Supporting Construction of Domain-Specific Representations in Textual Source Code

Tom Beckmann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Jan Reppien
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
jan.reppien@student.hpi.uni-potsdam.de

Jens Lincke
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
robert.hirschfeld@uni-potsdam.de

Abstract

Domain-specific representations (DSRs) allow programmers to view code in a manner better suited to the domain they are working in. Creating such DSRs, however, is not trivial as it requires knowledge of programming language internals to detect relevant patterns in the source code.

In this paper, we propose a workflow that facilitates the matching of syntax tree structures and creation of DSRs without requiring expertise in programming language internals. In our workflow, authors express the patterns they want using syntax familiar to them when possible and receive immediate feedback and means for simple discovery for the other cases. We demonstrate the feasibility of our workflow through a prototypical implementation and evaluate it through three case studies. Through this workflow, we aim to further support authors in creating views on source code that support their work better than text.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; *Application specific development environments.*

Keywords: domain-specific replacement, projections, visual programming

ACM Reference Format:
Tom Beckmann, Jan Reppien, Jens Lincke, and Robert Hirschfeld. 2024. Supporting Construction of Domain-Specific Representations in Textual Source Code. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3689488.3689990

PAINT '24, October 22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1212-8/24/10
https://doi.org/10.1145/3689488.3689990

1 Introduction

Domain-specific representations (DSRs) are views on parts of source code that better match the relevant domain than plain text [3]. These can support programmers in understanding and working with source code by better communicating properties of the code, such as constraints or actual values. Many editors support some degree of DSRs, also known as projections [18], ranging from a simple color picker inside source code to full visual editors of state machines [4, 10, 11, 14, 15]. From a user's perspective, a DSR appears as a visual user interface instead of or next to code. It may rewrite source code in response to the user's actions or merely visualize information.

To create a DSR, authors match a relevant pattern in the source code, define a view to be displayed, and may define actions that adapt the source code according to the author's interactions with the view [3, 9]. Important other concerns include versioning and sharing DSRs between authors or projects.

When new requirements surface in the code base's domain, creating new or adapting existing DSRs is not a straightforward task. Typically, creating a DSR will require the author to build an understanding of

- the structure of the host language's syntax tree,
- the editor's API, and
- a means to create user interface elements.

In particular when authors work with DSRs that are hosted on top of textual general-purpose programming languages, intricacies of syntax complicate the structure.

In this paper, we present ideas and initial implementations for a workflow that supports authors in creating their own DSRs on top of textual general-purpose programming languages. The workflow aims to reduce or even eliminate the requirement to understand language structures and offers guidance for matching source code to editing the source code with the user interface. Note that we assume that users

are generally familiar with constructing user interfaces in the respective editor's host environment. However, we still describe the implementation of some user interface widgets that facilitate the interface's interaction with the source code—in our own experience the most challenging part of the user interface code.

The vision our proposed workflow contributes to sees programmers become authors which create and adapt DSRs for the textual general-purpose code they are writing, to better support their own or teammates' understanding of the code. For instance, these DSRs may be used to remove failure points during edits, by placing constraints. Or, they may act as an inlined form of documentation, for example by creating dropdowns for enums or by pulling values from a specific runtime to provide better feedback [3].

This paper focuses on the creation of DSRs. Sharing, versioning, and conflicts of DSRs are future work. The remainder of the paper is structured as follows. In section 2, we add context to the challenge and describe related work. In section 3, we describe our own system. In section 4, we then describe several use cases using our system. We then discuss our system and describe future work in section 5 before concluding the paper in section 6.

2 Background and Related Work

In this section, we first briefly introduce the representation of syntax trees. Next, we give an overview of some examples of matching structures in syntax trees. We then briefly discuss challenges when going beyond matching to editing of source text. Finally, we describe some approaches to matching chosen by related work.

2.1 Concrete and Abstract Syntax Trees

Textual general-purpose programming languages are usually parsed and turned into a syntax tree for further processing by the language ecosystem. These syntax trees offer a reliable way for authors of DSRs to identify textual ranges of interest: the syntax tree approximates the same structures that authors would think of in terms of constructs they formulate while programming.

However, they typically use terminology that authors are not necessarily familiar with, include details that they may not care about, and do not directly communicate higher-level structures that authors may expect. As an example, take the default JavaScript parser[1] written for the Tree-sitter parsing framework[2]. A `const` or `let` assignment gets parsed into a "lexical declaration" node – a term that programmers may not necessarily use. Further, this node does not directly include the value it is assigned to. Instead, it includes an arbitrary number of "variable declarator" nodes, separated by commas – while programmers will likely know that declaring multiple

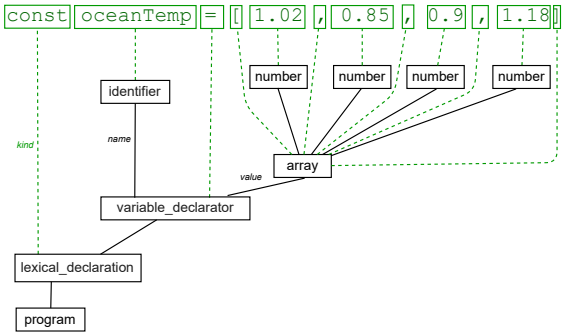


Figure 1. AST, CST and Text: the nodes that are only in the concrete syntax tree are shown in green, with the abstract nodes in black. As the black nodes communicate structure, while the green nodes communicate content, it depends on the use case whether they are of relevance.

variables in one declaration statement is possible, they may not always expect or want to handle it, if they are merely looking for a simple assignment to a value. Lastly, Tree-sitter differentiates between named and unnamed nodes: typically, named nodes are those that are more important on a semantic level, such as an identifier or an expression statement, and unnamed nodes are syntactic details, such as a semicolon. However, in case of the "lexical declaration", the parser designers chose for the `let` or `const` to be an unnamed node – an information that may be important to programmers in some scenarios.

Two types of syntax trees exist the concrete syntax tree (CST) and the abstract syntax tree (AST), as illustrated in Figure 1. A concrete syntax tree (CST) includes all information that can be obtained from the textual source, such as whitespaces, delimiters, or keywords, whereas an abstract syntax tree omits most of these and focuses on the semantically most meaningful structures, such as identifiers, control flow constructs, or declarations. Consequently, to go from a CST to an abstract syntax tree (AST), we want to remove information that may be considered secondary, for example whitespace. To facilitate this, some parsers communicate relevance of the syntax tree node, such as Tree-sitter's information on whether a node is named. For example, an entire `if` condition statement remains as a node in the AST, but the parts that it is made up of, the keyword, whitespaces, or braces, which are part of the CST, are omitted in the AST.

2.2 Finding Structures in Syntax Trees

The first challenge facing authors of DSRs is capturing the relevant parts for the replacement from the syntax tree. In this section, we will evaluate a selection of possible approaches for finding a pattern in the CST. We selected the approaches to demonstrate a broad range of possibilities. As a running example throughout the paper, we will try to capture colors

[1] https://github.com/tree-sitter/tree-sitter-javascript/
[2] https://github.com/tree-sitter/tree-sitter

defined as a JavaScript object with a field for the red, green, and blue color channels:

```
const color = { r: 0.54, g: 0.8, b: 0 };
```

Parsed by Tree-sitter, this results in the following tree:

```
program
  lexical_declaration
    variable_declarator
      name: identifier
      value:
        object
          pair
            key: property_identifier
            value: number
          pair
            key: property_identifier
            value: number
          pair
            key: property_identifier
            value: number
```

Indent shows a parent-child relationship, an identifier with a colon signifies a child that is at a specific named field. We want to capture the `object` and if possible the values for each field.

Regular Expressions. When faced with the challenge of matching a pattern from text, regular expressions are an often used tool. Matching the above example can be achieved using this regular expression:

```
\{ r: ([0,1]\.?\d*), g: ([0,1]\.?\d*), b: ([0,1]\.?\d*) \}
```

Regular expressions are a well-known technique with a large ecosystem for authoring and testing. Since they operate directly on the input string rather than the syntax tree, no knowledge of the underlying language is required. However, regular expressions are limited and exceptionally brittle for a complex use case such as this one. For instance, the above regular expression assumes an exact layout of whitespaces. While this can be adapted, it significantly complicates the expression, as most parsers allow whitespace between every single token. Similarly, parsing balanced parentheses, a very common construct in textual programming languages, using regular expressions requires special extensions.

jq. The command-line utility jq exposes a terse language for matching and transforming JSON objects. The output of a parser can be trivially mapped into JSON. For the following example, we choose a JSON object structure with a field for the node's `type` and a list of `children`. A query in jq for our example can look as follows:

```
jq '..
  | select(.type? == "object" and
    (.children | length) == 3).children
  | select(.[0].text == "r" and
    .[1].text == "g" and .[2].text == "b")'
```

Here, the author is required to understand internals of the language, such as the type of the object and its composition.

Imperative. Given the parse tree, a straightforward way to find the desired structure is through a depth-first search that imperatively checks properties of the current node and its surroundings. While this approach is versatile and can be adapted to match any structure in the tree, it also requires internal knowledge about Tree-sitter and CSTs in general. The results of this approach tend to be long blocks of code, which depending on the complexity might be hard to comprehend for someone who wants to make changes to the DSR. Below is an example in JavaScript:

```
function matchRecur(tree, captures, cond) {
  if (cond(tree)) captures.push(tree);
  return tree.children.forEach((it) =>
      matchRecur(it, captures, condition));
}
const captures = [];
matchRecursive(tree, captures, (tree) =>
  tree.type == "object" &&
  tree.children.length == 3 &&
  tree.children[0].text == "r" &&
  tree.children[1].text == "g" &&
  tree.children[2].text == "b"
);
```

Tree-sitter Queries. As in other contexts like relational databases, some specialized query languages for syntax trees exist. Tree-sitter itself offers a declarative language for matching syntax tree patterns. A query consists of a tree pattern written in S-Expressions with optional checks for fields of nodes. Nodes that match can be captured by marking them using a symbol with an "@"-sign. To obtain the name of the assignment to an array for the example above, we can use the following query:

```
(object
    (pair
        key: (property_identifier)
            @rk (#match? @rk "r")
        value: (number) @r)
    (pair
        key: (property_identifier)
            @gk (#match? @gk "g")
        value: (number) @g)
    (pair
        key: (property_identifier)
            @bk (#match? @bk "b")
        value: (number) @b)
) @color
```

Consequently, to create a query with the Tree-sitter query language the author has to know the specific names of the language's syntax nodes the authors of the parser implementation have chosen.

AST-Grep. AST-Grep trades some versatility compared to the tree-sitter query language for a much easier and faster query creation. Instead of letting the author construct the pattern, AST-Grep generates the pattern from code in the same language as the CST by using Tree-sitter. Special syntax within the code used to construct the pattern allows for parts to be matched. This allows the author to just copy the code that should be matched, and then replace the generic parts of it. On this basic level the author just writes code and only needs very little if any knowledge about tree-sitter or CSTs.

```
{r: $R, g: $G, b: $B}
```

Gremlin. Since trees are a subset of graphs, the syntax tree can be translated into a graph database. These databases usually come with their own query language(s). Since these languages are designed for graphs they cannot leverage the special properties of trees. Still, it is possible to use them to capture the necessary parts from the syntax trees. With Gremlin as query language and Apache TinkerPop as database, a possible query could look like this:

```
g.V().hasLabel('object').as('o').
 where(
  out('child').hasLabel('pair').as('p').
   out('key').hasLabel('property_identifier').
   values('text').is('r')
 ).where(
  out('child').hasLabel('pair').as('p').
   out('key').hasLabel('property_identifier').
   values('text').is('g')
 ).where(
  out('child').hasLabel('pair').as('p').
   out('key').hasLabel('property_identifier').
   values('text').is('b')
 ).select('o')
```

Similar results can be obtained by using Neo4j as database with Cypher as a query language.

Matching of simple patterns like the running example of this section is achievable with all approaches, with varying complexity. AST-Grep is the only of the presented query mechanisms that does not require the author to know intricacies of the language but also constrains matching to entire subtrees, unless its more complicated YAML interface is used.

2.3 Editing Syntax Trees

A DSR may not merely want to communicate information but also make it editable. For example, a color picker or a dropdown for enum values will need to change the source code in response to interaction with its user interface. These modifications of source code must be performed in a manner that maintains a valid source tree, which, depending on the language, can be challenging. Aspects to be considered include:

- When updating the value of a JavaScript string, the DSR must make sure that quotes are correctly escaped.
- When inserting into an array, a new comma may have to be added, unless a trailing one was already present.
- When changing an expression, precedence rules may require adding parentheses.
- Users may wish for the changes to be performed according to their formatting preferences, e.g., indentation, line length limits, or brace positions.

With these constraints, DSR authors have to anticipate user input well and accommodate the quirks of their host language's syntax. A seemingly simple synchronization between a text field and a string may become a challenge if the host language does not support multi-line strings.

In a projectional editor, where syntax is merely a visualization of the structure, these modifications are trivial. In text editors, authors have to come up with their own, often complicated, logic to reliably modify the source text to incorporate their change and still yield a valid tree.

2.4 Related Work

Several systems and approaches exist that facilitate creating DSRs. MPS [17] is an IDE for language workbenches [7] that include visual elements, such as state charts or machines, as for example seen in mbeddr [15]. MPS, as a projectional editor, is fully visual. Users define composable languages and users explicitly invoke a construct from a language while authoring. As such, a matching step over a syntax tree is not needed.

DSRs can be considered a form of term rewriting [12]: we detect patterns in source code and transform matched pieces of code into a different view. Unlike term rewriting for meta-programming, the transformation does not have an impact on runtime but is a bijective, temporary mapping that only occurs in the editor view. From an end-user perspective, DSRs may appear akin to embedded domain-specific languages (DSLs), where a DSL is embedded within a host language. DSLs can appear in textual environments [7] but also fully visual [16].

Moldable Tools and its implementation Glamorous Toolkit facilitate the construction of custom tools and custom views on objects [5] through the polymorphic invocation of well-defined methods for inspection. The matching process is thus inverse: objects communicate themselves how they can be visualized. Lorgnette [9] aims to make the definition of projections within code easier by allowing multiple types of patterns to detect relevant pieces of code—our approach could be used to facilitate formulating Lorgnette's syntax pattern. Similarly, Barista [11] is a structured editor that supports swapping out elements with visual alternatives.

Several systems, such as Moonchild [6], Interactive Visual Syntax [1], Larch [8] and LiveLits [13], demonstrate scenarios for what a replacement of source code in favor of a visual

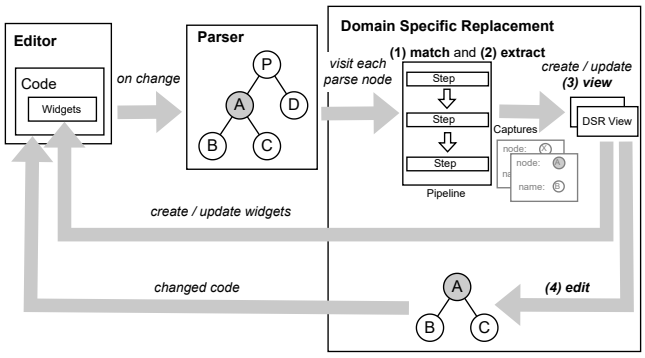


Figure 2. Update cycle in our workflow. The host editor signals changes to the code. The code is then parsed and the resulting parse tree is passed to the DSR system. Here, we update or create any DSRs and pass resulting widgets to the host editor. When a change in the view occurs, we inform the host editor of the new source text.

editing interface could look like. Matching in these systems occurs through custom code without specific framework support or is limited to matching literals.

3 Approach

We propose a workflow that requires less experience with the host language's structure and the editor's API for the creation of DSRs. To demonstrate its feasibility, we describe a reference implementation capable of the outlined functionality. Its implementation is accessible open source on GitHub[3].

Our reference implementation is designed for Tree-sitter grammars in a simple web code editor based on the popular CodeMirror[4] editor framework. The main challenges tackled in this paper, matching structures in syntax trees and editing source text, are not tied to these technologies and can be transferred to any other parsing library by adapting the methods for tree traversal.

3.1 Creating a DSR

We can describe the creation of a DSR as a process of four steps, embedded within a textual code editor.

3.2 Editor Interface and Update Cycle

The approach as we describe it is independent of a specific editor implementation, as long as it

- informs our system of changes to source files, and
- is able to display user interface elements on top of or near its source code.

The update cycle is shown in Figure 2. A suitable parser takes the changed source code and turns it into a parse tree for the matching step.

[3] https://github.com/hpi-swa-lab/sb-augmentation-builder
[4] https://codemirror.net/

This is where our proposed system comes in and the list of DSRs that are active. First, the author needs to formulate a **matching** pipeline to reliably find the desired language construct in the parse tree. As above, we use as a running example a JavaScript defining a color. The matching pipeline is run against all nodes in the parse tree to find matches.

Second, the author **captures** information relevant to their use case from the matched parse tree nodes. This could for instance be the textual value of the identifier or the textual range of the array.

Third, the author defines a **view**: a user interface to be displayed instead of, or near, the relevant text of the source code. In our example, the author may want to show a colored rectangle next to the color definition.

Finally, the user interface may not only display information from the source code but also **edit** the source code. For example, when clicking the button a color picker opens and, once a new color is chosen, it is written into the object.

3.3 Matching Source Code Structures

In section 1, we formulated as a goal to allow authors to match language constructs without the need to know language internals. To approach this problem, we implement two ideas:

- stick to the surface syntax of the language, familiar to the authors, as much as possible, and
- show the intermediate state of the matching process at all times and make it interactive, such that users can recognize, rather than recall, when the syntax of the language is not sufficient to constrain a match.

Authors begin the creation of a DSR by locating or formulating an example piece of source they would like to match, to facilitate showing the state of the matching process. In our reference implementation of the proposed workflow, authors select the relevant source code and hit a shortcut to create a DSR. Now, a DSR builder window opens, resembling the user interface shown in Figure 4. In the window, we include the example the author had selected, as well as a query that matches this example exactly.

Queries. The query mechanism we use is similar to AST-Grep queries, as described in subsection 2.2. Specifically, we extend it with optional and parent matches such that the patterns can be better integrated in the editing flow described in section 3.6. As an example, consider the following snippet of code defining an object literal for an RGB color:

```
const color = { r: 0.3, g: 0.1, b: 0.6 };
```

Our DSR author would like to match the object literal and capture the RGB values. A query returns either `null` or the syntax tree nodes captured by the wildcard tokens starting with a dollar sign.

```
{ r: $r, g: $g, b: $b }
```

Our DSR author wants to support a fourth, optional field that may be set on the object denoting the minimum and maximum value of each color channel:

```
const color = {r:3, g:1, b:6, range: [0,10]}
```

To match this expression, we can contain the optional element within special `(?...?)` markers.

```
{r:$r, g:$g, b:$b, (?range: [$low, $high]?)}
```

To capture this array, we can use another special token of the form `$_container`. This token captures its parent instead of denoting an element, allowing us to specify that we expect an array and also capture it.

```
{ r: $r, g: $g, b: $b, (?range: [$_range]?) }
```

If, instead, we want to match all channels at once, we could use the special `$$$rest` token that allows a container to contain arbitrarily many children and captures these.

```
{ $$$colorChannels, (?range: [$_range]?) }
```

The design of the query mechanism allows capturing a variety of elements while sticking to the host language's syntax that is familiar to users, as opposed to names of node types in the parse tree. The query language is not able to express all queries that may be of interest to the user. For example, imagine we want to ensure that each RGB channel field contains a number. In terms of the parse tree, in JavaScript the decimal numbers above would parse to nodes of type `number`, while for example in Ruby they would parse to nodes of type `float`. To support DSR authors to quickly figure out these internal type names, queries in our DSR builder are contained within pipelines that show outputs of matches, which we will describe next.

Pipelines. Queries in our workflow are steps in a pipeline, as seen for example in Figure 3. Each step produces an output that is then used as input for the next step. If the output of a step is `true` then the input of that step is its output. This allows to filter elements. If any step produces no output or `false`, the entire pipeline fails. In the DSR builder, outputs are visualized and thus allow DSR authors to understand what data flows through their matching system and how it is structured. For context, you may want to first look at a full pipeline, as seen for example in Figure 4. Steps of a pipeline may be:

arbitrary code Receives the input as `it` and may return any output. Multiple statements are allowed as well.

```
it.sourceString == "[1.02, 0.85, 0.9, 1.18]"
```

query Execute a query, as described above, on the current input, and return any captures as matches.

```
const $arrayName = [$_array]
```

queryDeep Execute a query on the current input and all its children, stopping on the first match, and return any captures for that match as output.

```
const $arrayName = [$_array]
```

extract Access a field of the input object and return it as output.

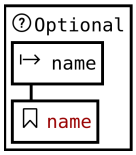
```
↦ arrayName
```

type Check that the input node's type matches.

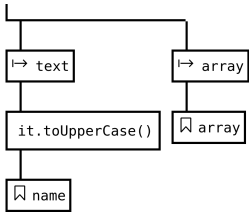
```
= array
```

In addition, several types of steps facilitate control flow within a pipeline:

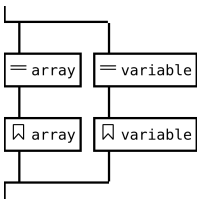
optional Execute a pipeline without failing the outer pipeline on failure.



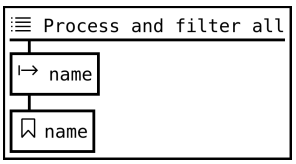
all Fork the pipeline: the current input is provided to all the pipelines specified in the all step.



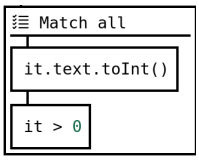
first Given a set of pipelines, execute the pipelines in order until one of them succeeds.



spawnArray Expect that the current input is an array and invoke the given pipeline for every element of that array. Elements that do not match are filtered out.



matchAll Works like spawnArray but expects a pipeline which results in `true` or `false`. The pipeline is executed on each element of the array. If any element does not produce `true` then matchAll returns `false`.



The graphical shapes as shown above are in fact DSRs, as further described in subsection 4.3. Consequently, authors can decide between the graphical editing mechanism shown above or a plain text editor, should they not need the feedback mechanism or other hints that facilitate editing. Below is the textual representation of the pipeline shown in Figure 4.

```
[
 query('"$content"'),
 (it) => it.content,
 capture("node"),
 (it) => it.text,
 first(
  [
   (it) => /rgb(( d+),( d+),( d+))/i.exec(it),
   (it) => ({
     r: parseInt(it[1], 10),
     g: parseInt(it[2], 10),
     b: parseInt(it[3], 10)
   })
  ],
  [
   (it) =>
   /#([a-z0-9]{2})([a-z0-9]{2})([a-z0-9]{2})/i
     .exec(it),
   (it) => ({
    r: parseInt(it[1], 16),
    g: parseInt(it[2], 16),
    b: parseInt(it[3], 16),
   }),
  ],
 ),
 all(
  [(it) => it.r, capture("r")],
  [(it) => it.g, capture("g")],
  [(it) => it.b, capture("b")],
 ),
]
```

Returning to our example, Figure 3 shows the construction of the pipeline that adds the additional check that all provided

Figure 3. Constructing the pipeline that matches an object where every value is a number. By right-clicking in the object explorers showing the result value of each step, the system offers contextually relevant actions to constrain or transform the values.

color values need to be numbers. To build this pipeline, we begin with the query from above, which is expressed in the DSR's target language, here JavaScript, with dollar-sign captures.

Using the output previews of the steps, we can explore the structure of the outputs to understand how we need to constrain or transform the values. This way, we are guided to some aspects of language internals, namely that the desired language internal name for the value nodes should be `number`.

3.4 Capturing

Once we have successfully matched a subtree, we want to capture the nodes and values relevant to build our user interface. For this purpose, the pipeline has a special step:

capture Marks the current input and stores it under a name for use by the view.

All values that have been captured are returned as the result of a successful match of a pipeline. If the pipeline match fails, no view is constructed. The set of captures of a successful match is then passed to the view.

To make updates more efficient, it is preferred to transform values into the final form that the user interface expects before capturing them, as opposed to extracting values while constructing the view. For instance, if we want to capture whether a string literal has the value `"active"`, we do not capture the string literal node, but instead compare its string contents and capture a boolean. This is in particular beneficial for performance, as it allows us to inform the view whether or not it needs to re-render by comparing if the previously captured values differ from the newly captured ones.

3.5 View

Now that we have captured a set of named nodes or values, the author can use them to construct the view. As already illustrated in Figure 2, the source code editor is responsible for informing our system when changes to a file occur. On a change, we re-run the pipelines of all defined DSRs and see if the captured values have changed compared to the previous render. If so, we provide these updated values and allow the user interface to re-render itself.

Our implementation allows the author to use any framework supported by the host editor for constructing the view. As described, our system provides the author with the output from the capturing step and allows the view to cause changes to the source code through our system's editing support.

The widgets authors can use to construct the view depend on the chosen framework of the implementation. If the host platform has a visual means to construct user interfaces, authors might again not need specific expertise.

To tackle some of the challenges we observed when constructing user interfaces, we suggest for the host platform to offer three built-in widgets:

- A *NodeList* widget allows authors to display a list of widgets that each correspond to a syntax tree node. The NodeList takes care of displaying remove and insert buttons between items in the list that change the source code accordingly, as described next in subsection 3.6.
- A *TextArea* widget allows editing the, optionally unescaped, contents of a syntax tree node. The means of unescaping text are also further explained in subsection 3.6.
- An *EditorPane* nests an instance of the host editor view in a widget and thus allows nesting editable source code expressions.

3.6 Editing

When the user interacts with the view, the intent is likely to edit the underlying source code. In the simplest case, the user might want to replace the textual contents of a number with another number through the view. For this, our implementation offers a `node.replaceWith(string)` call that replaces the text range of the given node in the source code with the given string.

During our own experiments with constructing DSRs, we observed a number of scenarios that complicate propagating edit operations to the code. Below, we describe how our system can accommodate these while hiding as many details concerning the language's structure and complications that arise from it from the DSR author.

Editing Lists. As an example, take a JavaScript array of the form `[1, 2]`, in which we want to append the number `3` at the end. Here, we need to also insert a comma before the number. In our reference implementation, we automate this process. The author can do a simple insert call of the form:

```
arrayNode.insert(text: "3", index: 2)
```

Given that call, the system searches the grammar definition of the container node for points where a variable number of nodes can be inserted. Once the system found such a point in the definition, we partial-parse [2] the nodes already contained in the container, meaning we take the nodes of the container and its definition in the grammar and identify to which grammar operators they belong. When we encounter a suitable repeating grammar operator, we begin counting the number of nodes already contained within, until the index the user requested is reached. Through a heuristic, we analyze the grammar operators within the repeat operator to find if the list is delimited. Once a delimiter is detected, we can see if there already is a trailing delimiter. With that information obtained, the system can finally deduce the exact position and string the system needs to insert to arrive at the final expression `[1, 2, 3]`.

Editing Textual Contents. We allow users to specify that they want to make the textual contents of a node editable in a text field to the user and specify a mapping for translating from source code to text field. As examples, picture the name of an attribute, `obj.attr`, or the contents of a string, `"my string content"`.

A straightforward mapping simply takes the source code as-is and displays it in the text field. A mapping for source code that is subject to escaping rules, requires a list of substitutions that should occur. For example, in a JavaScript string, a new line must be escaped as \n or a nested quote as \". Given the list of substitutions, the system then takes care of translating edits by following the set of rules. For instance, if the user types a quotation mark in a JavaScript string that is also delineated by quotation marks, the system

instead inserts an escaped quotation mark in the source code but displays only the simple quotation mark in the text field.

Another type of mapping optionally adds delimiters. This is relevant for example for JavaScript object keys: a definition such as `{a: 3}` is valid JavaScript, but the key requires quotation marks if it contains special characters such as `{"a-b": 3}`. The mapping takes care of inserting or removing the delimiters as needed.

Editing Optional Elements. We allow users to specify optional elements as described in subsection 3.3. Handling the presence and absence of optional elements would significantly complicate view code written by DSR authors. Fortunately, the query already provides all relevant information needed to automatically insert a missing element. For example, consider a query such as the following:

```
{ key1: $value, (?key2: [$_list]?) }
```

Here, we follow the Null Object pattern [19] where the system returns a proxy syntax tree node for the list that the author expects to find if it is missing in the code. If any edit operation affects the list, we first insert the entire optional element into the document and then proceed with the operation. Read operations on the missing element return an empty list. Consequently, the DSR author can write code as if the element was present and the system handles its creation if needed. The same applies for primitive values: `func((?$arg:false$?))` returns a captured node for the argument even if it is not there and returns `false` as its content for any read operations.

4 Evaluation

In this section, we show three brief case studies of DSRs, to demonstrate the range of tools we have constructed using our own system.

4.1 Color Picker

A commonly seen DSR is a color picker: for expressions that contain colors, a color picker or preview of that color is displayed instead of or next to the expression in code. In Figure 4, we show a pipeline that matches colors in hex notation `#ff9900`, as well as colors of the form `rgb(255,125,0)`.

We start by matching any string, extract its text, and see whether it complies with either of the formats we expect by using a regular expression. If so, we extract the color channels and store them in three separate captures. Capturing the whole regex match would have also been possible.

We also show the view construction code in Figure 4. Our reference implementation uses Preact for constructing user interfaces. We receive as inputs the captures and return a declaration of the view. When the color input changes, we simply replace the contents of the node containing the current color value with the new color.

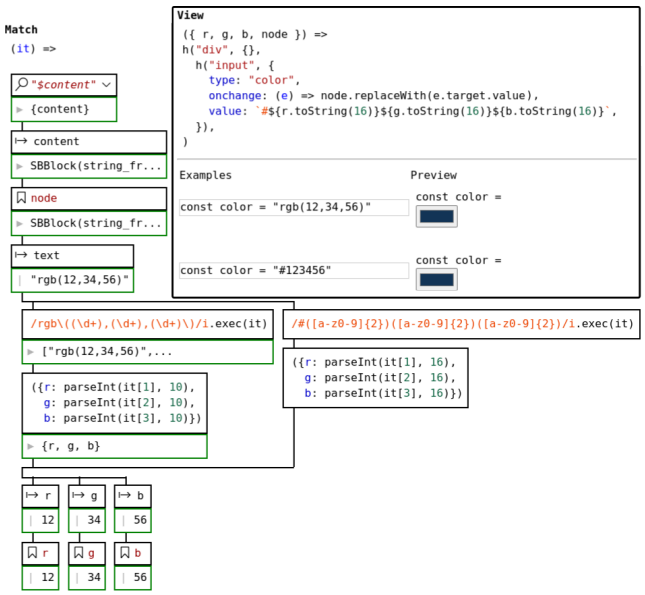


Figure 4. Pipeline, view, and examples that construct a color picker for colors in hex and in rgb() notation.

Below the view in Figure 4, we have created two examples. At the moment, the top example is active and is used to show the output previews in the pipeline. To the right of each example, we see a preview of the final user interface in an inline editor. The depicted code is all that is necessary to match and display the shown color picker. Depending on the host editor's API, the match and view code will need to be registered with the editor to be applied on load and to update when the user changes text in the editor.

4.2 2D Array to Table

In Figure 5 we find the pipeline to match nested JavaScript arrays where all the nested arrays have the same size. Here, we need four types of steps: query steps allow us to specify that we are looking for arrays; based on the feedback below each step, we added several field extract steps; we capture the number of columns and rows; and finally, a matchAll step containing an arbitrary code step verifies that the length of each of the nested arrays matches.

4.3 The Tool Itself

We bootstrapped the reference implementation of the presented workflow using the pipeline and view system itself. We edited the serialized, textual version of the pipeline instead of working with the user interface while it was still being constructed.

As seen in the excerpt of the first six cases in Figure 6, the user interface tends to overflow horizontally for large `all` or `first` steps. Notably, however, as in the previous case studies, no knowledge of language internals is required to formulate

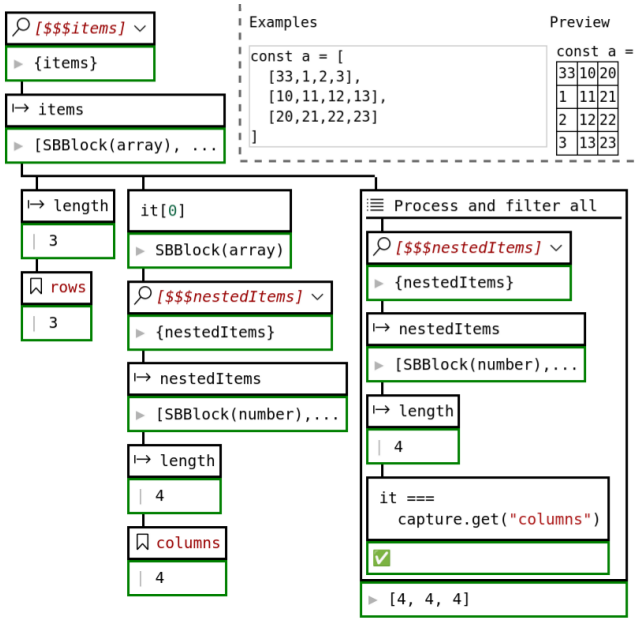


Figure 5. A pipeline matching two-dimensional JavaScript arrays where all nested arrays have the same size. We begin by matching an array (note the square brackets) and extracting the items it contains. We then capture its length as our number of rows, and query and extract the number of columns from the first row. Finally, we iterate over all rows and verify that they are indeed arrays and all have the same number of columns as the first column that we previously captured. In the top right, we can see a text field with a 4-by-3 nested array containing numbers. Next to it is another text field where the nested array is replaced with a table containing all the numbers.

the pipeline. The pipeline is essentially one large `first` step that matches the different steps described in subsection 3.3.

5 Discussion and Future Work

In the following, we first discuss how well our proposed system matches our set goals so far, before outlining future work.

5.1 Supporting Creation of DSRs

In section 1, we set the goal for our proposed workflow to allow authoring DSRs without knowledge of the host language's syntax tree, and while supporting authors with the editor's API and user interface.

In terms of knowledge of the language's internals, our system chooses to let authors start with familiar syntax and offer exploration means to learn what further constraints are needed. These further constraints will often concern language internals. As our tool's user interface offers contextual actions and supports the discovery of these internals, the chosen design may be a compromise to give authors full flexibility while still not requiring them to explicitly learn about the language internals. A user study will need to be conducted to better understand the effect.

In limited, informal user testing, we noticed a tendency for users to want to skip the pipeline system altogether for very small queries, e.g., just matching a number. Typically, the queries then started to grow, which led the users to then appreciate the feedback mechanism of our tool. These insights may indicate that the entry to start creating a DSR is not sufficiently easy yet.

Concerning scaling, especially in Figure 6 we see that the visual approach to constructing pipelines in our reference implementation may run into issues. While the code for the same purpose is also unwieldy, we can use functions to better compartmentalize concerns. Creating sub-pipelines is thus also a likely next step. Further, we could consider introducing step types for better management of layout and structure. For example, Kotlin has an `also` operator that allows running some code without changing the object that traverses the pipeline further.

The editor's API comes into play for transforming or filtering in pipelines beyond type matches or accessing children. For this purpose, a future design could investigate augmenting the output previews to not just make fields visible but also API methods. In addition, authors need to know the API to edit the source code. Here, we offer the built-in widgets that bind values directly to the view. For custom mappings, however, we currently offer no additional support.

Finally, for constructing the user interface, we are assuming that the author is familiar with the host platform's user interface construction means. As described in subsection 3.5 and subsection 3.6, our reference implementation offers authors widgets that facilitate editing source code.

From a user interface point of view, the graphical interface allows us to more easily communicate available actions and provide feedback. On the flip side, graphical interfaces tend to be associated with cumbersome, mouse-heavy interactions, especially in an otherwise keyboard-centric setting. We believe that adapting the user interface to remain graphical but be navigable via keyboard is possible and should satisfy both ease of use and a desire for the efficiency of a keyboard.

5.2 Matching Intent

As described, our goal is to remove the need for authors to have a deep understanding of language internals. However, programming language syntax often encodes various language-specific peculiarities that complicate matching. Below are some examples from Tree-sitter's JavaScript parser:

1. We get an extra `formal_parameters` node for `(a) => {}` when compared to `a => {}`. (Note the parentheses surrounding the parameter `a`).
2. A pattern like `obj.$key` would not match `obj["field"]`.

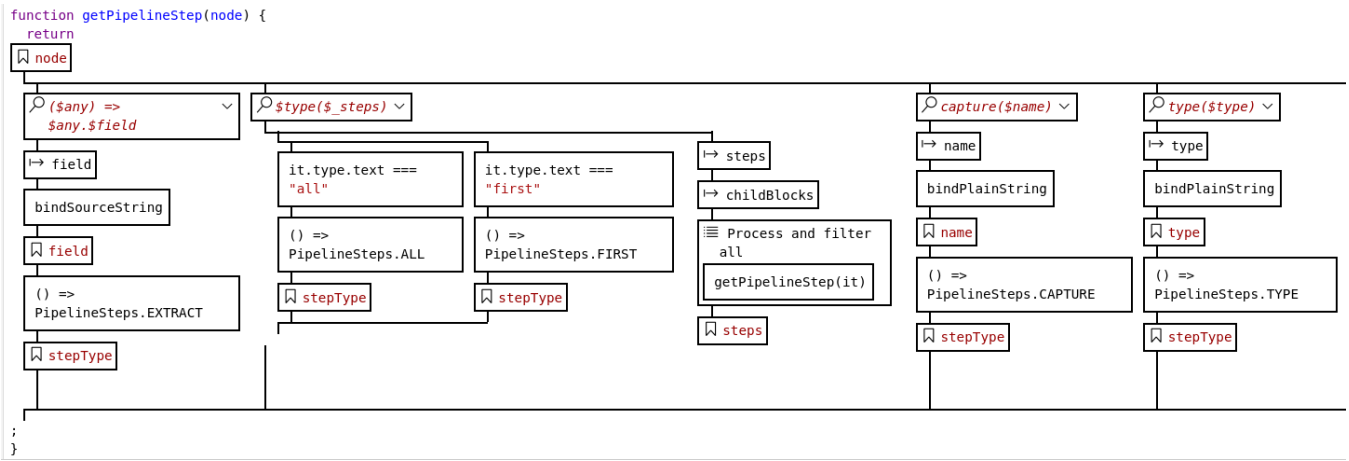


Figure 6. Excerpt of the first six steps for the pipeline used to construct our workflow's reference implementation.

3. `function a() {}` parses to a `function_declaration` node, whereas `function() {}` parses to a `function_expression` node.

While authors likely mean to simply search for desired language construct, these differences in the shape of the syntax tree would let their query fail for no apparent reason—until they compare the internal structure of the two syntax trees. In future work, these pitfalls would likely need to be addressed as a post-processing step for the grammar. The grammar could be annotated to note that some language constructs may be considered identical.

As another challenge, fields in object literals in JavaScript may be specified in any order. A query pattern to match against an object literal will, however, only match if the order is as specified in the pattern. As a solution, a future version could allow users to specify that order of children in a container does not matter. Alternatively, grammars for languages could be annotated to indicate which language constructs have no concept of order.

Similar challenges arise for language constructs that make heavy use of composition. As an example, consider the typical way of constructing objects in Smalltalk:

```
MyObject new
    field: 1;
    anotherField: 2.
```

Here, we receive a cascade node instantiating MyObject and setting two fields. The author wants to match the entire instantiation and capture any fields that are set. If the fields are optional, the following two instantiations would be equally valid:

```
MyObject new.
MyObject new field: 2.
```

The first of which is a unary message send node, while the second is a keyword message send node, each containing the instantiation. The author will likely be looking for the abstract construct "instantiations of MyObject", which requires three different queries for each of the above scenarios. A solution may be to offer built-in matching steps such as `smalltalkObjectInstantiation('MyObject')` that hide these variations. Authors may of course also write their own reusable steps to match patterns specific to their use of a framework or language.

5.3 Language Agnostic DSRs

A subset of DSRs is likely desirable to have in an unchanged form across languages or frameworks. As an example, a tabular display of arrays can be expressed in most languages and various frameworks have special constructors for creating two-dimensional data.

Here, the view remains the same across frameworks and languages. Only the matching, capturing, and editing steps vary. It would be desirable for authors to be able to specify that their framework or language also contains a notion of two-dimensional data literal and how it may be edited and for the table DSR to then automatically extend support accordingly.

We have investigated first steps through a `languageSpecific` step for pipelines that allows authors to list pairs of languages and steps. Finding an extension mechanism to allow extending the DSR without touching the original definition is future work, as it touches upon aspects of versioning and sharing.

6 Conclusion

In this paper, we present a workflow and prototypical implementation for building domain-specific replacements. The workflow's goal is to closely match the level of abstraction that programmers as authors of DSRs are familiar with from programming languages, without requiring knowledge of parser or language structure internals. We achieve this goal by making use of the language's syntax itself where possible and otherwise provide simple exploration means for authors

to identify just enough of the internals to build their DSR. In three case studies, we demonstrate that our approach works and identify further areas of improvement. Using our workflow, we hope to make it easier for authors to create their own DSRs and thus make their workflows better adapted to their problem domain.

Acknowledgments

This work was supported by the HPI–MIT "Designing for Sustainability" research program[5], and SAP.

References

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (nov 2020), 28 pages. https://doi.org/10.1145/3428290

[2] Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. 2022. Partial Parsing for Structured Editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/3567512.3567522

[3] Tom Beckmann, Daniel Stachnik, Jens Lincke, and Robert Hirschfeld. 2023. Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Cascais, Portugal) *(PAINT 2023)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/3623504.3623569

[4] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[5] Andrei Chis. 2016. *Moldable tools*. PhD thesis. Universität Bern.

[6] Patrick Dubroy. 2014. *Moonnchild*. Workshop on Future Programming (FP), SPLASH 2014. Retrieved 06 July 2023 from http://www.future-programming.org/2014/program.html

[7] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Comput. Lang. Syst. Struct.* 44 (2015), 24–47. https://doi.org/10.1016/J.CL.2015.08.007

[8] G. W. French, J. Richard Kennaway, and A. M. Day. 2014. Programs as visual, interactive documents. *Softw. Pract. Exp.* 44, 8 (2014), 911–930. https://doi.org/10.1002/SPE.2182

[9] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 71, 16 pages. https://doi.org/10.1145/3586183.3606817

[10] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 72, 18 pages. https://doi.org/10.1145/3586183.3606733

[11] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*, Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson (Eds.). Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[12] Ralf Lämmel. 2018. *A Suite of Metaprogramming Techniques*. Springer International Publishing, Cham, 335–397. https://doi.org/10.1007/978-3-319-90800-7_12

[13] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[14] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* (2019). https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/9

[15] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

[16] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting grammars into shape for block-based editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/3486608.3486908

[17] Markus Voelter. 2011. Language and IDE Modularization, Extension and Composition with MPS, In Generative and Transformational Techniques in Software Engineering IV, GTTSE 2011, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.). *GTTSE 2011* 7680, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[18] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

[19] Bobby Woolf. 1997. Null Object. In *Pattern Languages of Program Design 3*, Robert C Martin, Dirk Riehle, and Frank Buschmann (Eds.). Addison-Wesley.

Received 2024-07-15; accepted 2024-08-11

[5] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html