Exercise Solution Check Specification Language for Interactive Programming Learning Environments

Jakub Swacha

Institute of Information Technology in Management, University of Szczecin, Szczecin, Poland
jakubs@uoo.univ.szczecin.pl

Abstract

Automatic checking of the correctness of students' solutions of programming exercises for generating appropriate feedback is a necessary component of interactive programming learning environments. Although there are multiple ways of specifying such a check, ranging from mere string patterns to code written in general-purpose programming language, they all have their deficiencies, with the check specification being too verbose, too complicated, difficult to reuse, or very limited in its expressive capabilities. In this paper, a new language designed especially for this purpose is described. It provides both extension and replacement for RegEx-based pattern specification so that checks typical for programming exercise verification can be expressed in a concise and highly-readable manner.

1998 ACM Subject Classification D.3.2 [Language Classifications] Specialized Application Languages

Keywords and phrases automatic programming exercise solution verification, source code pattern specification, RegEx extension, RegEx alternative

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.6

1 Introduction

Learning programming is difficult (see [1] and works cited therein). A highly important element of this process is letting the students practice with writing their own code by providing them with adequately chosen programming exercises. It may lead to substantial learning progress if only the students receive relevant feedback after they submit their solutions of the exercises. In traditional learning environments, checking the students' solutions and giving them feedback belongs to the instructor. In interactive programming learning environments, most of this process is automated, so that the instructor can focus better on other types of teaching activities.

The downside is that the instructor who is preparing an exercise for students not only has to conceive it and write its description (the aim and rules, possibly also expected results), but also specify the automatic checks and feedback generation rules. From this author's experience, based on the development of an interactive course of Python [8], if the course is intended for small student groups, the overall time spent on specifying the checks is greater than time spent on traditional checking of the exercises. Much of the reason for it lies in the deficiencies in the form of specification of the automatic checks (see the following subsection). An obvious solution for this problem would be to use a better form of specification, designed especially for this purpose and thus free of the most frequent and onerous nuisances. The goal of this paper is to describe such a solution, in a form of a domain-specific language.

6th Symposium on Languages, Applications and Technologies (SLATE 2017).
Editors: R. Queirós, M. Pinto, A. Simões, J. P. Leal, and M. J. Varanda; Article No. 6; pp. 6:1–6:8

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Problem setting

The most basic automatic checking of programming exercise solution takes into consideration only the output it produces after its execution (*black-box testing*). As the exercise description may impose restrictions on the programming language constructs or functions allowed in the solution, or require to apply certain constructs, functions, algorithms, programming techniques or design patterns, also the source code of the solution has to be checked to verify that, in addition providing valuable input for generating meaningful feedback to the students. Moreover, interpreter syntax check and execution environment error messages may be checked to transform them into a form that would be related to the actual exercise description and more intelligible to the students.

There are various ways that can be used to specify such checks [5, p. 44]. In the case of interactive programming learning environments (with limitations of their own), the two most obvious alternatives are to use a general-purpose programming language (not necessarily the language the solutions are written in) or RegEx-based patterns [9]. The general-purpose programming languages are designed for far more than exercise checking, therefore the check code, in all but trivial examples, is somewhat verbose, and it is not easy to tell after a single look at it what is actually checked. It makes them difficult to reuse and may cause a need for translation when transferring the exercises to another learning environment. Moreover, the actual match patterns are often expressed using RegEx expressions embedded in the general-purpose programming language used, which makes the issues specific to RegEx (see below) still valid.

RegEx (short from regular expressions, though it extends regular languages as known from theory [3]) is a lingua franca of text pattern specification in today's computing world. Nonetheless, as Lowell D. Thomas stated, the RegEx-defined patterns are often "hard to read, even harder to write, and hard to maintain" [11].

The intrinsic issues with RegEx readability are significantly augmented by the type of content that is searched through (i.e., especially, program source code) and the character of patterns to be matched. Selected examples of those are discussed below.

For instance, many programming languages allow optional whitespaces. Accommodating RegEx expressions for it results in single spaces being replaced with wildcards, obfuscating the pattern specification, e.g.:

```
/\(\s*limit\s*=\s*50\s*\)/
```

Similar issues are due to alternative notation of quotes (single or double) or certain operators (e.g. `<>` and `!=`) allowed in some languages (e.g. Python).

RegEx defines a number of special characters (e.g. `.`, `(`, `)`, `[` or `*` which have to be escaped in pattern specification. The same characters are very frequently used in source code of most programming languages, which leads to obfuscation of the pattern specification by insertion of backslashes, e.g.:

```
/\*\*|f\.a\s*\*f\.a/
```

The problem grows (along with the number of inserted backslashes) when the checked source code itself is expected to contain backslash-escaped characters (due to the requirements of the programming language notation), e.g.:

```
/1\.\.\.\\n\\t2\.\.\.\\n/
```

Most of the patterns looked for in the solution source code are expected to match valid program instructions, but they can as well match within literal string constants and comments.

In order to overcome that, the pattern actually sought has to be prepended with another one which ensures invalid code is ignored, increasing the total pattern complexity. The example below is intended for Python and it does not even cover multi-line strings and escaped backslashes:

```
/^(?:[^#"']*(?:'(?:[^']|\\')*'|"(?:[^"]|\\")*"))*[^#"']*\bfor\b/
```

Moreover, the patterns are often used to check the solution's structure, i.e., that specific code is found within specific logical block (e.g. an `if` statement). This requires a sophisticated RegEx in the case of languages forming logical blocks by indentation (such as Python), e.g.:

```
/if\s+x\s*<\s*0\s*:(?:\s*|\n(\s+)(?:.*\n\1))x\s*=\s*-\s*x/
```

The exercise solution often has to produce a specified, exact number of results. RegEx expressions checking non-occurrence of specific elements look awkward and contribute to low readability of pattern specification, e.g.:

```
/^(?:[^i]|i[^f]|\Bif|if\B)*\bif\b(?:[^i]|i[^f]|\Bif|if\B)*$/
```

A similar problem is when the exercise solution has to produce results in a specified order (e.g. sorted ascendingly). Ensuring that using RegEx expression is a nuisance, as it requires explicit listing of all the possible expected values, e.g.:

```
/^\D*(?:0\D+)*(?:1\D+)*(?:2\D+)*(?:3\D+)*(?:4\D+)*(?:5\D+)*
(?:6\D+)*(?:7\D+)*(?:8\D+)*(?:9\D+)*(?:10\D+)*$/
```

There is even bigger problem if more than one of alternative patterns has to be found in the source code, in unknown order. The example below is for an exercise which expected the student to use at least two different parameter orders when calling a function taking three parameters:

```
/\(\s*(?:(?:a_1\D+)?1\D+\d\D+(\d)[\s\S]+\(\s*(?:(?:a_1\D+)?
1\D+\1|\D+[23])|\D+([23])\D+(?:a_1\D+)?(\d)\D+(\d)[\s\S]+
\(\D*(?:a_1\D+)?(?:\2\D+\4|(?:\3|\4)))/
```

3 Related work

The unreadability of RegEx prompted search for alternative ways of pattern specification. A good example of such is *apg-exp* [11], using a superset of Augmented Backus-Naur Form (ABNF) [2]. The most notable extensions of the original ABNF are the positive and negative look ahead/behind operators which make specification of context-relevant patterns easy. Although *apg-exp* brings a visible improvement in readability of pattern specification, it is hardly concise. And while it fits well with programming language syntax definitions which are usually expressed in some form of BNF, *apg-exp* was designed as a general-purpose replacement for RegEx and thus provides no direct shortcuts for exercise solution checking.

Looking at the existing work in this area, however, there are no solutions that fit exactly such needs. The specification format for programming exercises, PExIL (Programming Exercises Interoperability Language) defines elements (within *specification* element) only for output check and feedback specification [6]. The automatic assessment solutions that transform solution code into abstract syntax trees or graphs use standard query languages, respectively, XPath and GReQL, to define the checks [7].

Perhaps the most promising of the existing solutions is the domain-specific language proposed by Hadiwijaya and Liem for an "automatic generation of the output, input, and

source code checkers" [4]. However, only the output- and input-checking is based entirely on the syntax native to that language, whereas source code checking (which demands the most sophisticated checks) relies on RegEx expressions.

4 Language Specification

4.1 Text pattern specification

The proposed language is designed as both a replacement and an extension of RegEx. Regarding the latter, it introduces a number of textual instructions and operators that allow to specify additional requirements for the specified patterns to match. Regarding the former, it provides four ways of pattern notation:

- `word`: a sequence of non-space characters (from a limited set, mostly alpha-numeric), intended to match single numbers, instruction names and identifiers; note there is no need for delimiters other than whitespace;
- `words`: a sequence of characters delimited with apostrophes; it has special properties: a space matches any whitespace sequence, a double quotation mark matches any language-defined literal string constant delimiter, code comments within matched text are ignored, it may contain variable references (they have to be delimited on both ends with `$` or `#` characters - the difference is explained below);
- `string`: a sequence of characters delimited with double quotation marks; they match literally the given string (no wildcards, no special properties);
- `regex`: a RegEx expression (JavaScript flavor) delimited with slashes.

4.2 Referencing and combining patterns

All the available pattern notations can be used interchangeably, and even combined to form compound pattern expressions, as well as labeled for later reuse using `->` operator, e.g. the following code will look for decimal digits after `width =` , and label the numeric pattern as `$width`:

```
'width = ',/\d+/ -> $width
```

The labeled patterns can be referenced both by definition (to look for similar patterns elsewhere) or by value (to look for repeating occurrences of the value matched by the pattern earlier), e.g. the following code references the pattern defining variable identifiers allowed in Python (`$pyvar`, assumed to be defined earlier; not a part of the specification language) and the value last matched by pattern `$width` (also assumed to be defined earlier):

```
'for $pyvar$ in range ( #width# ) :'
```

Note the spaces in the above listing will be matched by any whitespace combination.

The patterns can be combined using the following operators (and round brackets):

- `seq`: match all the elements in the given order (also applies if no operator is specified),
- `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.

For example, the following code will report match only if there is either *for* or *while*, but not a combination of these (as when using `any`):

```
select 1 (for,while)
```

4.3 Match quantity requirement specification

Using the following operators, the expected number of matches can be specified:

- `just` *num*: the pattern must match exactly *num* times,
- `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***x* times, where *x* is any integer greater than 0.

For instance, the following code will report match only if the number of occurrences of two-digit numbers is even:

```
multiply 2 /{\b\d\d\b}/
```

4.4 Order and uniqueness requirement specification

Using the following operators, additional requirements in case of multiple matches can be specified:

- `distinct`: none of the fragments matched by the pattern can repeat,
- `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 following code will report match only if there occurs at least one number and none of the matched numbers repeats:

```
distinct /\b\d+\b/
```

4.5 Position requirement specification

By default, the pattern is matched in the whole text (i.e., the input, output, or source code of the exercise solution). This can be changed by explicitly defining match position using one of the following operators:

- `in` *block*: the pattern will be matched inside the *block* (see below for details),
- `after` *block*: the pattern will be matched no sooner than the block ends,
- `follows` *block*: 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 pattern (the sought pattern will be matched only in those blocks of that type which also match the context pattern). There are five types of blocks defined:

- `bracket`: matches only within the specified kind of brackets (one of `()`, `<>`, `[]`, and `{}`); in source code, brackets outside of language-specific valid code (e.g. inside comments or literal string constants) are ignored;
- `line`: matches only within a single line (having source-language-specific boundaries, e.g. `\` may disable line end in some languages);

- `compound`: matches only in the specified compound statement, having source-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 only within a single string (having source-language-specific boundaries, e.g. `\"` may disable string end in some languages);
- `comment`: matches only within code comments.

Note that the context pattern can also have the position requirement specified, therefore the following code is correct and matches `5` as `range` function parameters in a line containing `=` within `if` statement block inside of `for` loop:

```
5 in bracket 'range (' in line '=' in compound if in compound for
```

4.6 Assessment result and feedback generation

The result of a match may trigger acceptance or rejection of the solution. The language defines two instructions serving this purpose:

- `req`: considers the solution incorrect and generates feedback if the pattern does not match;
- `forbid`: considers the solution incorrect and generates feedback if the pattern matches.

The appropriate feedback for the student is specified using the `=>` operator. For instance, the following code will report an incorrect solution and provide the specified feedback (`Incorrect number`) only if there is `1` or `2` or `3` (or any combination of these numbers):

```
forbid any(1,2,3) => Incorrect number
```

4.7 Conditional requirements

The result of a match can be stored in a variable. A match can be executed on a condition defined with a specified Boolean expression which can use three Boolean operators (`not`, `and`, `or`) and reference any variables set earlier. This allows for, e.g., performing checks and generating hints depending on a number of factors. For instance, the following code will require only one of the three specified patterns:

```
$v1 = 'x = -x' in compound 'if x < 0'
$v2 = 'x = -x' in compound 'else' follows compound 'if x >= 0'
$v3 = 'abs ( x )'
if not $v1 and not $v2 req $v3 => Calculate the absolute value
```

5 Implementation and validation

The proposed language has been defined using ABNF [2]. Its parser has been generated automatically in JavaScript using APG [12] and served as the basis for a proof-of-concept implementation of its interpreter, tuned for checking solutions written in Python – it can still be easily adapted to any other programming language by replacing a set of callback functions.

The language and its interpreter were validated using multiple pattern examples from real-world programming exercises from the course mentioned earlier [8]. No major issues were

encountered, and minor issues were used as a base for improvements in language specification and its interpreter. Among others, the eight RegEx patterns presented in Section 2 of this paper were successfully translated to the proposed language (note its conciseness and high readability):

1. `( limit = 50 )'`
2. `any('f.a ** 2', 'f.a * f.a', 'pow ( f.a , 2 )')`
3. `'1...nt2...n'`
4. `for`
5. `'x = -x' in compound 'if x < 0'`
6. `just 1 if`
7. `incr /b1?db/`
8. `distinct 2 each ( any ('( 1','a_1 = 1'), 2, 3) in bracket '(' )`

6 Conclusion

The instant feedback based on automatic checking of the correctness of students' solutions should be considered as a crucial component of every interactive programming learning environment. The specificity of such an intended use (large number of short exercises compared to small number of relatively long exercises used usually in programming contests) results in significant work effort needed to specify the acceptance and feedback rules even for a single course. The described deficiencies of existing forms of specification often turn even simple checks into verbose, complicated and difficult to reuse pattern definitions. The new language introduced in this paper provides ways to specify checks typical for programming exercise verification in a concise and highly-readable manner.

Currently, there is work undergoing on developing a new open specification format for interactive programming exercises which will feature the language described in this paper [10]. In the next step, the mentioned interactive Python course [8] will be converted to the new format, and the check code of all its exercises will be translated to the proposed language, which will confirm its usefulness and also provide statistical evidence on its conciseness.

References

1 Yorah Bosse and Marco Aurélio Gerosa. Why is programming so difficult to learn?: Patterns of difficulties related to programming learning mid-stage. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–6, 2017.

2 David H. Crocker and Paul Overell. Augmented BNF for syntax specifications: ABNF. RFC 5234, IETF, January 2008.

3 Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. Regex and extended regex. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata*, volume 2608, pages 77–84. Springer, 2003.

4 Ryan Ignatius Hadiwijaya and M. M. Inggriani Liem. A domain-specific language for automatic generation of checkers. In *International Conference on Data and Software Engineering*, pages 7–12, 2015.

5 Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Conference on Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

6 Ricardo Queirós and José Paulo Leal. Making programming exercises interoperable with PExIL. In José Carlos Ramalho, Alberto Simões, and Ricardo Queirós, editors, *Innovations in XML Applications and Metadata Management: Advancing Technologies*, pages 38–56. IGI Global, 2013.

7 Michael Striewe and Michael Goedicke. A review of static analysis approaches for programming exercisesd. In Marco Kalz and Eric Ras, editors, *Computer Assisted Assessment. Research into E-Assessment*, volume 439, pages 100–113. Springer, 2014.

8 Jakub Swacha. An interactive Python course: development and evaluation. Forthcoming, 2017.

9 Jakub Swacha. Scripting environments of gamified learning management systems for programming education. In Ricardo Queirós and Mário Pinto, editors, *Gamification-Based E-Learning Strategies for Computer Programming Education*, pages 278–294. IGI Global, 2017.

10 Jakub Swacha. SIPE: a domain-specific language for specifying interactive programming exercises. Forthcoming, 2017.

11 Lowell D. Thomas. An alternative to regular expressions: apg-exp, July 2016. SitePoint. `http://www.sitepoint.com/alternative-to-regular-expressions`.

12 Lowell D. Thomas. JavaScript APG, 2017. Coast to Coast Research. `http://www.coasttocoastresearch.com/docjs2/apg/index.html`.