

SIPE: A Domain-Specific Language for Specifying Interactive Programming Exercises

Jakub Swacha

Abstract The paper introduces a new domain-specific language designed essentially for specifying exercises for interactive programming courses. The language covers the four crucial elements of interactive programming exercise, i.e.: metainformation, exercise description, solution checking and feedback, and aims at conciseness, human readability and ease of use. The paper discusses the issues with alternative forms of specifying programming exercises, presents the general concepts and syntax of the language, and reports on its implementation and validation.

Keywords Programming learning environments • Exercise specification • E-learning

1 Introduction

An important element of an effective teaching of computer programming is providing the students the ability to try the code they write as an exercise and obtain feedback on whether it is syntactically and semantically correct. Traditionally, it was a role of the instructor to check the solutions produced by the students, and provide them with an appropriate feedback. With the development of interactive programming learning environments, the instructor could be relieved from this time-consuming duty.

Automating this process brings also other major benefits, by allowing the students to proceed with their own pace, and get feedback whenever they fail or succeed, with no delay and regardless of the time of day. The value of automatic feedback for the effectiveness of programming learning has been confirmed in research [1].

J. Swacha (✉)
Institute of Information Technology in Management,
University of Szczecin, Szczecin, Poland
e-mail: jakubs@uoo.univ.szczecin.pl

It is crucial that the feedback, though automatic, should be meaningful, that is help the students understand the reasons of the errors they committed and possibly suggest ways to overcome them. This can only be achieved by triggering the feedback using specially designed tests based on the knowledge of the problem, appropriate solution methods, used programming language and the relevant typical programming mistakes [2].

Thus, the traditional instructor's procedure of exercise preparation consisting of three stages: conceive, explain to students, check solutions (of which the latter two have to be repeated for each lesson), is replaced by a new procedure, consisting of another three stages: conceive, describe, specify the tests and feedback (of which none has to be repeated for each lesson). Although there is a time gain from not having to repeat specifying the tests and feedback for each lesson, the difficulty and arduousness of the specification can make this gain disappear or even become a loss.

In this paper it is assumed that the way an interactive programming exercise is specified is related to the necessary instructor's effort. It can be argued that a well-designed language for describing programming exercises and specifying tests and feedback can reduce the time required to perform these steps both directly, by making it simpler to translate from concepts to specification and the result more concise, and indirectly, by making the specification simpler and more human-readable, thus easier to understand, correct, reuse, and improve. The aim of this paper is to introduce a domain-specific language designed essentially to meet these goals. For ease of reference, the language was called SIPE (from *a language for Specification of Interactive Programming Exercises*).

As much as most of the innovation comes from real-world needs, the inspiration for SIPE were the issues encountered by this author during many hours spent on specifying exercises for his own interactive programming learning environment [3]. These issues are discussed in Sect. 3 of this paper. Before that, Sect. 2 provides a short review of related work on programming exercise specification.

The key Sect. 4 describes the general design of SIPE and its syntax, whereas Sect. 5 focuses at the crucial ingredient of SIPE, that is the sublanguage used to specify the test and feedback parts of exercise specification. Section 6 describes the real-world implementation and validation of SIPE, and the final section concludes.

2 Related Work

Although interactive environments for programming learning have become a commodity these days, so far no standard for specifying exercises for such systems has been established and widely adopted, and often ad hoc-designed, proprietary formats are used for this purpose. The closest proposal to a what could be considered a standard is PExIL [4]. It covers all the primary aspects of exercise specification, including metadata and task description as well as definition of output

tests and feedback. Still, PEXIL is XML-based and as such does not meet the goals, presented above, of conciseness and ease of viewing and editing as text.

Although aimed at quite a different goal, but worth mentioning in this context, is the Unified Exercise Answering Format presented in [5], used as standardized form of reporting solutions of programming exercises.

Looking at the test and feedback specification alone, the usual approaches are to check the existence (or lack) of text patterns specified explicitly or using regular expressions. A general-purpose programming language is also often employed for this purpose, which may but need not be the language the solution is written in [2], but that seems to be a far too sophisticated tool in the case of interactive programming courses where the most of the performed tests are very simple. Even operating system shell scripts could be used for this purpose [6, p. 4], but such a tool, though relatively simpler, has a lot of limitations of its own.

Another approach is to use special annotation embedded in the model solutions to specify the feedback [7], yet it is not applicable, when there is no model solution.

The concept of providing a wrapper extending regular expressions was investigated in [8], though that solution uses a specific notation only for output- and input-checking, whereas mere regular expressions are used for pattern specification for source code checking.

3 Issues of Exercise Specification

The issues discussed below are based on actual issues encountered by this author during the development of his own interactive programming course [3]. Although the goal of this paper is not to document the way it was implemented (much of which can be found in [9]), some basic knowledge regarding it is necessary to understand the context of the issues.

The above-mentioned course uses JSON files to store information about the exercises. One such file contains specification of all exercises belonging to one course unit (lesson). The exercises are defined as an array of objects, each containing fields with the respective exercise specification elements, such as ‘title’ (text to display as the exercise title), ‘task’ (text of HTML description of what the student is instructed to do), ‘outputHas’ (array of search phrases defined with regular expressions each of which has to match the contents of the output generated by the student’s solution for it to be considered correct), or ‘outputHasHint’ (corresponding array of text messages containing feedback for the student in case search phrases were not matched, e.g. if the first search phrase was not matched, the first message is displayed as hint).

The explicit examples of some of the discussed issues are listed in Table 1 at the end of this section, which also presents how such issues disappear when using SIPE.

The first issue experienced with all the three elements mentioned above is the required use of delimiters and separators ensuing from the fact that JSON has to conform to JavaScript syntax rules. The text has to be put into quotes, and while it lasts for several editor lines, a backslash has to be appended at the end of each non-terminal line. The most frequent reason of syntax errors with exercise specification found was forgetting to put the ‘\’ at the line end. The second: missing a comma between elements of an array—usually after appending a new element to it in a new line.

The second issue was the existence of forbidden characters and phrases, and the required use of escape characters as a work-around for this limitation. As for the HTML-formatted text, the most frequent issue was caused by comparison operators (‘<’, ‘>’), which were often forgotten to be replaced with respective entities (‘<’, ‘>’). As for the source code (both the initial solution source code and examples contained in the exercise description), the problem was with quotes and the backslash. This issue was augmented by the fact these characters have to be escaped both in the language of the source code (Python in this instance) and the language of the specification (JavaScript) which not only made it easy to make a mistake, but also made source code strings containing a number of literal backslash characters (which thus were required to be doubled twice) to be represented in a form highly obscure to a human reader.

The third issue was the frustrating nuisance due to the verbosity of HTML markup, requiring typing tag names within triangle brackets, and doing it twice—at the begin (sometimes also with class attributes) and the end of every marked phrase.

3.2 *Test Specification*

Using regular expressions to define the patterns that should or should not be found in the solution source code or the output of its execution revealed a number of issues, ranging from small but frequent nuisances (like the optional whitespace problem) to rare but serious difficulties requiring use of sophisticated and elaborate expressions (like the multiple compound expressions with different order of items problem). While all of them reduce the human readability of the specification, the latter make it even more difficult to analyze, verify and reuse. In the list below, nine such problematic cases are described.

- The optional whitespace problem. Most programming languages permit optional whitespaces, which have to be accommodated for in the test patterns. As a result, the single spaces are replaced with whitespace wildcards, obfuscating the pattern specification.
- The special characters problem. The special characters of regular expressions (e.g. ‘.’, ‘(’, ‘)’, ‘[’ or ‘*’) are very frequently used in program source, thus also frequently included in patterns, in which they have to be escaped. As a result, many backslashes are inserted, obfuscating the pattern specification.

- The escaped characters problem. Backslash is used as the escape character in regular expressions and many programming languages, where it can also serve other purposes. As a result, the number of inserted backslashes grows.
- The valid code problem. Most of the patterns looked for in the solution source code is expected to match valid program instructions, but they can as well match literal string constants and comments. A special pattern can be inserted in front of the actual pattern sought to avoid that but it obfuscates the pattern specification.
- The code within block problem. In languages forming blocks by indentation (such as Python), looking for pattern that is expected inside a compound statement defined by another pattern requires a sophisticated regular expression, using group references. Similar problem is with looking for patterns within string literals, which can be expressed using various notations (different delimiters, joined single-line, multi-line).
- Exact number of occurrences problem. Regular expressions checking non-occurrence of patterns look awkward and contribute to low readability of test specification.
- Alternatives without specified order problem. Sometimes there are several ways to do something and there is need to check that more than one way of doing it was actually used in the solution. An exemplary exercise included checking whether a student is able to call functions using different order of three parameters. The used regular expressions was long and complex.
- Ordered list problem. Exercises may require the solution to produce output in sorted order. Checking that using regular expressions is a nuisance, especially if the range of expected values is large.
- Pattern reference problem. The regular expression dialect implemented in JavaScript is somewhat limited, for instance it lacks references to user-defined patterns (references to match groups are of course supported). In case specific pattern is used multiple times in a regular expression, its definition has to be repeated each time, making the expression unnecessarily long and thus difficult to grasp.

Table 1 presents examples of the issues described above; its rightmost column shows how the same goal can be achieved using far less complicated code written in SIPE.

4 SIPE: Basic Concepts and Syntax

The main design aims behind SIPE were:

- make the exercise specification easy to write in a general-purpose text editor,
- make all the necessary functionality (e.g. adding meta-information, special formatting of phrases within exercise description, specifying check patterns) available to the user,

- avoid the verbosity and complexity issues experienced with HTML, JSON and solution checks based only on regular expressions,
- keep it simple, especially the often used specification constructs,
- make it flexible, as individual course authors may have needs that would be hard to foresee.

In a quest for simplicity, a flat document structure was chosen as the basis for SIPE. This means that there are no logical brackets (like e.g. ‘{’ and ‘}’ in JSON), and only the position of an element in a document defines its relation to other elements. Thus, if element type *lesson* (see further down for more details) describes a set of exercises, *exercise*—a single exercise, and *initcode*—a specific component of an exercise, the content of an *initcode* element belongs to the last preceding *exercise*, and the content of an *exercise* element belongs to the last preceding *lesson*. Thanks to that, whole SIPE document is a sequence of just two interleaving types of blocks:

- section header, defining the meaning of the content of a subsequent section,
- section content, i.e. the actual specification of the exercise element denoted by the header.

4.1 Section Header

The section header is defined as follows (from now on, ABNF notation [10] is used for specification of key SIPE language elements):

```
section-header = section-name [ "." content-type ] ":"
                *WSP [ section-title ] LF
```

The predefined section names are listed in Table 2. Only the *exercise* and *task* sections are required (without them, the document could not be called an exercise specification). As for the others, for the sake of flexibility, SIPE takes on a ‘defined if exists’ approach: the remaining sections are optional (they can be omitted), but:

- if a section is included in a document, it should contain exactly what the specification defines for such a section,
- if the content defined by the specification is included in a document, it should be put into the section having relevant name.

In other words, the predefined section names cannot be used for purposes other than specified, and their corresponding content types cannot be kept in sections having other names. The parser accepts any section name conforming to requirements (only letters are allowed). Note that all the structural elements of SIPE (including section names) are case-insensitive.

Table 2 Predefined SIPE section names

Section name	Content	Default content type
Lesson	Exercise collection title and metadata	Name: Value
Exercise	Exercise title and metadata (required!)	Name: Value
Intro	Text introducing the concepts needed to solve the task	Markdown
Task	Text explaining the task the student is supposed to perform (required!)	Markdown
Precode	Source code that will be injected at the beginning of the student's code before its execution	Plain text
Initcode	Source code that will be presented to the student as a skeleton for the solution	Plain text
Postcode	Source code that will be injected at the end of the student's code before its execution	Plain text
Inputparameters	Comma-separated values that should be fed to the tested program (exercise solution) as input	Plain text
Checksource	Code to check the tested program source code	SCSL
Checkinput	Code to check the tested program user-specified input	SCSL
Checkoutput	Code to check the tested program output	SCSL
Checkerrors	Code to check error messages generated by the tested program	SCSL

The content type can be changed from default by specifying the actual content type following the section name. The predefined names for content types are: *HTML*, *MD* (for Markdown), *NameValue* (for Name:Value pairs), *plain* (for plain text) and *SCSL* (for the Solution Check Specification Language). The parser accepts any content type name conforming to requirements (only letters are allowed). The interpreter may but cannot be guaranteed to handle user-defined content types.

The section title is optional, and, in practice, is used only for *lesson* and *exercise* sections. The section header ends with the end of line.

4.2 Section Content

The section content is defined as follows:

```
section-content = content LF LF labelchar
```

Content can be a sequence of any characters excluding the ending sequence (two line ends and the *labelchar* character). Thanks to this simple notation, no delimiters or escape characters need to be introduced to the actual content (as is the case with

XML or JSON), regardless of its type. The only limitation is that the content cannot contain the ending sequence, which is hardly an issue in practice.

The default *labelchar* is colon (‘:’). It can hardly be expected in natural language texts and most programming languages to be found at the line beginning following an empty line. However there are cases where such awkward notation could be encountered—for instance, consider a course on batch programming, where a colon at the line beginning defines a label. For this reason, the *labelchar* can be redefined to any other character or even a sequence of characters using one of the predefined options (*endcontent*) specified in *lesson* metadata. The list of predefined metadata for *lesson* and *exercise* section contents is given in Table 3.

The two sections containing exercise description that is to be presented to the student (*intro* and *task*) must allow the instructor to include text having various formatting (standard text, text with emphasis, headings, programming instructions within sentences, blocked code examples) as well as illustrations or links to other resources. Although HTML provides all this functionality, its markup is verbose and there is number of other issues (see Sect. 3 of this paper) that make it tiring for hand editing. For this reason, Markdown [11] has been chosen as a default content type for the *intro* and *task* sections. It provides all the functions mentioned above

Table 3 Predefined metadata types

Metadata name	Content
Lesson section	
Language	Natural language used to teach the lesson (required!)
Programminglanguage	Programming language taught in the lesson (required!)
Endcontent	Character (or their sequence) marking the end of section content
Author	Author(s) of the content
Email	E-mail of the corresponding author(s)
Version	Version of the lesson
Created	Date of creating the first version of the lesson
Edited	Date of last edit of the current version of the lesson
Exercise section	
id	Exercise identifier. By default, exercises are numbered incrementally from 1 in their order of specification
Difficulty	Exercise difficulty level
Reward	Points or other virtual reward for completing the exercise
Precodevisible	Whether <i>precode</i> is visible to the student. By default, it is invisible
Postcodevisible	Whether <i>postcode</i> is visible to the student. By default, it is invisible
Blocked	Whether exercise can be taken by the student without completing others first. By default, it is
Unblocks	List of exercises that will be unblocked after completing this one

with an ease of marking down text by inserting but few characters instead of repetitive HTML tags.

The three sections containing source code (*precode*, *initcode* and *postcode*) as well as predefined input parameter values (*inputparameters*) do not require any formatting (as they are displayed in a code editor with a visual scheme of its own, or not displayed at all), and are represented in plain text which means there is no need for delimiters or character escape codes.

The remaining four predefined sections (*checksource*, *checkinput*, *checkoutput*, *checkerrors*) describe tests that the submitted exercise solution will have to be passed through primarily to determine its correctness, but also to generate hints that will help the student to solve the problems causing syntactical, execution or semantic errors, as well as improve possibly even a correct but suboptimal solution. Being aware of the issues with the use of mere regular expressions, a new, specially designed language (i.e., a sublanguage of SIPE) has been created for this purpose. The Solution Check Specification Language (SCSL), as it was called, is described in the following section (with more information provided in a forthcoming publication [12]).

Due to article length limitations it is impossible to present here a real-world SIPE-based exercise specification. However, to give a glimpse of how it looks like, a very simple exercise specification expressed in valid SIPE is provided below:

```

lesson: First things first

:exercise: Hello World!

:intro:
print command prints

:task:
Print "Hello World!"

:initcode:
"Hello World!"

:checksource:
req print => You missed the command.

:checkoutput:
req "Hello World!" => I cannot see "Hello World!".

:

```

5 Test and Feedback Specification

In SIPE, the automatic tests of the exercise solution, as well as the feedback generated after passing (or failing to pass) these tests, are specified, by default, using Solution Check Specification Language (SCSL). Each set of tests (i.e., the

content of a *checksource* section of a given exercise) is represented as a sequence of statements which are executed from top to down. Statement is defined as follows:

```
statement = [boolvar *wsp "=" *wsp]
            ["if" 1*wsp boolexp sep] [test 1*wsp]
            phrase [sep ">" sep feedback] *wsp
            [%d13] %d10 [%d13] %d10
sep = 1*wsp / *wsp [%d13] %d10 *wsp
```

Observe that:

- the result of a match can be stored in a boolean variable,
- a match can be executed on a condition defined with a specified boolean expression which can reference any boolean variable set earlier (this allows for, e.g., generating hints depending on a number of factors),
- the *test* can be either *req* or *forbid*, following the well-established practice of using both pessimistic (“the solution is wrong unless a certain structure is present”) and optimistic (“the solution is correct unless a undesirable structure is present”) rules [13, p. 22],
- *test* is optional, when it is not specified, the result of the pattern matching does not affect the solution correctness check, but the specified feedback can still be generated (if the pattern does not match; useful for improvement hints) and the result can be stored in a boolean variable (thus obtaining component result for a composite check),
- *feedback* is a possibly multiline text generated if the *phrase* matches (for *forbid*) or does not match (otherwise);
- the statement ends with double linefeed (i.e., an empty line); this allows for splitting long check specifications into multiple lines as well as increases the readability.

The actual pattern (or a sequence of patterns) to be sought is given in *phrase* element, consisting of one or several (separated with commas) patterns. If several patterns are given, they are considered by default as sequence, i.e. each pattern is matched from the position at the end of the match of the pattern preceding it. The following operators can be used to modify this behavior:

- *each*: match all the elements in any order,
- *none*: match none of the elements,
- *any*: match at least one of the elements,
- *select num*: match exactly *num* of the elements in any order, e.g.:
- *req select 1* (*a*, *b*, *c*) reports correct only if there is either *a* or *b* or *c* matched (but not a combination of these).

By default, the pattern is matched in the whole text (as defined by the SIPE section name; moreover, for *checksource* the match can only start at valid code, i.e. not inside comments or literal string constants). The match range can be limited to

the block defined after *position* operator, which defines the relation between the sought pattern and its context (*block*), and can be:

- *in*: the pattern will be matched inside the *block* (see below for details),
- *after*: the pattern will be matched no sooner than the block ends,
- *follows*: the pattern will be matched right after the block ends (only language-specific whitespaces are allowed in-between).

The *block* can be specified either by type only (the pattern will be matched in each block of that type) or also specified by *phrase* (the pattern will be matched only in those blocks of that type, which match such *phrase*). There are five types of *block*:

- *bracket*: matches within the (specified) kind of brackets (one of ‘()’, ‘<>’, ‘[]’, ‘{}’); in *checksource* section, brackets outside of language-specific valid code (e.g. inside comments or literal string constants) are ignored; the context *phrase* may include text preceding the opening bracket, but it must explicitly include the opening bracket and it cannot include any other brackets; the sought pattern may include the opening and closing brackets (as anchors),
- *line*: matches within the (specified) line (in *checksource* section, defined by language-specific boundaries, e.g. ‘\’ may disable line end in some languages);
- *compound*: only applicable to *checksource* section, matches in the (specified) compound statement, having language-specific boundaries; this is designed especially for languages which do not use brackets (like ‘{’ in C) to delimit compound statements, as e.g. Python; the context *phrase* matches from the beginning of the compound statement (i.e. including its header);
- *string*: matches in the (specified) string (delimited with quotes; in *checksource* section, defined by language-specific boundaries, e.g. ‘\’ may disable string end in some languages);
- *comment*: only applicable to *checksource* section, matches in the (specified) code comment.

For patterns returning multiple matches, their required quantity can be set using the following operators:

- *just num*: the pattern must match exactly *num* times; for instance, the statement `req just 3 /\b\d\d\b/` reports correct only if the number of occurrences of two-digit numbers is 3;
- *atleast num*: the pattern must match *num* or more times,
- *atmost num*: the pattern must match *num* or less times,
- *between num1 and num2*: the pattern must match at least *num1* and at most *num2* times,
- *multiply num*: the pattern must match $num \cdot x$ times, where x is any integer ≥ 1 ; for instance, the statement `req multiply 2 /\b\d\d\b/` reports correct only if the number of occurrences of two-digit numbers is even.

Also, additional requirements can be defined regarding the order of matched values:

- *distinct*: none of the fragments matched by the pattern can repeat
- (e.g. `req distinct /\d +/` reports correct only if there is at least one number, and, if there is more than one of them, they all have different values),
- *same*: each fragment matched by the pattern must have the same content,
- *incr*: each subsequent fragment matched by the pattern must have greater value (lexicographic order is used for non-decimals);
- *decr*: each subsequent fragment matched by the pattern must have smaller value (lexicographic order is used for non-decimals); for instance, the statement
- `req decr /\b\d +\b/` reports correct only if there is at least one number and, if there is more than one of them, they are ordered with decreasing value.

There are several ways of expressing a pattern available:

- *word* (e.g.: `print`): a sequence of non-space characters (from a limited set, mostly alphanumeric), intended to match single instruction names and identifiers; there is no need for delimiters other than space; strings identical to SCSL keywords cannot be specified this way;
- *words* (e.g.: `'x/2'`): a sequence of characters delimited with apostrophes; it has special properties: a space matches any whitespace sequence and a double quotation mark matches any language-defined literal string constant delimiter;
- *string* (e.g.: `"Good Bye"`): a sequence of characters delimited with double quotation marks; they match literally the given string (no wildcards, no special properties);
- *regex* (e.g.: `/p[a-z]*t/`): a regular expression (JavaScript flavor) delimited with slashes;
- *varref*: a name of variable preceded with '\$' character, it is an alias of a pattern defined earlier using the '->' operator (see below);
- *varvalue*: a name of variable preceded with '#' character, it matches the last value matched by the referenced pattern or returns no match if there was no prior match to set the value.

A single pattern can be represented using a combination of the ways listed above:

```
/[A-Za-z_][0-9A-Za-z_]*/' = '#val1
```

Each pattern can be labelled for later reuse using -> operator, e.g.:

```
/-?\b\d +\.\d +\b/ -> float
```

Table 1 in Sect. 3 gives more examples of SCSL expressions.

6 Implementation and Validation

The SIPE (including SCSL) has been defined in Augmented Backus-Naur Form (ABNF) [10]. The SIPE section parser and interpreter has been implemented directly in JavaScript by this author. Micromarkdown.js [14] is used as the Markdown converter. The JavaScript parser for SCSL has been generated automatically using APG [15] and served as the basis for its interpreter. Currently, only an interpreter tuned for Python source code (i.e. the exercise solutions are expected to be written in Python) has been developed, still it is configurable and can be easily adapted to any other programming language by replacing a set of callback functions.

The interpreter is being used as an engine for version 2.0 of the Python interactive course by this author [3]. All its code will be released as open-source as soon as the new course is published. At the moment of writing these words, the course is in the process of content conversion, improvement and extension which includes translation to SIPE of almost two hundred exercises and over one thousand tests.

The SIPE and its interpreter were validated using multiple exercises from the mentioned course [3]. This included all exercises from the initial part of the course and hand-picked exercises featuring tests based on especially complex regular expressions from later parts of the course. No major issues were encountered, and minor issues were used as a base for improvements in language specification and its interpreter.

7 Conclusion

In the paper, a new domain-specific language has been described, designed for specifying exercises for interactive programming courses. The language defines a simple structure of programming exercise specification, including metadata, exercise description, solution checking and feedback. It aims at conciseness, human readability and ease of use, especially the ability to edit the specification in generic text editors without nuisances.

For this reason, Markdown [11] has been chosen as the tool for exercise description whereas for the purpose of test and feedback specification, a new sublanguage, SCSL has been proposed [12]. It allows for a very concise definition of patterns typically sought for in exercise tests, and the resulting specification is far more readable than one based merely on regular expressions.

The language has been implemented in JavaScript and validated on numerous exercise examples. It will be soon published as an engine behind a new version of this author's interactive Python course [3].

References

1. Fernandez, J.L.: Automated assessment in a programming tools course. *IEEE Trans. Edu.* **54** (4), 576–581 (2011)
2. Swacha, J.: Scripting environments of gamified learning management systems for programming education. In: Peixoto de Queirós, R.A., Teixeira Pinto, M. (eds.) *Gamification-Based E-Learning Strategies for Computer Programming Education*, pp. 278–294. Information Science Reference, Hershey, PA, USA (2017)
3. Swacha, J.: Interaktywny kurs języka Python. <http://uoo.univ.szczecin.pl/~jakubs/kurs> (2016)
4. Queirós, R., Leal, J.P.: Making Programming Exercises Interoperable with PExIL. In: Ramalho, J.C., Simões, A., Queirós, R. (eds.) *Innovations in XML Applications and Metadata Management: Advancing Technologies*, pp. 38–56. IGI, Hershey, PA, USA (2013)
5. Zeng, C., Xie, L., Chen, G., Arikawa, S., Ishihara, Y.: OPECSS: an on-line programming exercise checking support system. In: *Proceedings of Conference on Educational Uses of Information and Communication Technologies*, pp. 199–205. Publishing House of Electronics Industry, Beijing, China (2000)
6. Joy, M., Griffiths, N., Boyatt, R.: The BOSS online submission and assessment system. *JERIC* **5**(3), art. 2 (2005)
7. Gerdes, A., Jeurig, J., Heeren, B.: An interactive functional programming tutor. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, pp. 250–255. ACM, New York, NY, USA (2012)
8. Hadiwijaya, R.I., Liem, M.M.I.: A domain-specific language for automatic generation of checkers. In: *International Conference on Data and Software Engineering*, pp. 7–12. IEEE, Yogyakarta, Indonesia (2015)
9. Swacha, J.: Implementacja interaktywnego kursu programowania w technologii webowej. *Studia Inform. Pomerania* **3**(41), 103–112 (2016)
10. Crocker, D., Overell, P. (eds.): *Augmented BNF for Syntax Specifications: ABNF*. RFC 5234. Network Working Group. <https://tools.ietf.org/html/rfc5234> (2008)
11. Gruber, J.: Markdown. <http://daringfireball.net/projects/markdown> (2004)
12. Swacha, J.: *Exercise Solution Check Specification Language for Interactive Programming Learning Environments*. Forthcoming (2017)
13. Goedicke, M., Striwe, M., Balz, M.: Computer aided assessments and programming exercises with JACK, ICB-Research Report, No. 28. <http://hdl.handle.net/10419/58160> (2008)
14. Waldherr, S.: Micromarkdown.js. <http://simonwaldherr.github.io/micromarkdown.js> (2017)
15. Thomas, L.D.: APG ... ABNF Parser Generator. <http://www.coasttocoastresearch.com> (2017)

Towards a Synergistic Combination of Research and
Practice in Software Engineering

Kosiuczenko, P.; Madeyski, L. (Eds.)

2018, VIII, 221 p. 61 illus., Hardcover

ISBN: 978-3-319-65207-8