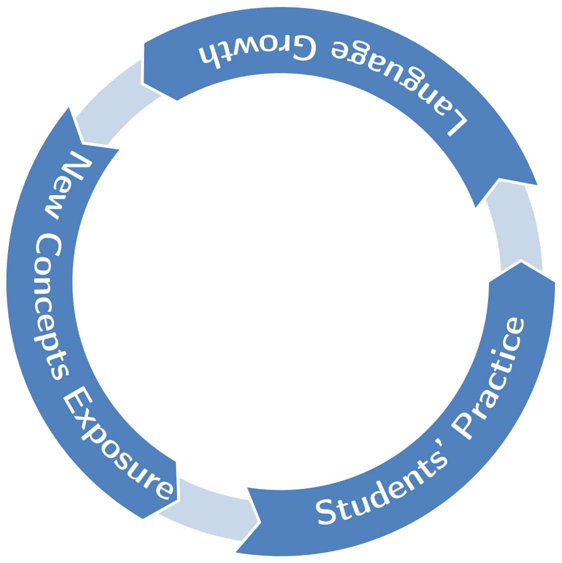


Fig. 1. The learning stage

1) language growth;
2) exposure to new concepts; and
3) students' practice

An iteration through these three steps is—what we call—a *learning stage* and it coincides with the teaching/learning of one sub-language. The three steps have to be reiterated until all the sub-languages have been taught or the desired language coverage is reached.

Language growth. This step corresponds to the work the teacher has to do to pass from one learning stage to the next in the sequence. As the language grows—that is, when the teacher decides it is time to present new programming concepts—the support tools have to grow as well—that is, the students must have a compiler/interpreter that supports exactly the programming concepts they should know at the end of that learning stage. For example, if the language used in the previous learning stage and the new one differ for the presence of the `for` loop the new compiler will be compliant to the one used by the students so far and it will also support the `for` loop. Technically speaking, the teacher decides which programming concepts wants to teach and to let the students experiment on; selects the modules corresponding to such programming concepts among those provided by the modular language design framework and composes them to form the needed compiler. Each learning stage relies on the tools prepared in the previous stage enriched only by the modules for the newly introduced concepts [31]. The selection and composition of the modules can be eased by using specific tools [27], [29]. Basically, the idea is that each new learning stage provides a number of new concepts that the students have to focus on. Thus, the approach is more effective as the number of new concepts to master is small and strictly related to each other. The new support tools are installed in the laboratories and passed to the students.

Exposure to new concepts. The teacher exposes the students to the new concepts. This is done through a batch of frontal lectures and some hands-on sessions where the students will experiment the new concepts on the development framework prepared in the previous step. The length of the exposure depends on the number of new concepts introduced, the estimated difficulty of the new concepts and the learning pace of the class. This last point affects the length of a single learning stage that could change from a course edition to another. The teacher can shorten or widen the learning stage acting on the number of hands-on sessions: if the students master the new concepts faster than expected, some hands-on sessions can be removed, otherwise, some should be introduced to help the students.

The gradual introduction of programming concepts helps the students to exclusively focus on the new concepts. This has the benefit that at any time any example the teacher shows to the students will only use those concepts that the students are acquainted with. Let us reconsider the `HelloWorld` example, presented in the introduction, and the number of issues it has when used in the first lecture on Java. With our approach these issues can be avoided by restricting the first Java's sub-language to only contain constant strings and the printing functionality, as in:

```
println("Hello world")
```

As said before, the programming concepts in a real programming language are intrinsically intertwined and the later introduction of a new concept can affect how the concepts already in the language behave. For example, a partition where the expression language on integers is a sub-language that is extended with strings and the juxtaposition operator (+). In this case, integers and strings can be used in the same expression—as in `"a"+0`—whose interpretation hides some extra concepts—as implicit type conversion. Thus, the frontal lessons should face the intertwining of the new concepts with those already learned and should show their interoperability within the language context. Particular emphasis should be put on how some language constructs evolve through these additions, and how the composition of constructs from different sources can increase the complexity and the variability of the language semantics. As with a standard course, each of these new topics should be contextualized within a proper set of examples, which will work as guidelines for the next steps; some examples can be used along the whole course to show how these can vary when new concepts are introduced.

Students' practice. The last step of a learning stage focuses on self-learning. Students cannot really master new programming concepts if their only exposure to them is through the guidance of the teacher: they have to autonomously work with the new concepts on some homework prepared by the teacher. In the traditional approach this activity represents a potential risk for information smuggling: students can look for help on different sources—as the Internet, friends and relatives—that could provide suggestions that pass over the compartmentalization imposed by the teacher. As seen in [4], to anticipate new concepts can disorient the students—they can apply the solution without understanding how and why it works—, the students will focus their attention on exploring the new concepts rather than mastering the concepts already at their disposal and last but not least the students will spend more time on the programming language than on the problem solving aspect [32]. It is important to notice that in every learning stage only a subset of the programming language is available and the code written for that subset cannot run in a standard environment for that language but it only runs on the environment prepared by the teacher. Therefore, each stage has its own support tools that will render useless those helps that need programming concepts not in the language yet. Students are forced to solve these problems within the boundaries of their environment, and in some cases they are driven to solve previous problems using a whole different approach. Being driven to make as much profit as possible out of such a limited set of resources is what in our view trains problem solving versatility in the most effective way.

C. *Modularity as a learning asset*

The high mutability derived from modular composition could improve a programming course in several ways. First of all, a compiler created in this way can be adapted to fit in—without exceeding—what the students should learn lesson by lesson. A restricted scope helps to concentrate and exploit available features in a better way. This is particularly important when the teacher has to deal with both novice learners and autodidact programmers. It is undeniable and commonly appreciated that the interest in a computer science degree often derives from a self-nurtured interest. However unsupervised learning has the possible downside of resulting in some acquired quirks, not to say completely wrong coding behaviors. In our field experience, this commonly involves misuses of language constructs, limited problem solving capabilities—they code before thinking—and poor coding etiquette. The proposed teaching process can therefore be seen as a complete rewiring of what students should already know, aimed at making them (re)discover all possible approaches when facing a problem.

Moreover, the flexibility of a modular structure enables different learning paths tailored on the degree characteristics and course targets. So far, we chose—for sake of comprehensibility—to describe the approach as to be bound to a linearly incremental model, but this is far from being mandatory. Let us remind that language features cannot be considered as independent compartments: most of them actually have strong cooperative bounds between each other representing their related crosscutting concerns. These relationships would come to light as soon as any pervading feature gets included into the compiler bundle. If the teacher perceives that this could trigger some learning dynamics which he wants to postpone, he can as well decide to temporarily remove some old features in order to force the students to focus on the new features and to postpone the effects these concepts have on those already learned.

Such versatility implies another interesting asset: compiler growth guidelines do not have to be unique and predefined.

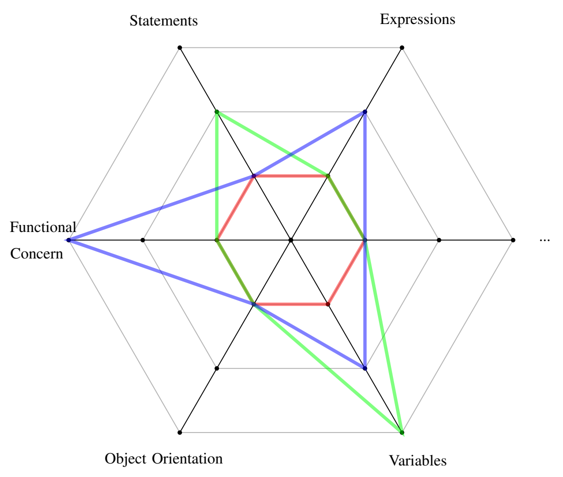


Fig. 2. Radar chart describing some of the potential learning paths

It is true that the final goal should always be to help students learn programming and master a programming language as a versatile and complete tool, but we are not limited to a single learning path. Figure 2 shows the interdependences between the programming concepts of a language and the possible learning paths a teacher can follow. The origin of the radar chart represents the empty language and each axis represents the degree of completion of a given programming concept—the complete language coincides with the external boundaries of the chart. According to what the teacher wants to prioritize, the growth of the language can protrude toward different directions before completeness. For example, it is either possible to prepare a course that starts by exploring in detail imperative constructs and internal state mutability, or a course based on the functional paradigm with first order functions, list type, operations and so on. The essential condition is that within every learning stage, modules should be added with respect to their dependency requirements (see [28]–[30]). This has to be done to maintain the internal language grammar consistent and devoid of gaps or unsatisfied rules—e.g., adding a *list comprehension* feature before defining other basic list manipulation constructs is useless. Both the incremental nature of this method and the finiteness of the language guarantee that regardless of the chosen approach, the final progression status will coincide with the complete language.

D. Course/language partitioning

Few things should be considered when partitioning a programming language for teaching:

- Each sub-language must be usable, that is, at each learning stage the sub-language contains enough programming concepts to permit the students to experiment with them.
- Each programming concept depends on other programming concepts—e.g., conditional statements need boolean expressions or something equivalent.
- Each sub-language must be self-contained—i.e., all the dependencies of a programming concept must be satisfied by the concepts in the sub-language itself.

Dependencies among programming concepts provide a relationship useful to group the programming concepts and partition the language. Programming concepts can be structured in a dependency graph and topologically sorted. Slicing the dependency graph according to the topological sort provides a set of sub-languages that respect the last two considerations. On the other hand, the resulting partition could be far from any teaching asset and the sub-languages could contain too many programming concepts to fit in the proposed learning model or could break the first consideration at all. Dependencies can be used to automatically provide an initial partition, but the teacher has to fine tune the result to his teaching decisions.

Dependencies represent also the blocking factor for the whole process. Programming concepts could be too interdependent and therefore difficult to split in separate sub-languages according to the teaching needs. For example, exception handling in Java relies on objects—exceptions in Java are objects—but exception handling is a general programming concept and it could be desirable to introduce it independently of the object-oriented concepts. To accomplish this need, some less demanding variants can be provided for any programming concept—e.g., an exception handling mechanism that deals with integers instead of objects—that will be replaced by the full version once the students are acquainted with all the needed concepts, i.e., they are in the sub-language used in the current learning stage.

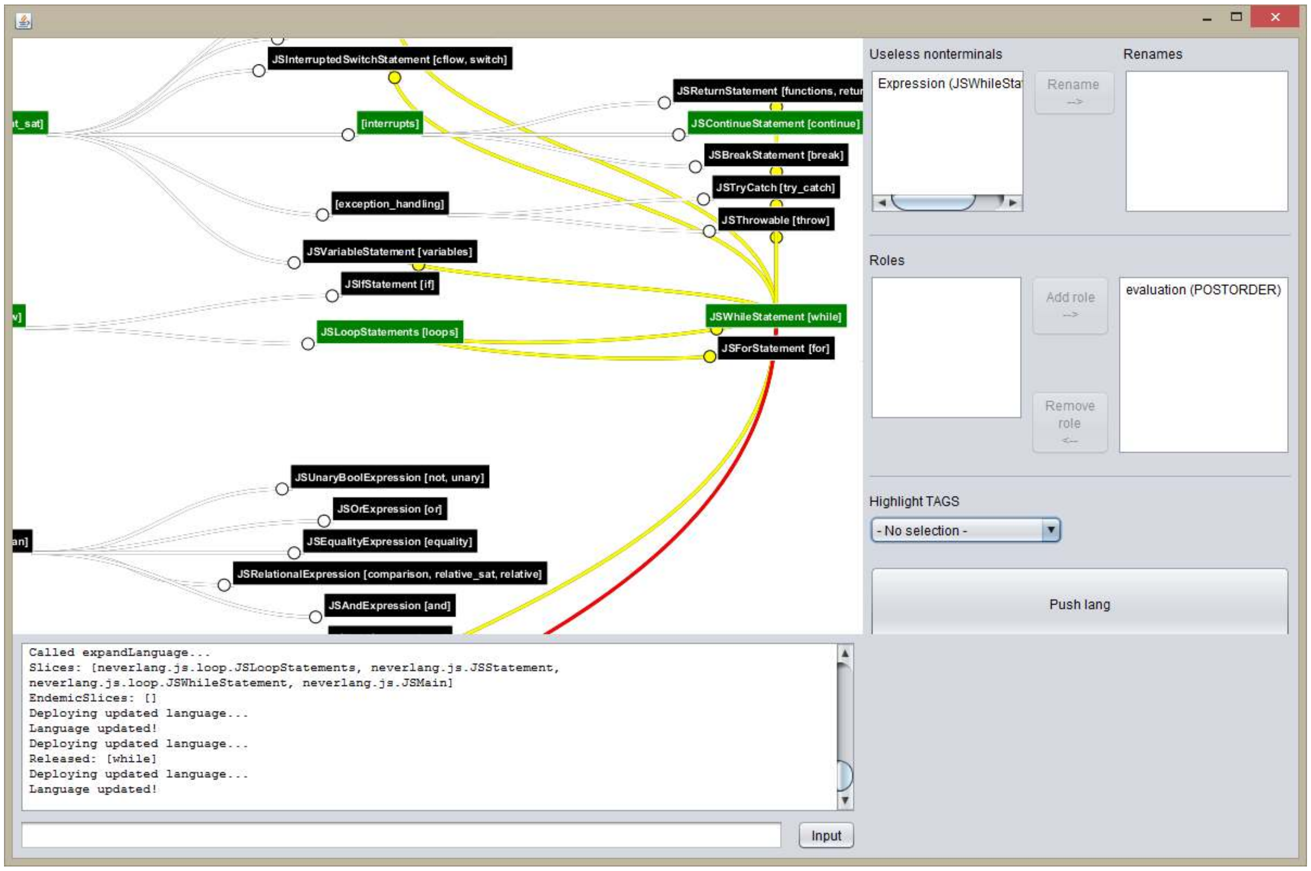
Partitioning can be carried out by directly choosing which modules should or should not be composed in the sub-language compiler. This, even if feasible, implies that the teacher has enough skills and time to dig through compiler code and can manually recognizes the dependencies to respect. This is not always the case and this process should be assisted by specific tools as in [27], [28].

III. Gradual learning: An Experimental Set-up

In the previous sections, we motivated and presented a general model for gradually teaching programming to freshmen students. Basically, this model relies on exposing the students to few programming concepts at a time with the support of an extensible language whose growth follows the teacher's teaching pace. Now we present an instantiation of the model that exploits the Neverlang development framework, the description of an experiment with the gradual teaching of Javascript, and the discussion of some initial results.

A. Neverlang language environment & AiDE

The Neverlang [23]–[25] development framework promotes code reuse and sharing by making language units first-class concepts. Language components are developed as separate units that can be compiled and tested independently, enabling developers to share and reuse the same units across different language implementations. The base unit is the **module** (Listing 1). A module—that coincides with a programming concept—may contain a **syntax** definition and/or a semantic **role**. A role defines actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique. Syntax definitions are portions



Expression (JSWhileSta

evaluation (POSTORDER)

```
Called expandLanguage...
Slices: [neverlang.js.loop.JSLoopStatements, neverlang.js.JSStatement,
neverlang.js.loop.JSWhileStatement, neverlang.js.JSMain]
EndemicSlices: []
Deploying updated language...
Language updated!
Deploying updated language...
Released: [while]
Deploying updated language...
Language updated!
```

Fig. 3. AiDE screenshot showing part of the feature model for Neverlang.JS (selected features are in green).

```
module com.example.AddExpr {
  reference syntax {
    AddExpr ← Term
    AddExpr ← AddExpr "+" Term;
  }
  role(evaluation) {
    0 .{ $0.value = $1.value; }
    2.{ $2.value = (Integer) $3.value + (Integer) $4.value; }.
  }
}
slice com.example.AddExprSlice {
  concrete syntax from com.example.AddExpr
  module com.example.AddExpr with role evaluation
}
language com.example.CalcLang {
  slices com.example.AddExprSlice com.example.MulExprSlice
         com.example.ParenExprSlice com.example.ExprAssocSlice
         com.example.NumbersSlice
  roles syntax < evaluation < ... // other roles
}
```

Listing 1: Neverlang's **slice** and **language** constructs.

of BNF grammars represented as sets of *grammar rules* or *productions*. Semantic actions are defined as code snippets that refer to nonterminals in the grammar.

Syntax definitions and semantic roles are tied together using **slices**. For instance, module `com.example.AddExpr` in Listing 1 declares a reference syntax for sum, and actions are attached to the nonterminals on the right of the two productions by referring to their position in the grammar. The slice `com.example.AddExprSlice` declares that we will be using this concrete syntax in our language with that particular semantics. Finally, the **language** descriptor (Listing 1), indicates which slices are required to be composed together to generate the compiler for the language. The **language** descriptor is the cornerstone of the whole mechanism and allows for easily restricting or extending a programming language. To transparently cope with the dependencies each programming concept has on other programming concepts, the AiDE tool is provided to select the desired components and to automatically generate the corresponding **language** descriptor.

The AiDE tool uses an internal clustering algorithm exploiting Neverlang dependency definitions to synthesize, optimize and manipulate the language's *feature model* (See Fig. 3). Through an interactive graphical user interface, the user can toggle different nodes of the feature model, triggering chain activation toward their parents or deactivation toward their leaves if required. While, an internal builder dynamically updates an internal implementation of a temporary language, adding or removing slices as necessary. The user can bind the ongoing language interpreter configuration to an interactive console, to verify the consistency of its language and test its behavior. When the language satisfies the required expectations, a stable copy of the development environment is prepared and ready to be dispatched to any JVM compliant

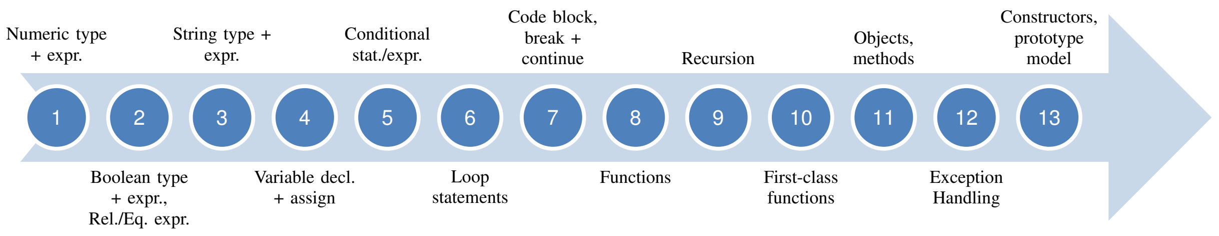


Fig. 4. Learning stages used to teach Javascript.

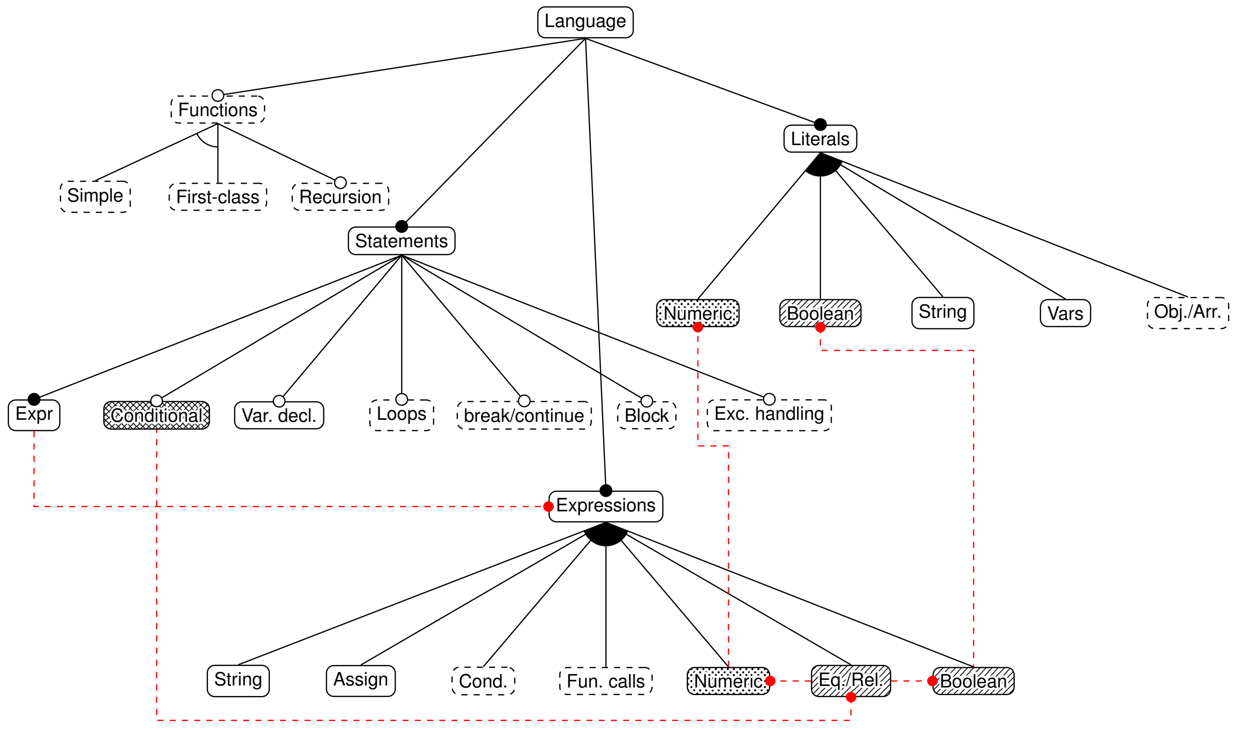


Fig. 5. An abstraction of the Javascript feature model.[3]

workstation.

The Neverlang development framework is shipped with a modular implementation of Javascript [33]. This implementation covers the ECMAScript 3 language specification. This Javascript modularization provides more than 50 programming concepts implemented in 73 slices—over 3000 code lines). A Java modularization is under implementation.

B. Gradual learning experiment

At the moment, the experimentation is quite limited and it has mainly focused on tuning up the whole process. Javascript was a forced choice since it is the only real programming language with a modular implementation in Neverlang. While it is true that rarely Javascript is mentioned as a propaedeutic programming language [34], its richness of programming concepts provided an interesting test bed for the course compartmentalization.

[3]The full feature model we get from AiDE is available at http://neverlang.di.unimi.it/aide/njs_graph.png.

The Javascript has been split in 13 sub-languages or learning stages. Figure 4 shows the chosen partitioning and also the adopted teaching path. Each sub-language extends the previous in the sequence with more programming concepts. The last sub-language coincides with the full Javascript. The considered sub-languages are:

1) the expression language on numbers;
2) boolean type with boolean and relational operators;
3) string type with operators on strings;
4) variable declarations and assignments;
5) conditional statements;
6) loop statements;
7) code block, scope, break and continue statements;
8) functions without recursion;
9) recursion;
10) first-class functions;
11) objects and methods;
12) exception handling and
13) constructors and the prototype model

```
var n = 10
var result = 1
while (n > 1) {
  result = result*n;
  n = n-1;
}
result;
```

a) with the 7th sub-language

```
function factorial(n) {
  var result = 1;
  while (n > 1) {
    result = result*n;
    n = n-1;
  } return result;
}
factorial(10);
```

b) with the 8th sub-language

```
function factorial(n) {
  if (n <= 1)
    return 1;
  else
    return n*factorial(n-1)
}
factorial(10);
```

c) with the 9th sub-language

```
function MyMath() {
  this.factorial = function(n) {
    if (n <= 1) return 1;
    else return n * this.factorial(n-1);
  }
}
new MyMath().factorial(10);
```

d) with the 11th sub-language

Listing 2: Evolution of the factorial implementation with the evolution of the Javascript sub-sets.

Once decided the learning stages, we followed the idea of a smooth expansion of the language capabilities, starting from an essential core calculator and gradually adding different kinds of expressions, literals and statements to finish with more complex concepts as functions, objects and exception handling, in accordance to the programming concepts relevant to each single stage in the chosen learning path. From the teaching point of view, we focused on the main programming concepts: to what extent these have to be taught is at teachers' discretion.

Each sub-language has its own interpreter that imposes the students' boundaries about what they can or cannot experiment with. Particular relevance relies on the evolution of the programming concepts when other concepts are introduced, e.g., in the first ten sub-languages numbers and booleans were just primitive types; once the objects are introduced in the language (11th sub-language) they are wrapped into objects whenever a method is applied to them. This was realized through different implementations of the same concept to be used according to the progress of the learning path. These extra modules were realized for this experiment since not available in the original modularization. The code we additionally implemented represents more or less the 7% of the total.

The subdivision of the concepts in learning stages has been done without considering the language modularization provided by Neverlang but according to the teaching needs. Once all the stages were clearly defined we started to cope with Neverlang and AiDE to realize the support tools (the interpreters of the sub-languages in our case). AiDE (Fig. 3) provides a graphical interface to the available programming concepts to choose from. Basically, all the programming concepts are structured in a feature model [35] that shows the dependencies, alternatives, and incompatibilities between concepts. Selecting a programming concept automatically includes all the programming concepts such a concept depends on—e.g., the inclusion of the $*$ operator automatically includes the numeric primitives. The few variants—of the same concept we had to define—were introduced as mutually exclusive alternatives and an explicit choice was necessary to include the correct version at each learning stage.

Figure 5 depicts an abstract view of the Javascript feature model. In particular, it shows the AiDE configuration related to the fifth sub-language. All the programming concepts with a solid border are part of the language. Nodes filled with the same pattern represent programming concepts belonging to one of the previous sub-languages: numeric (▦), booleans (▨), and *conditional* statement concern (▩). The red dashed lines (- - -●) represent dependency relationships between single features, that must be respected to guarantee a stable growth. Inner nodes represent groups of programming concepts, while nodes with a dashed border (⬚) denote programming concepts not selected yet. Out of this configuration we get an interpreter supporting only the selected programming concerns and that can be used by the students in the corresponding learning stage.

Each learning stage focused on the study of the new programming concepts both in theory through frontal lectures but especially on hands-on sessions and homeworks. The hands-on session had a semi-tutorial structure and helped in focusing on some proposed problems and their solution with the available programming concepts. Some of the proposed problems are repeated from a learning stage to the other to permit the students to experiment on how the problem solution can benefit from the newly introduced concepts. For example, the factorial is one of the recurring problems and Listing 2 reports some of the most interesting variants. These four code fragments show the evolution of the factorial implementation along with the progresses on the Javascript coverage:

1) The first script has been realized in a pure imperative context, with only variables, numeric and boolean types, arithmetic and relational expressions, assignments, conditional and loop statements.
2) The context evolves through the addition of functions without either recursion or first-class.

3) The next step adds the availability of recursion within function body allowing for a significant code simplification.
4) In the last version, an object is created as a container for the value, and the function is used as one of its methods. This further expands the idea of variable scoping and contextual binding (**`this`**).

Few things should be said about the considered learning path. We decided to initially focus on the imperative aspect of Javascript then on functions and only finally on objects. We are aware that this could be arguable, but this is not our point: any teacher can be comfortable in teaching programming following the path he prefers. The proposed model is not stick to a particular learning path, but it depends only on the teacher's decisions: this is the point. It is pretty easy to change the learning path and give more emphasis to functional programming: it is just a matter of selecting the programming concepts related to function declaration, function call, recursion, and first class functions before those about variables, assignments and control flow statements. Roughly speaking the 8th, 9th and 10th sub-languages in Fig. 4 should be set before the 4th sub-language in the learning path. The "roughly" is because the new partitioning would have completely new sub-languages than those we used. This also bring forth to some different implementations of the proposed exercises, e.g., a first significant version of the factorial script can already be realized with the 5th sub-language:

```
function factorial(n) {
  return ((n <= 1) && 1) || (n * factorial(n-1));
}
factorial(10);
```

that will be completely functional and a gradual language increment will divert factorial implementation from those shown in Listing 2—e.g., to show a functional version that does not use recursion becomes impossible since without both loops and recursion you cannot implement the factorial.

Also the linear progression is not set in stone, as programming concepts can be added from a learning stage to the other it is also possible to remove them and it is completely up to the teacher. Hence, for example, if the teacher decides that functions and recursion are not necessary—if not an obstacle—to the learning of the full extent of the control flow statements he can remove them from the programming language and fork the learning path.

C. Experimental evaluation

The initial experimentation has been done with a set of 10 subjects chosen among the freshmen enrolled to the first year programming course of the computer science degree. The students volunteered and they represented more or less the 10% of the whole class. The students went to different high schools with different backgrounds and different final marks; half of them already learned programming, a couple of them were autodidacts and one of them was already acquainted with Javascript.

The course lasted 14 weeks with 8 teaching hours per week. Basically each frontal lecture has at least one hands-on session

week n.	*slots*	*learning stages*
1	*num. & expr.*	1. numeric type and operators
	hands on session	
	boolean	2. boolean type and relational operators
	hands on session	
2	*strings*	3. string type and operators
	hands on session	
	var, array	4. arrays, variables and assignment
	hands on session	
3	*ternary expression*	5. conditional expressions and statements
	hands on session	
	if, switch	
	hands on session	
4	*while, do while*	6. loop statements
	hands on session	
	for, for in	
	hands on session	
5	*block, break, continue*	7. code blocks, break and continue statements
	hands on session	
	hands on session	
	hands on session	
6	*function definition*	8. function definition and use
	arguments, return	
	hands on session	
	hands on session	
7	*hands on session*	
	hands on session	
	recursion	9. recursion
	recursion vs iteration	
8	*hands on session*	
	accumulator pattern	
	hands on session	
	first-class function	10. first class function
9	*hands on session*	
	closures, generators	
	hands on session	
	hands on session	
10	*objects, methods*	11. objects and methods
	hands on session	
	hands on session	
	setter, getter	
11	*hands on session*	
	generator object	
	hands on session	
	hands on session	
12	*exception*	12. exception handling
	try, catch, throw	
	hands on session	
	hands on session	
13	*func., gen. constructors*	13. constructors and prototype model
	hands on session	
	hands on session	
	hands on session	
14	*inheritance, prototype chain*	
	hands on session	
	hands on session	
	hands on session	

TABLE I
DISTRIBUTION OF THE LEARNING STAGES OVER THE 14 WEEKS

associated. Homeworks were done out of the teaching hours. The first two weeks covered the learning stages from one to four; since then any learning stages lasted one week apart 4 weeks for the stages from eight to ten and 2 weeks each per the eleventh and thirteenth stages. In parallel the remaining students were learning Java with the traditional approach. The 10 students had to do the exams with the other students on Java, thus a few more lessons (a week) to catch up the differences with Java were necessary. Table I summarizes the distribution of the learning stages over the fourteen weeks. Note that the term week is used as a logical container for four lessons not necessarily these lessons occurred in the same calendar week but they could span several calendar weeks to consider holidays, breaks and strikes. Table I also highlights the number of frontal and hands on sessions dedicated to each learning stage (each slot lasts 2 consecutive hours).

The experiment has been evaluated through a questionnaire to answer the following research questions:

1) does the proposed learning process favorite problem-solving over programming language?
2) does the proposed learning process annihilate previous knowledges about programming?

The questionnaires were anonymous and delivered to the students once they passed the exam; this freed the students from the fear of retaliation in case of a negative judgement. The questionnaire is divided into two parts. The first part asked for the students' previous knowledge of programming with questions as which languages they know, how and when they learned it and so on—the figures at the begin of this sub-section summarize the results of these questionnaires—the second part were open questions about the course and the learning process. Some of the questions were:

- Did the limited number of concepts at disposal help you focusing on problem solving?
- Did an interpreter that supports only the known concepts help you focusing on problem solving?
- If you already knew how to program did you benefit from your previous knowledges?
- If you already knew how to program did you learn programming differently from the first time?

Single answers were terse (more or less one sentence each) and we could manually analyze them. All the answers to the same question where collected together. From each answer we extracted all the nouns and the associated adjectives. For each noun the extracted adjectives have been normalized through the use of a simple ontology and all synonyms were replaced by a champion adjective provided by the ontology; the normalization was necessary to level the different terminology used by the students. Then the frequency of the normalized adjectives has been calculated and used to establish the percentage of satisfaction. The comments were positive. All the students stated that the approach nurtured the focusing on problem solving (50% of the students gave an excellent judgement whereas the other 50% a very good judgement, these last set includes all the students already acquainted with programming). The question about the interpreter got identical remarks and the sensation is that the students tend to unify the number of concepts with the interpreter that supports them. All the students already acquainted with programming admitted that they did not benefit from the previous knowledges and that learning was different since they had to approach problem solving from a different perspective—limited and fixed number of concepts at disposal. To further sum up the answers:

i) a development framework that grows with students' knowledges helps to focus on the problems rather than on the language,
ii) experienced students are challenged to solve the problems without relying on the previous knowledges and from a different perspective.

This positively satisfy our initial research questions. The feelings of the students were in line with our expectations. In particular, we were aware that the students already acquainted with programming could be disappointed or annoyed by the proposed approach, in fact, reading the comments (and not their simplification) we discover that they were initially frustrated by the approach that limited their possibility especially on the initial problems too easy for them. In the end they were happy about the process since it permitted to approach different solutions to the same problem with different programming concepts.

We are far from a real evaluation but we are pretty satisfied with the students' comments. Especially considered that all of them were able to pass the exam on a different—even if—close language. In the future, we are going to do a more exhaustive evaluation tailored on the learning rather than on the process that will involve more colleagues and different programming languages.

IV. Related work

The **Mini-languages** approach [32], [36] arose significantly after the first release of the didactic language Logo [37]. Those languages consist of a simple and mostly graphic programming environment, where an *actor* has to find his way to solve a given task within its microworld: this is possible through the insertion of a list of commands in sequence, given by the student, by way of a simple but structured scripting language. Most of them provide basic control structures—conditionals, loops, recursions—and even mechanisms to create custom instruction sets. There are many examples of them, with distinct types of target audience ranging from elementary school students to college freshmen. It was largely demonstrated how the short time required to master a mini-language allowed the students to focus on the more important issues of problem solving and algorithm development [32], therefore setting a well-founded basis for the learning of more complete languages. The downside is that these languages can hardly be extended outside the scope of their micro-world, and so they mostly fulfill their utility as a preliminary learning tool. Domain-specific languages can be considered today's *mini-languages*, their development is flexible enough to ease their growing [31], [38] and therefore they could be used to teach programming as described in this work.

SP/k [39] was one of the first experiments aimed at teaching how to program in a complete language—in this particular case *PL/I*—by providing a controlled subset. In a similar fashion to our proposal, they developed distinct learning steps: each of them adds new language features while retaining the previous ones. The difference is that these steps are precompiled, non interchangeable and strictly sequential: *SP/1* starts from simple expression interpretation, *SP/2* introduces variables and assignment, *SP/3* selection and repetition, and so on. Though being considered a subset of PL/I, SP/k has its own compiler, since some semantic concerns implicitly solved in PL/I—e.g., automatic type conversion, typos in reference names—are treated differently in SP/k, to solicit the attention of the students. Of course being PL/I an obsolete language itself, it is not realistic to think to employ such tool within modern programming courses.

DrScheme [40] is a tool based on the Scheme programming language, which acts as a proxy environment. It is designed to limit the syntax for those Scheme features that are commonly misleading when used by unexperienced programmers. Moreover it is capable of giving more precise responses when dealing with particular errors or corner cases. The provided environment allows to enable different levels of syntax and protection by some predefined settings. As a complete toolset it includes a syntax checking tool and a static debugger. Even DrScheme is not actually modular by definition, and as much as SP/k is created as a set of increasing clusters upon the language syntax and features. As a learning toolset, it is strictly paired with its graphical development environment and its usage within the original Scheme interpreter is hardly possible.

ProfessorJ [11] is another similar tool which uses the same approach of DrScheme applied to Java, as its graphical programming environment itself maintains the same structure. As an additional feature, much of Java auxiliary constructs—class and method modifiers for instance—are given as hidden, implicitly defined, and non-rewritable at the core of the environment, to show their proper usage little by little, each increasing cluster upon the language. Similarly to DrScheme the increasing clusters are quite coarse and prearranged in a fixed way.

V. Conclusions

Teaching how to program is a hard task. It is particularly difficult to separate the teaching of the programming language from the teaching of the problem solving aspect. Teachers try to compartmentalize the programming concepts to expose the students to few concepts at a time and get their attention on solving the problem rather than on learning the language syntax. This approach tends to fail due to a lack of support by the tools that instead immediately expose the students to the whole language. This work proposes an *assisted* teaching model that permits to expose the students to few programming concepts at a time with the support of an incremental development framework. We experimented this teaching model by teaching how to program with Javascript. The experience is just at the beginning but the results look promising.

References

[1] L. E. Winslow, "Programming Pedagogy—A Psychological Overview," *ACM SIGCSE Bulletin*, vol. 28, no. 3, pp. 17–22, Sep. 1996.

[2] A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, Mar. 2003.

[3] Y. Hofuku, S. Cho, T. Nishida, and S. Kanemune, "Why Is Programming Difficult? Proposal for Learning Programming in "Small Steps" and a Prototype Tool for Detecting "Gaps"," in *Proceedings of the 6th International Conference on Informatics in Schools: Situation, Evolution and Perspectives (ISSEP'13)*, Oldenburg, Germany, Feb.-Mar. 2013, pp. 13–24.

[4] I. Milne and G. Rowe, "Difficulties in Learning and Teaching Programming—Views of Students and Tutors," *Journal of Education and Information Technology*, vol. 7, no. 1, pp. 55–66, Mar. 2002.

[5] P. Charters, M. J. Lee, A. J. Ko, and D. Loksa, "Challenging Sterotypes and Changing Attitudes: The Effect of a Brief Programming Encounter on Adults' Attitudes toward Programming," in *Proceedings of the 45th ACM Symposium on Computer Science Education (SIGCSE'14)*, J. Dougherty and K. Nagel, Eds. Atlanta, GA, USA: ACM, Mar. 2014, pp. 653–658.

[6] R. F. Paige, F. A. C. Polack, D. S. Kolovos, L. M. Rose, N. Matragkas, and J. R. Williams, "Bad Modelling Teaching Practices," in *Proceedings of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS'14)*, Valencia, Spain, Oct. 2014, Keynote.

[7] G. M. Schneider, "The Introductory Programming Course in Computer Science: Ten Principles," in *Proceedings of the SIGCSE/CSA Technical Symposium on Computer Science Education (SIGCSE'78)*, K. Williams, Ed., vol. 10, no. 1. ACM, Feb. 1978, pp. 107–114.

[8] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A Survey of Literature on the Teaching of Introductory Programming," in *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*, J. Carter and J. Amillo, Eds. Dundee, Scotland: ACM, Jun. 2007, pp. 204–223.

[9] S. Wiedenbeck and V. Ramalingam, "Novice Comprehension of Small Programs Written in the Procedural and Object-Oriented Styles," *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 71–87, Jul. 1999.

[10] K. Malan and K. Halland, "Examples that Can Do Harm in Learning Programming," in *Proceedings of the 19th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, J. M. Vlissides and D. C. Schmidt, Eds., Vancouver, BC, Canada, Oct. 2004, pp. 83–87.

[11] K. E. Gray and M. Flatt, "ProfessorJ: A Gradual Introduction to Java through Language Levels," in *Proceedings of the 18th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, G. Steele, Jr and R. P. Gabriel, Eds. Anaheim, CA, USA: ACM Press, Oct. 2003, pp. 170–177.

[12] C. Bruce, L. Buckingham, J. hynd, C. McMahon, M. Roggenkamp, and I. Stoodley, "Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University," *Journal of Information Technology Education*, vol. 3, no. 1, pp. 143–160, 2004.

[13] B. Victor, "Learnable Programming," Sep. 2012, available at http://worrydream.com/LearnableProgramming.

[14] J. Rogalski and R. Samurçay, "Acquisition of Programming Knowledge and Skills," in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, Eds. Academic Press Limited, 1990, ch. 2.4, pp. 157–174.

[15] M. Petre, "A Paradigm, Please—and Heavy on the Culture," in *User-Centred Requirements for Software Engineering Environments*, ser. Lecture Notes in Computer Science 123, D. J. Gilmore, R. L. Winder, and F. Détienne, Eds. Springer, 1994, pp. 273–284.

[16] R. Mason and G. Cooper, "Why the Bottom 10% Just Can't Do It—Mental Effort Measures and Implication for Introductory Programming Courses," in *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)*, M. de Raadt and A. Carbone, Eds., vol. 123, Melbourne, Australia, 2012, pp. 187–196.

[17] J. Bennedsen and M. Caspersen, "Model-Driven Programming," in *Reflections on the Teaching of Programming*, ser. Lecture Notes in Computer Science 4821, J. Bennedsen, M. E. Caspersen, and M. Kölling, Eds. Springer, 2008, pp. 116–129.

[18] A. L. Olsen, "Using Pseudocode to Teach Problem Solving," *Journal of Computing Sciences in Colleges*, vol. 21, no. 2, pp. 231–236, Dec. 2005.

[19] P. A. Kirschner, J. Sweller, and R. E. Clark, "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experimental, and Inquiry-Based Teaching," *Educational, Psychologist*, vol. 41, no. 2, pp. 75–86, 2006.

[20] M. Mernik and V. Žumer, "Incremental Programming Language Development," *Computer Languages, Systems and Structures*, vol. 31, no. 1, pp. 1–16, Apr. 2005.

[21] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: an Extensible Attribute Grammar System," *Science of Computer Programming*, vol. 75, no. 1-2, pp. 39–54, Jan. 2010.

[22] L. C. L. Kats and E. Visser, "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*, M. Rinard, K. J. Sullivan, and D. H. Steinberg, Eds. Reno, Nevada, USA: ACM, Oct. 2010, pp. 444–463.

[23] W. Cazzola, "Domain-Specific Languages in Few Steps: The Neverlang Approach," in *Proceedings of the 11th International Conference on Software Composition (SC'12)*, ser. Lecture Notes in Computer Science

7306, T. Gschwind, F. De Paoli, V. Gruhn, and M. Book, Eds. Prague, Czech Republic: Springer, Jun. 2012, pp. 162–177.

[24] W. Cazzola and E. Vacchi, "Neverlang 2: Componentised Language Development for the JVM," in *Proceedings of the 12th International Conference on Software Composition (SC'13)*, ser. Lecture Notes in Computer Science 8088, W. Binder, E. Bodden, and W. Löwe, Eds. Budapest, Hungary: Springer, Jun. 2013, pp. 17–32.

[25] E. Vacchi and W. Cazzola, "Neverlang: A Framework for Feature-Oriented Language Development," *Computer Languages, Systems & Structures*, 2015.

[26] S. Apel, C. Kästner, and C. Lengauer, "Language-Independent, Automated Software Composition," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. Vancouver, BC, Canada: IEEE, May 2009, pp. 221–231.

[27] J. Liebig, R. Daniel, and S. Apel, "Feature-Oriented Language Families: A Case Study," in *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, P. Collet and K. Schmid, Eds. Pisa, Italy: ACM, Jan. 2013.

[28] E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale, "Variability Support in Domain-Specific Language Development," in *Proceedings of 6th International Conference on Software Language Engineering (SLE'13)*, ser. Lecture Notes on Computer Science 8225, M. Erwig, R. F. Paige, and E. Van Wyk, Eds. Indianapolis, USA: Springer, Oct. 2013, pp. 76–95.

[29] E. Vacchi, W. Cazzola, B. Combemale, and M. Acher, "Automating Variability Model Inference for Component-Based Language Implementations," in *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, P. Heymans and J. Rubin, Eds. Florence, Italy: ACM, Sep. 2014, pp. 167–176.

[30] T. Kühn, W. Cazzola, and D. M. Olivares, "Choosy and Picky: Configuration of Language Product Lines," in *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, G. Botterweck and J. White, Eds. Nashiville, TN, USA: ACM, Jul. 2015.

[31] W. Cazzola and D. Poletti, "DSL Evolution through Composition," in *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*. Maribor, Slovenia: ACM, Jun. 2010.

[32] P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller, "Mini-Languages: A Way to Learn Programming Principles," *Journal of Education and Information Technologies*, vol. 2, no. 1, pp. 65–83, 1997.

[33] E. Vacchi, D. M. Olivares, A. Shaqiri, and W. Cazzola, "Neverlang 2: A Framework for Modular Language Implementation," in *Proceedings of the 13th International Conference on Modularity (Modularity'14)*. Lugano, Switzerland: ACM, Apr. 2014, pp. 23–26.

[34] R. Ward and S. Martin, "JavaScript as a First Programming Language for Multimedia Students," in *Proceedings of the 6th Annual Conference on the Teaching of Computing (ITiCSE'98)*, G. Davies and M. Ó'Higeartaigh, Eds., Dublin, Ireland, Aug. 1998, pp. 249–253.

[35] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Technical Report CMU/SEI-90-TR-21, Nov. 1990.

[36] H. F. Ledgard, "Ten Mini-Languages: A Study of Topical Issues in Programming Languages," *ACM Computing Surveys*, vol. 3, no. 3, pp. 115–146, Sep. 1971.

[37] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon, "Programming Languages as a Conceptual Framework for Teaching Mathematics," *ACM SIGCUE Outlook*, vol. 4, no. 2, pp. 13–17, Apr. 1970.

[38] I. Fister, Jr, T. Kosar, I. Fister, and M. Mernik, "EasyTime++: A Case Study of Incremental Domain-Specific Language Development," *Information Technology and Control*, vol. 42, no. 1, pp. 77–85, 2013.

[39] R. C. Holt, D. B. Wortman, D. T. Barnard, and J. R. Cordy, "SP/k: A System for Teaching Computer Programming," *Communications of the ACM*, vol. 20, no. 5, pp. 301–309, Apr. 1977.

[40] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen, "DrScheme: A Pedagogic Programming Environment for Scheme," in *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, ser. Lecture Notes in Computer Science 1292. Southampton, UK: Springer, Sep. 1997, pp. 369–388.

Walter Cazzola is currently an Associate Professor at the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory.

He is the designer of the mChaRM framework, of the @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang general purpose compiler generator. He has written over 100 scientific papers. His research interests include reflection, aspect-oriented programming, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics.

Dr. Cazzola has taught several courses about programming either in the undergraduate, graduate and Ph.D. course of studies, in the last twenty years.

Diego Mathias Olivares is currently pursuing the Ph.D. degree with the Department of Computer Science, Università degli Studi di Milano. He is a teaching assistant with the Università degli Studi di Milano. He is also a member of the ADAPT laboratory.

He is one of the contributors to the Neverlang framework; in particular, he developed the AiDE tool and the Neverlang implementation of Javascript. His research interests are about the teaching of programming and development of programming languages.