CEDALION's Response to the 2016 Language Workbench Challenge

David H. Lorenz[1,2] *
[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology, Haifa 32000, Israel
dhlorenz@cs.technion.ac.il

Boaz Rosenan[1,3]
[3]University of Haifa, Mount Carmel, Haifa 31905, Israel
brosenan@gmail.com

Abstract

This paper represents CEDALION's answer to the *2016 Language Workbench Challenge*. We provide solutions to 11 out of the 25 benchmark problems that were most relevant to CEDALION.

1. Introduction

CEDALION [7] is a programming language that combines *logic programming* [6] and *projectional editing* [5]. It is unique in the way projectional editing is integrated into the language, allowing developers to define new concepts with their own projections from within the language. In such settings, the concept declaration, its projection definition, semantic definitions, and usage are all done from within the same program. This is in contrast to *generative* language workbenches, in which a language is defined in one environment, and then the language definition is compiled to an editing environment used for developing target programs in that language.

1.1 Is CEDALION a Language Workbench?

CEDALION is in essence a programming language. Its source files are stored in a textual (Prolog-like) format, which could be edited using a text editor. However, the common and preferred way to edit CEDALION files is with the CEDALION WORKBENCH, an Eclipse-based IDE for CEDALION development.

The CEDALION WORKBENCH is a projectional editor that is integrated with the language. The projection of language concepts, markers around the code (e.g., error messages), and other forms of behavior (e.g., a context menu available for different AST nodes) are all derived from the program being edited, by consulting it at runtime.

Fowler, who coined the term *Language Workbench* [3], defines loosely what a language workbench is:

> "...a new class of software development tool, designed to build software through a rich environment of multiple, integrated, Domain Specific Languages." [4]

This definition seems to overlook programming languages such as CEDALION (a language workbench is said to be a *tool*), but the CEDALION WORKBENCH certainly complies with this definition.

Because the implementation of the CEDALION WORKBENCH is inseparable from the language itself, we refer to it simply as "CEDALION" throughout this paper.

1.2 Object Language

The object language on which we implement the challenge is a DSL named MINIPERATIVE, which provides features in CEDALION such as imperative variables,[1] commands (that are executed sequentially), input and output, etc. Most programming environments today are based on imperative languages and therefore these features are taken for granted in those environments, but CEDALION is a purely declarative programming language, so imperative features do not come for free, and have to be defined.

We designed MINIPERATIVE as an alternative to MiniJava [8]. We chose MINIPERATIVE over MiniJava because, on the one hand, it is simpler to implement; and, on the other hand, sufficient for illustrating a solution to the benchmark problems. Fig. 1 shows an example of code written in MINIPERATIVE.[2]

```
procedure fooProc ( X :: number , Y :: string )
    while not ( X :: number == 0 ) :
        • output Y :: string
        • X = X - 1 :: number
```

Figure 1: Example of a procedure written in MINIPERATIVE

* Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

[1] Imperative variables change their value over time, unlike logic variables, which are bound to a single value.

[2] The complete definition of MINIPERATIVE can be found on GitHub at `https://github.com/brosenan/minimperative`

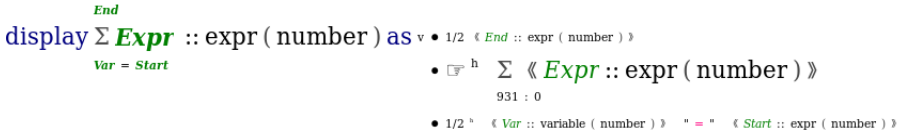


Figure 2: The projection definition of the *sum* concept

1.3 Outline

The rest of this paper corresponds to the three categories of benchmark problems, namely: *Notation* (Sect. 2), *Evolution and Reuse* (Sect. 3), and *Editing* (Sect. 4).

2. Notation

We address the following challenges dealing with the appearance of source code [2], including support for:

- *Mathematical symbols*: support for mathematical symbols in addition to textual notation (Sect. 2.1).
- *Metadata annotations*: annotation of program elements without changing their core meaning (Sect. 2.2).
- *Optional hiding*: hiding parts of the code, without losing the content, while retaining editability (Sect. 2.3).
- *Multiple notations*: alternative notations for the same language construct (Sect. 2.4).

2.1 Mathematical Symbols

The object language "should be extended such that users can use conventional mathematical notation such as sum symbols, fractions, and square root symbols" [2, §6.5.1]. Normal object language expressions "must be allowed inside the mathematical notations (e.g., in the sum index, the numerator or denominator of fractions, and under the square root)" [2, §6.5.1].

Assumptions This solution is based on the MINIPERATIVE object language.

Implementation The concepts *sum* and *div* were implemented, representing summation and division expressions, respectively. Both are numerical expressions in MINIPERATIVE.

The *sum* concept takes four arguments: a numeric variable, numeric expressions representing its initial and final values, and the expression to be evaluated. Its semantics is defined in two steps. First, the boundary expressions are evaluated. Then, a recursive evaluation of the sum occurs. The semantics is expressed as a new clause to the predicate *eval*, which defines the semantics of expressions in MINIPERATIVE. The concrete syntax for *sum* is given using the *projection definition* ("`display...as`" statement) in Fig. 2. To allow the Σ symbol to grow with the expression to its right, we use the *brackets* visualization concept to wrap the expression, where the left bracket is Σ and the right concept is null (Unicode 0). Brackets are the CEDALION equivalent to LaTeX's `\left` and `\right` macros. The Σ symbol and the expression are laid out vertically, with the lower bound and the variable name below them, and the upper bound above them. The bounds are displayed using smaller fonts, following the original mathematical notation.

Note how the left hand side of this projection definition is projected according to this very definition. This happens because the projection definition takes effect as soon as it is saved to disk for the first time. From that time on the CEDALION WORKBENCH will consult it when projecting the *sum* concept, even within its own projection definition.

The *div* concept is defined using the `/` operator already supported in MINIPERATIVE. Its projection definition (Fig. 3) consists of a vertical layout containing both arguments with a box element in between, responsible for the division line.

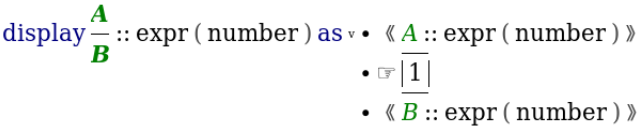


Figure 3: The projection definition of the *div* concept

In both projection definitions we want the middle element in the vertical layout (the Σ symbol with the expression, and the division line, respectively) to be vertically aligned with the rest of the line. This is achieved by using the *pivot* visualization concept, which selects an element in a vertical layout and tells CEDALION to align this element with other elements laid out horizontally with it.

Fig. 4 shows a usage example of the *sum* and *div* concepts.

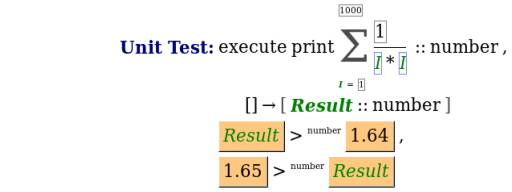


Figure 4: A unit test showing *sum* and *div* concepts in action in approximating $\sum_{i=0}^{\infty} \left(\frac{1}{i^2}\right) = \frac{\pi^2}{6}$.

Variants None.

Usability The problem with Σ and a division line is that they cannot be easily typed on a keyboard. For this reason, a user must be familiar with a textual name associated with each concept. This is similar to how LaTeX users who wish to present these notations need to use `\sum` and `\frac` to represent Σ and the division line, respectively.

We named these concepts *sum* and *div*, respectively. To enter a sum, users write "`sum`" (or some prefix of that) and press Ctrl+Space to get the completion menu. Then they choose the correct form (based on namespace and number of arguments) in case there are more concepts matching this prefix that are legal in this context. Finally, they fill in the blanks: variable name, bound expressions, and *sum* expression.

The process for a division line is similar, starting with the user typing "`div`" and Ctrl+Space. Note that although we chose the name `div` for the division-line concept, we can still help users proficient in LaTeX by adding the alias `frac` to the *div* concept. This way they could get the division line by typing either "`frac`" or "`div`".

Impact Both concepts were added without changing the DSL they extend or any other code that existed before them. They were added in their own file in a separate project dedicated to the LWC challenge.

Composability Expressions projected using mathematical notations compose (e.g., Fig. 4). Since both *sum* and *div* are numeric expressions that take numeric expressions as arguments, they can be used in concert, and combined with other mathematical notations they can form complex mathematical formulae.

Limitations Ideally, the *sum* concept should have used a grid layout, which would align the bounds to the center of the Σ symbol. As CEDALION does not currently support grid layouts, a (left aligned) vertical layout is the most reasonable available choice.

Similarly, the division line would have been better off with centered vertical layout. CEDALION's vertical layout does not (yet) support alignment, and therefore the numerator and denominator are left aligned.

Uses and Examples The concepts defined in response to this challenge are useful in order to make implementations of mathematical algorithms look more similar to their textbook origins. As a result, these implementations are easier to review and to modify by subject matter experts. Additionally, the *sum* concept is a declarative feature of an imperative language. The sum is calculated without modifying the state of the program (unlike the alternative of performing this summation imperatively, in a loop). This gives the user less room for error.

Effort (best-effort) Defining the *sum* concept took about 30 minutes, and the division line about 15 minutes. Both include declarations, projection definitions, semantic definitions, and unit tests.

Other Comments None.

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/notation-math.ced`

2.2 Metadata Annotations

"Users should be able to attach annotations to arbitrary program elements without affecting the program's core meaning. A full solution should allow the annotation itself to have structure enforced by the workbench" [2, §6.5.1].

Assumptions None

Implementation CEDALION supports annotations, which are concepts that are removed from the AST before it is evaluated by the interpreter. Any concept of the namespace *annotation* is removed from the AST when the program is loaded, and is replaced by its first argument, on which the process of removing annotations continues.

Fig. 5 shows the definition of an annotation named *author* which can wrap any concept (in MINIPERATIVE or any other), and preserves its type T. We defined its projection as the string "`@author`" followed by a string parameter written in a half-size font above the annotated code fragment. We give it the alias "`@authour`", to allow users to use this name (and not only "author" – its natural name) when inserting it to the code. Finally, the figure shows an example code fragment (inside a yellow *sandbox* concept, a place where an arbitrary piece of CEDALION code can be written without taking effect) where the `@author` annotation is used three times, on three different concepts.[3]

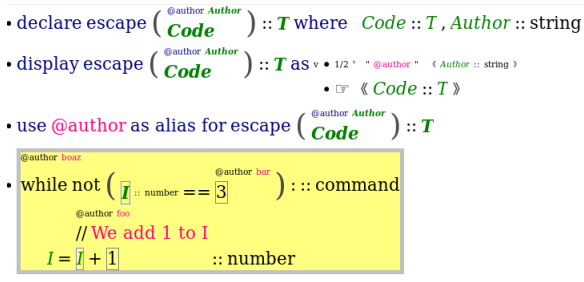


Figure 5: Definition and usage example of the `@author` annotation

Variants None

[3] The *escape* annotation seen in Fig. 5 is used to allow annotations to be defined. Like other annotations, it is removed from the AST before evaluation, but unlike other annotations, the removal process does not continue to child elements. This allows other annotations to appear in the AST evaluated by the interpreter, so that it can be declared and given a projection definition.

Figure 6: The MINIPERATIVE block concept projection definition, expanded and collapsed

Usability To add an `@author` annotation users need to select the code fragment to be annotated and type "`author`" (or "`@author`", which is a defined alias) and hit Ctrl+Space to select the correct concept. Then they need to select the author name placeholder and fill it with the name (as a string).

Impact This is a standalone concept, defined in its own source file independent of any DSL. Note, however, that the annotation namespace is a single namespace and this annotation can potentially collide with another annotation of the same name. Choosing the name ("`author`", in this case) should be done carefully.

Composability Annotations are composable. The author annotation, for example, can be combined with a comment, as seen on the last line in Fig. 5.

Limitations As described above, all annotations are allocated within the same namespace. This makes it possible for annotations from two different packages to collide.

Uses and Examples Annotations are used in CEDALION. They include:

- Comments on arbitrary code.
- Parentheses around arbitrary code fragments, to make structure visible.

Effort (best-effort) Definition took about 10 minutes, including concept declaration, projection definition, and embedding in sample code.

Other Comments None

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/notation-meta.ced`

2.3 Optional Hiding

"Aspects or parts of programs should be hidden optionally. Hidden aspects must be stored and users must be given the option to redisplay them later; the non-hidden parts must remain fully editable. A full solution should allow users to customize which parts of a program they can hide" [2, §6.5.1].

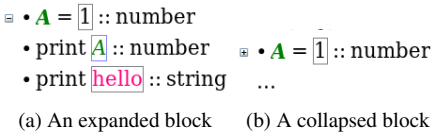
Assumptions This solution is based on the MINIPERATIVE object language.

Implementation We used optional hiding as a part of the concrete syntax of MINIPERATIVE. CEDALION has an *expand* visualization concept, which takes its collapsed and expanded visualizations as arguments. This allows DSL designers to specify two different ways to project a certain concept. The end user can, in turn, choose from them by expanding or collapsing it.

In MINIPERATIVE, we use *expand* for projecting blocks, where the collapsed projection shows only the first element of the block followed by an ellipsis ("..."). The expanded projection shows the entire list. Fig. 6 shows the projection definition of a block. The two lines in Fig. 6 are actually the same statement, having the right hand side expanded (in the first line) and collapsed (in the second line). In both modes the block is projected as a list of its elements, but two different projection modes are used. When expanded, *vert* mode is used, laying out all commands vertically preceded with bullets. When collapsed the *justShowFirst* mode is used, showing the first element followed by ellipsis.

Variants None

Usability Fig. 7 shows the projection of an example block, containing an assignment of the number 1 to variable A, followed by the printing of A and the printing of the string "`hello`". In its expanded form (Fig. 7a) all its statements are visible, while in its collapsed form (Fig. 7b) only the first element is visible. A ⊞ or ⊟ sign to the left of the block indicates whether the block can be expanded or collapsed, respectively.



(a) An expanded block (b) A collapsed block

Figure 7: Example block

In CEDALION making a certain part of the code collapsible is done in the projection definition of a concept. Users using a collapsible concept in a program must be aware that not all of the concept are shown by default (in CEDALION the expand visualization is currently displayed collapsed by default), and they need to expand it to see and edit the full content.

Impact Making the projection of an existing concept collapsible requires changing its definition. There are ways to define alternative projections for existing concepts (as will be discussed in Sect. 2.4 on multiple notations), and by that, existing concepts can be made collapsible, but that requires the user using the concept to explicitly choose the alternative (collapsible) projection.

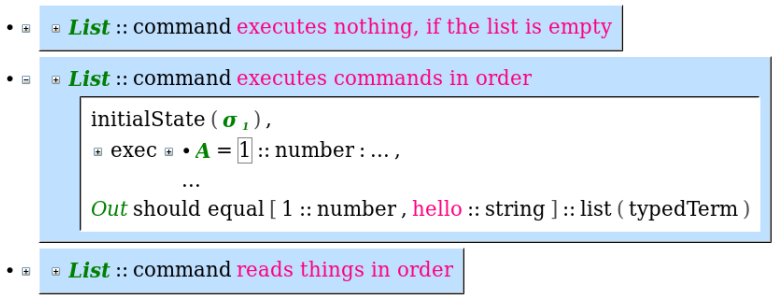


Figure 8: Behaviors specifying the semantics of the *block* concept

Composability Optional hiding can be used in conjunction with, e.g., mathematical notation, as discussed in Sect. 2.1. Since the *expand* visualization allows DSL designers to define two alternative projections from which a user can choose by collapsing and expanding, two alternative projections can be given to, e.g., the *sum* expression. The expanded form displays the complete expression, while the collapsed projection hides the bounds, and only shows the Σ sign along with the summed expression. This allows this expression to fit in a typical line.

The division line (the *div* concept) can also be made collapsable. In this case there are no good candidates for hiding, but we can make the collapsed projection more compact by replacing the division line with a / operator, laying out the numerator and denominator horizontally.

Limitations The *expand* visualization is currently presented collapsed by default. DSL designers should (and currently don't) have a way to specify whether they should be collapsed or expanded by default.

Uses and Examples The *behavior* concept used for BDD-style documentation and testing in CEDALION is shown collapsed by default, where only the concept being specified and the textual behavior description are presented. Expanding the behavior reveals the unit test that proves this behavior. This helps focus the reader on the text and the corresponding definitions (which are typically given alongside the behavior in the same source file), while the unit test code (which provides an example to the defined behavior) can still be viewed when desired.

Fig. 8 shows the three behaviors specifying the *block* concept in MINIPERATIVE. Two behaviors are collapsed and one is expanded. The unit-test shown in the expanded behavior includes two expandable concepts: the *block* concept from Fig. 7 and the *exec* predicate which only shows its first parameter (the command to be executed), and hides the other six arguments (variable assignment, input stream and output stream, each with its state before and after the execution of the command).

Effort (best-effort) One or two minutes. This only involves a modification of the projection definition of a concept.

Other Comments None

Artifact

- `https://github.com/brosenan/imperative/blob/master/block.ced`.

2.4 Multiple Notations

"This one addresses editing the same structure using different notations, based on a user's preference. A full solution will allow the free switching between multiple notations. As an example, mbeddr's state machines can be edited as text, diagram or table" [2, §6.5.1].

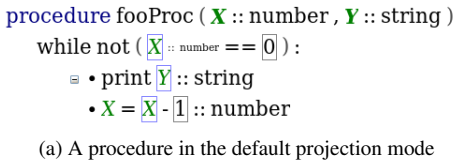
Assumptions This solution is based on the MINIPERATIVE object language. We modified the assignment to allow a user to replace the syntax of an already-implemented imperative procedure from its original Python-like syntax to a C-like syntax featuring curly-braces and prefix types.

Implementation CEDALION's projection logic, which converts a CEDALION AST to a tree of visualization objects, supports *modes*. Modes are an extra parameter given to the projection predicate which determines how to convert an AST node into a subtree of visualization objects. A single concept (AST node type) can be given different projection definitions for different modes, each associating it with different visualization subtrees. CEDALION has a default mode which is used implicitly in projection definitions. However, special syntax allows projection definitions of a particular mode.

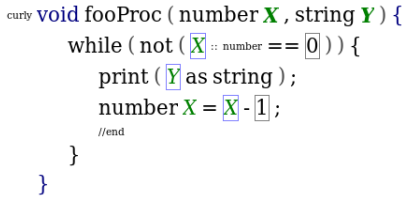
Providing an alternative projection for an existing DSL is done in two steps:

- Defining alternative projections using a new mode.
- Using the *changeMode* annotation, requesting that a phrase in the CEDALION program (e.g., a procedure in MINIPERATIVE) be projected using that new mode.

We called our new mode *curly*, and redefined (in *curly* mode) most of the concrete syntax of MINIPERATIVE, this time in a C-like manner. Procedures and while loops have curly braces around their blocks of code, while statements have parentheses around their conditions, assignment statements end with a semicolon, and formal arguments to procedures



(a) A procedure in the default projection mode



(b) A procedure in the *curly* projection mode

Figure 9: An example procedure in two different projection modes

are written in a C-like syntax (type followed by a name). To help users convert the projection of procedures to this C-like syntax we added a context menu entry that applies to procedure definitions, and wraps them with the *changeMode* annotation, with the *curly* mode as parameter.

Fig. 9 shows an example procedure named *fooProc* with two different projections. Fig. 9a shows it in the default projection mode, with Python-like syntax and types that appear after the variables they refer to, while Fig. 9b shows the same procedure in the *curly* projection mode, showing it in a C-like syntax. The word "`curly`" to the left of "`void`" is the projection of the *changeMode* annotation, which was inserted by the user to force the projection mode to change.

Variants None

Usability Switching the syntax from the default syntax to *curly* can be done using a context menu (right-click a procedure definition, select *Present in curly mode*), thanks to the context menu entry we defined along with the new syntax. This changes the display into the new syntax, while leaving a small "`curly`" annotation to the left of the definition. If users wish to return to the original syntax they need to remove the wrapper annotation. Removing a wrapper in CEDALION is done by copying the content (the definition itself) and pasting it in place of the wrapping annotation.

Impact This new syntax was added in a separate project, independent of the original DSL. If new concepts are added to the original DSL and we do not have projection definitions for them, CEDALION will fall back to the default projection (the projection defined for the original DSL) for these concepts.

Composability The *changeMode* annotation can be used anywhere in the code, including inside itself. This allows different concrete syntaxes to be mixed together. See *Uses and Examples* below for examples of where this can be useful.

Limitations CEDALION does not support tabular or graphical notations at this point, so changing notations can only be done among different textual representations.

Uses and Examples Sect. 4.2 discusses two cases where alternative projection is used to allow the end user affect the appearance of the code. In both cases the objective is to make the code fit the width of the screen.

Effort (best-effort) Defining an alternative syntax for a large subset of MINIPERATIVE took about 30 minutes. Doing so for the complete DSL would probably remain well below an hour of work.

Other Comments None

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/notation-curly.ced`

3. Evolution and Reuse

We address the following challenges related to modularity, composition, language versions, and migration [2]:

- *Language extension*: extend a language with new syntactic constructs in a modular fashion (Sect. 3.1).
- *Language embedding:* embed a separate language inside another (Sect. 3.2).
- *Extension composition*: combine independently developed extensions (Sect. 3.3).
- *Beyond grammar restrictions*: disallow constructs in certain scopes, without modeling this in the (abstract) syntax (Sect. 3.4).
- *Syntax migration*: support migrating programs when concrete syntax changes (Sect. 3.5).

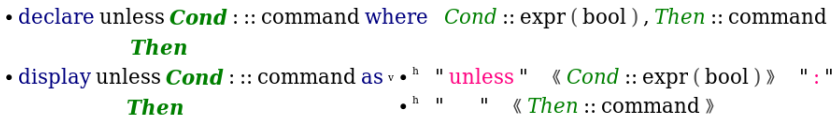
3.1 Language Extension

"One form of language composition involves adding additional constructs to existing syntactic categories. For instance, one could add support for an *unless* statement... How can such extensions be supported in a modular fashion...?" [2, §6.5.2].

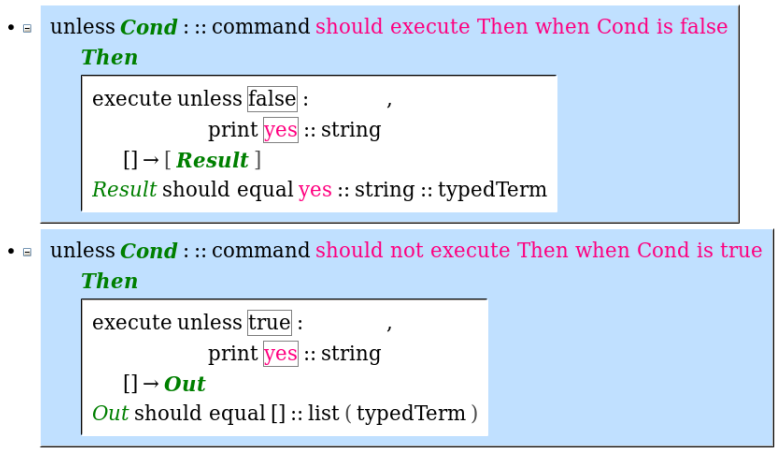
Assumptions This solution is based on MINIPERATIVE.

Implementation The *unless* command is defined as a new concept under the *lwc* namespace (not under the *imperative* namespace like the core commands of that DSL). Its definition includes a declaration (type signature) and a projection definition (concrete syntax), both shown in Fig. 10a, two behaviors (unit tests specifying its behavior when the condition is either true or false) shown in Fig. 10b, and a semantic definition shown in Fig. 10c.

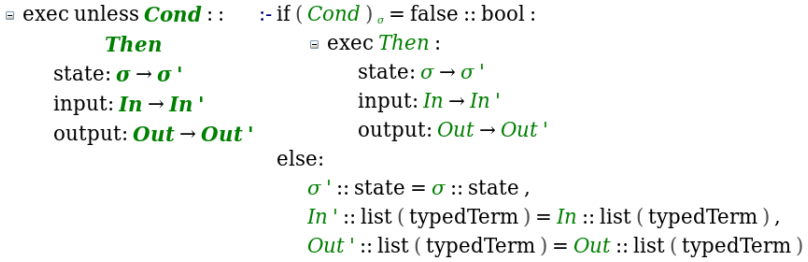
The semantic definition is a clause added to the *exec* predicate (in the MINIPERATIVE namespace). The *exec* predicate defines the semantics of a command by providing its effect



(a) Declaration and projection definition



(b) Behaviors



(c) Semantic definition

Figure 10: Definition of the *unless* command

to the state (the values of variables), the input stream, and the output stream. In CEDALION, any source file residing in any project can contribute clauses to predicates declared in any project. This allows third party projects (e.g., *lwc*) to extend DSLs defined in any other project (e.g., MINIPERATIVE).

Variants None

Usability Once defined, the *unless* command acts like any other MINIPERATIVE command. As such, to use it in a program a user needs to select it from the auto-completion menu.

Impact The definition of *unless* is external to MINIPERATIVE which it is extending.

Composability Since this solution involves only an addition to the language, it can be composed with other additions made, such as the sum expression or the division line.

Limitations None.

Uses and Examples Language extension plays a key part in DSL creation in CEDALION. CEDALION is a host language for internal DSLs, meaning that DSLs *extend* the core CEDALION language. Each DSL by itself can be extended to allow other DSLs to leverage its power. For example, CEDALION's *functional* DSL provides its users with the ability to define functions and expressions. This DSL is defined logically, through the *eval* predicate which determines the value of an expression. New DSLs can be defined using deep or shallow embedding, by extending the *functional* DSL with more expressions types.

Effort (best-effort) Implementing this solution took about 10 minutes.

Other Comments None.

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/evolution-unless.ced`

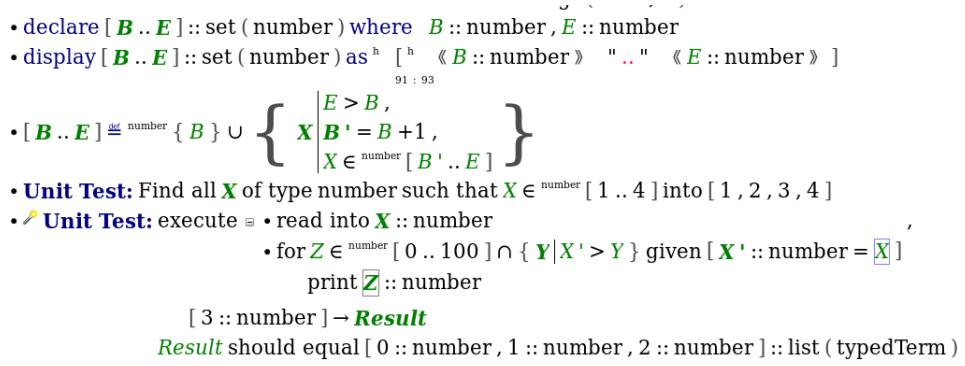


Figure 11: Composition between MINIPERATIVE and CEDALION's *sets* DSL

3.2 Language Embedding

"An important challenge here is to achieve the embedding with minimal "glue" code to tie the languages together, and without invasively changing either of the languages" [2, §6.5.2].

Assumptions The solution to this benchmark problem is based on MINIPERATIVE. We address a variant of the benchmark problem in which we embed arbitrary CEDALION DSLs in MINIPERATIVE.

Implementation CEDALION has a DSL for sets, defined as a part of its core library. This DSL defines set expressions, consisting of set comprehensions, singleton sets, a union or an intersection of sets, etc. It allows users to define sets (give a name to a set expression) and check if a value is a member of a set, as well as iterate over all members of a set using the $\in$ operator.

We wish to allow users to embed the sets DSL in code written in MINIPERATIVE. For example, we may wish to input some number from the user, and then output all the numbers in the range $0 \ldots 100$ that are smaller than that number. We can easily do this using pure MINIPERATIVE, by initializing an iterator at 0 and advancing it until it reaches either the number we read or 100, but using the sets DSL may result in a more concise and descriptive code.

To allow the sets DSL to be embedded in MINIPERATIVE programs we define the *for* command. The *for* command allows MINIPERATIVE programs to iterate over the members of a set defined using the sets DSL. It has three parts:

1. An iteration over a set: a condition of the form $X \in S$, where S is a set expression and X is a logic variable;
2. A list of bindings: MINIPERATIVE expressions to be made available to the set expression;
3. A command to be executed for each value of X (body).

Its semantics is defined as an evaluation of the bindings, followed by an evaluation of a set comprehension, providing an instance of the body for each result, followed by an evaluation of the resulting MINIPERATIVE block.

Fig. 11 shows a usage example of the *for* command to demonstrate language composition between MINIPERATIVE and the sets DSL. The first four statements are a definition of a range set, which contains all integers between B and E, inclusive. The fifth statement is a unit-test executing a small MINIPERATIVE program which reads a number from the input stream and then prints all the numbers between 0 and 100 (inclusive) that are also smaller than the number that has been read.

Variants None.

Usability Once defined, the *for* command becomes just another command, like *if* or *while*.

The only part that is potentially harder to understand or use is the bindings list. For every value that needs to pass from the MINIPERATIVE program to the other DSL, the user needs to add a binding, which consists of an MINIPERATIVE expression (e.g., a variable reference), its type, and a free variable which will receive the expression's value.

These bindings are necessary because MINIPERATIVE expressions are state-aware, while other DSLs are not. The state used by MINIPERATIVE is defined as a part of that DSL. As result, MINIPERATIVE variables and expressions cannot be used in other DSLs. It is therefore necessary to extract their values before exiting the MINIPERATIVE domain.

Once we have these values we can evaluate a goal, which can refer to any DSL. The resulting values can then be used in MINIPERATIVE expressions as constants, so no binding is required in that direction.

Impact The *for* command stands on its own, and is defined as a part of this solution, independent of MINIPERATIVE it is extending.

Composability Language embedding is composable. Since any DSL can be evaluated using logic goals. Thus, any CEDALION DSL that provides a way to evaluate goals opens the door to language embedding.

Limitations Language embedding in CEDALION is limited to CEDALION DSLs. Languages that are not represented as

CEDALION DSLs (e.g., Java or C) cannot be embedded this way.

Uses and Examples Language embedding is a core principle at the root of Language Oriented Programming (LOP) [9, 1]. DSLs provide expressiveness in their own problem domain. However, software is never about a single domain. Language embedding provides an effective way to write software using multiple DSL, integrating seamlessly to one another.

Effort (best-effort) Implementation of the *for* command took about 45 minutes.

Other Comments CEDALION DSLs (MINIPERATIVE itself included) are defined in terms of predicates. For example, CEDALION's *functional* DSL is defined in terms of the *eval* predicate, which determines how expressions are evaluated. MINIPERATIVE is defined in terms of the *exec* predicate, which determines how commands interact with the world (state, input and output). Similarly, the sets DSL is defined in terms for the $\in$ predicate. The *for* command described here is not limited to the use of $\in$ as its iteration condition, and allows any predicate to be used from within MINIPERATIVE code. This makes it a universal "embedder" of DSLs into MINIPERATIVE: code in any CEDALION DSL can be embedded in MINIPERATIVE using the *for* command.

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/for.ced`

3.3 Extension Composition

What has to be done to use multiple independent extensions in the same object language.

Assumptions This solution is based on MINIPERATIVE and three extensions created as solutions for two other challenges: The *until* command defined in Sect. 3.1, the sum expression, and the division line expression defined in Sec. 2.1.

Implementation Fig. 12 shows a procedure, which prints "not one" unless a certain numerical expression evaluates to 2. This numerical expression will evaluate to two if the

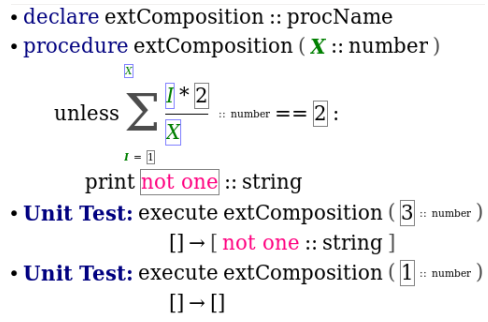


Figure 12: Extension composition example

argument is 1. The expression itself uses a sum and a division, as defined for the Mathematical Notation benchmark problem (Sect 2.1). Two unit-tests demonstrate that the procedure works as expected.

Variants None.

Usability In CEDALION, there is no difference between using a core DSL and using a DSL extension. Using a concept defined as an extension is exactly the same as using a core DSL concept, except for choosing the concept from a different namespace.

Impact This solution is independent of both MINIPERATIVE and the extensions used.

Composability N/A

Limitations None.

Uses and Examples Extension composition is an essential part in how CEDALION DSLs grow. Same as software typically grows as a composition of software libraries or modules, CEDALION programs grow using packages (Eclipse projects) which may define new DSLs, but may also extend existing ones. As users are free to extend DSLs, and as these DSLs seamlessly compose, packages can be used together without restriction.

Effort (best-effort) It took about five minutes to implement the solution.

Other Comments None.

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/extension-composition.ced`

3.4 Beyond Grammar Restrictions

"Language grammars partially restrict the programs that can be expressed. However, in some cases, additional constraints may be necessary to statically reject semantically undesirable programs... How can we restrict the context in which a language concept can be used?" [2, §6.5.2].

Assumptions Our solution is based on MINIPERATIVE with the *unless* extension. We solved a variant of the assignment: adding a restriction so that a negative condition on an *if* statement will not be legal. Instead, *unless* should be used.

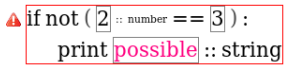
Implementation CEDALION decorates the program AST with markers provided by a predicate named *check* that is given the various AST nodes, as well as the location of the current node in the tree (the path from the root). Clauses of this predicate may contribute results, in the form of markers. Markers can come in different forms and represent different things, but the most important of them is the error marker.

Errors can be contributed by clauses written by anyone. CEDALION's *bootstrap* package comes with clauses implementing its type system, which is used, among other things,

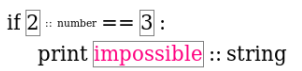
to enforce the program's conformance to the abstract syntax. However, by creating additional clauses for specific scenarios we can go beyond the abstract syntax and enforce other constraints.

In our case, there is no grammatical way to allow, on the one hand, *not()* expressions in general Boolean expressions, but, on the other hand, not to allow them at the top level of the condition in an if command. However, we can define a specific checker (a clause for the check predicate) that enforces that rule specifically.

Fig. 13 shows two *if* commands, one with a negative condition (Fig. 13a), and the other with a positive condition (Fig. 13b). The former is marked with as an error, while the latter is not.



(a) Error reported due to negative condition in *if*



(b) *if* with positive condition does not produce an error

Figure 13: *if* commands with and without error

Variants None

Usability To a user, the added checker appears as an error marking on their code. No action is required of them to see this error.

The user can double-click the icon to the left on an error to get a list of proposed fixes. In our case, the user will get a proposal to replace the *if* command with *unless*.

Impact This solution depends on MINIPERATIVE and indirectly, the *unless* extension. However, it is external to both.

Composability Checkers compose. We can, for example, add another checker that makes it illegal to use constant conditions (ones that do not depend on variables) in *if* commands. In such a case, the condition *not(2==3)* defies both restrictions and will be marked with both errors.

Limitations CEDALION checker implementations have full access to the language and therefore any computable restriction can be enforced. However, some restrictions are easier to enforce than others.

The restriction in the original assignment is currently not easy to implement in CEDALION. Checkers are not context aware, so that checking that a certain construct is used within a certain context requires building the right facility. This could be a generic facility that allows users to (1) define contexts based on AST nodes, and (2) provide markers (including errors) for AST nodes based on these contexts. The whole thing needs to be integrated as a checker. With such facility in place, solving the original assignment is as simple as defining a context for state entry code, and then checking for each state transition command that it is performed from within the correct context.

Uses and Examples Unit tests are one important example for the use of restrictions in CEDALION. Unit tests are logic goals that are expected to succeed. This expectation is enforced by a checker, that contributes an error if the goal fails, throws an exception or times out. By making this a semantic restriction we convey the notion that it is not enough for a program to be structured correctly to be valid, it also needs to behave as expected.

Effort (best-effort) Implementing the checker along with its supporting unit tests took about 35 minutes. An additional one or two minutes went into specifying the fix for this error.

Other Comments None

Artifact

- `https://github.com/brosenan/lwc2016/blob/master/lwc/unless-restriction.ced`

3.5 Syntax Migration

"How can programs be migrated from the old to the new syntax?" [2, §6.5.2].

Assumptions Our solution is based on the MINIPERATIVE DLS, and especially the *print* and *read into* commands.

In this solution we modify the syntax of the *print* and *read into* commands replacing these keywords with *output* and *input* respectively.

Implementation The implementation was a modification of the projection definition of both concepts. No further changes were necessary.

Variants None

Usability To an end user the only effect is that what was seen as a *print* command is now seen as an *output* command. The user can type either "`print`" or "`output`" to add such a command to the code.

Impact This solution required a change in the definition of the original DSL. However, no changes are necessary in user code.

CEDALION persists code as ASTs (actually, as logic terms, which in turn are trees). Compound terms (representing non-leafs in the AST) have a name, which is an identifier qualified with a namespace. This name references a declaration (type signature) and optionally other definition, including a projection definition. A projection definition tells the projectional editor how to display the term.

In this solution we modified the projection definition, but did not change the underlying name. The *print* command is no longer displayed as "`print`" (it is now displayed as "`output`"), but is still named "`print`" in the AST. For this

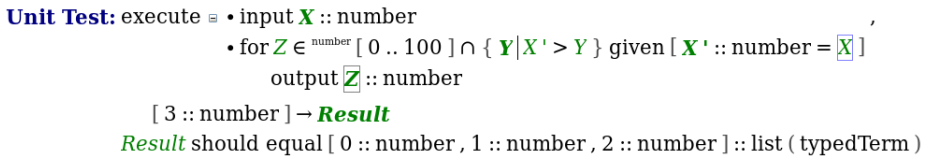


Figure 14: The language embedding example with the modified projection

reason no other piece of code had to change to make this change happen.

Fig. 14 shows the last statement of Fig. 11 displayed after changing the projection definition of *print* and *read*. Note how "`print`" became "`output`", and "`read into`" became "`input`". No changes were made to the source file presented in this figure.

Composability N/A

Limitations Modifying the concrete syntax when code is already written has the side-effect that DSL users may not recognize their own code. As result, this should be done sparingly, with proper documentation and notification.

Uses and Examples The fact that concrete syntax can be determined at any point in time in the evolution of a program makes the choice of concrete syntax less critical. This is in contrast to parsed languages, where it is highly important to get the concrete syntax right from the start. Changing concrete syntax often invalidates existing programs. For example, the introduction of a new keyword has the side-effect of invalidating any program using this keyword as an identifier. Projectional editing does not suffer from this problem, as the concrete syntax is not used to persist the code.

Effort (best-effort) Making the change for both commands took less than 5 minutes.

Other Comments None

Artifact

- `https://github.com/brosenan/imperative/blob/master/print.ced`
- `https://github.com/brosenan/imperative/blob/master/read.ced`

4. Editing

We address the following challenges [2] exercising how the language user interacts with code:

- *Editing incomplete programs:* support for syntactically malformed programs (Sect. 4.1).
- *End-user defined formatting*: show if and how user can change the visual appearance of the program (Sect. 4.2).

Challenges .

4.1 Editing Incomplete Programs

Assumptions None

Implementation N/A

Variants None

Usability CEDALION programs consist of *statements*, which are logic terms. A logic term may be a compound term, consisting of a name and zero or more subterms, a number, a string or a logic variable.

CEDALION programs always consist of complete terms, but these terms may be very simple, or partial with respect to how the user would like them to be eventually.

An empty source file consists of an empty list of statements (Fig. 15a). To add content to the file the user needs to insert an element to the list. This element is, by default, a free variable (an unnamed variable denoted as "_"), as seen in Fig. 15b. The user can then choose the top level concept for this statement by selecting the free variable and selecting a concept to replace it. Fig 15c shows the result of choosing a *procedure* definition in MINIPERATIVE. Once the concept is chosen, it appears with its parameters as unnamed variables. The user can then select each of these variables and replace it with content, and so on and so forth.

Impact N/A

Composability N/A

Limitations The CEDALION WORKBENCH is currently sensitive to partial statements being saved to disk. The CEDALION WORKBENCH consults the CEDALION program being edited as a part of its projection mechanism. At any given point in time the version being consulted is the last saved version.

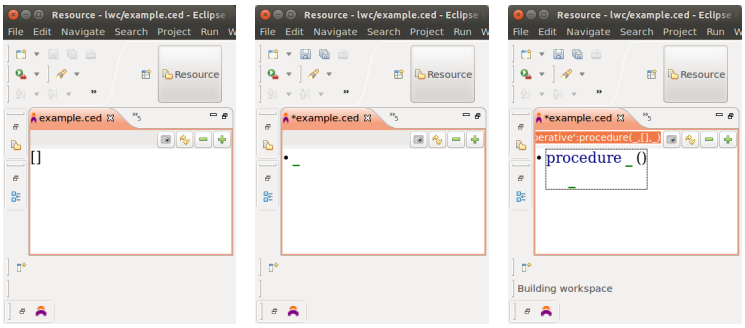
If this saved version consists of partial statements (e.g., statements that still have unbound variables in places where actual code should exist) these statements will take effect as a part of the program being consulted. If these statements are consulted in the projection logic, this could, in some cases, cause the CEDALION WORKBENCH to misbehave or even crash. We are currently working on ways of mitigating this, e.g., by better segregating calls to the program by the workbench.

Uses and Examples N/A

Effort (best-effort) N/A

Other Comments None

Artifact N/A



(a) An empty source file (b) An empty statement (c) An empty procedure

Figure 15: Incomplete Program Examples

4.2 End-User-Defined Formatting

Assumptions N/A

Implementation N/A

Variants None

Usability CEDALION supports a limited form of end-user defined formatting. While the concrete syntax is defined by the projection definitions of the various DSL concepts, it is possible for a concept to have more than one projection. As discussed in our solution for Multiple Notations (Sect. 2.4), CEDALION's projection logic supports the notion of projection modes. A concept can have different projections for different modes. Modes are typically used to refine the concrete syntax, giving concepts a different look based on the context in which they are used. However, modes can also be used to allow the end user choose a projection that better matches a certain piece of code.

Two prominent examples are the *shrink* mode, and list modes.

Shrink is a mode in which concepts are expected to take less width, at the expense of extending longer vertically. It should be used, for example, where lines are too long to fit on a typical screen. While developers are free to define their own "shrunk" projections, a projection under the shrink mode is automatically derived from the default projection for concepts for which the default projection is a horizontal layout with three elements or more. In such cases (as can be seen in Fig. 16a), the layout is broken into two lines, with the first two elements shown on the first line (in this case, the word "declare" and the concept along with its type), and the rest of the elements (in this case, the argument types) are shown, indented, on the second line. The user can choose to use the shrink mode by wrapping the element they wish to shrink with the *shrink annotation* (shown as a small downwards arrow at the top-right corner of Fig. 16a). This annotation is a change in the AST, but this change does not affect the meaning of the program. Annotations are removed from the AST before it is interpreted.

A list can be displayed either horizontally or vertically. In many cases, the default projection for a concept that has a list as a child chooses horizontal layout for that list. This is based on the assumption that the list is short enough to fit in the width of a typical screen. However, if the list becomes too long, a vertical layout is preferred. The end user can choose this layout by annotating this list with a *changeMode* annotation (discussed in Sect. 2.4) with the *vert* mode. To use this annotation, a user needs to right-click the list and choose "display vertically" from the context menu. Fig. 16b shows a procedure definition where the long list of arguments is presented vertically. Since the parentheses are part of the original projection for procedure arguments, changing the projection mode in this case replaced them with bullets. The parentheses shown around the arguments in Fig. 16b are yet another annotation (the *parentheses* annotation), which draws scalable parentheses around any AST node. Here too, neither annotations affect the semantics of the code, as they are removed before interpretation.

Impact N/A

Composability None

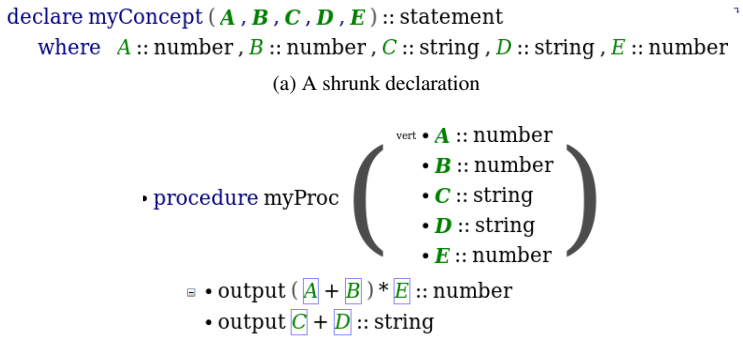
Limitations This form of control over layout is very limited. The end user cannot move code fragments arbitrarily, and is instead confined to the modes defined in the DSL. The end user can define new, alternative projections for new modes (as we did in our solution for the Multiple Notations challenge in Sect. 2.4), but we can think of very few scenarios where this will be beneficial.

Uses and Examples We use control over layout to allow statements that would otherwise exceed the width of a typical screen, to fit within its boundaries.

Effort (best-effort) N/A

Other Comments None

Artifact N/A



(a) A shrunk declaration

(b) A procedure definition with a vertical argument list

Figure 16: End-user define formatting examples

5. Conclusion

In this paper we address 11 of the 25 benchmark problems published in the 2016 Language Workbench Challenge [2]. The benchmark problems not addressed here are either inapplicable to CEDALION, or are supported by CEDALION so trivially that they would just not be considered interesting benchmarks. One example is *Skeleton Editing*, which is derived directly from the use of projectional editing.

It is worth noting that for the sake of scientific integrity we did not modify or extend CEDALION to make it better suite the challenge. Our implementation includes the MINIPERATIVE DSL and the individual benchmark problems. However, that said, we consider the limitations discussed for the various benchmark problems in this paper as improvement opportunities for further development of CEDALION.

References

[1] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[2] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches. *Computer Languages, Systems & Structures*, 44:24–47, dec 2015.

[3] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[4] Martin Fowler. Language workbench. Martin Fowler's Bliki. `http://martinfowler.com/bliki/LanguageWorkbench.html`.

[5] Martin Fowler. Projectional editing. Martin Fowler's Bliki. `http://martinfowler.com/bliki/ProjectionalEditing.htmlx`.

[6] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.

[7] David H. Lorenz and Boaz Rosenan. Cedalion: A language for language oriented programming. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'11)*, pages 733–752, Portland, Oregon, USA, October 2011. ACM.

[8] Eric Roberts. An overview of minijava. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 1–5, New York, NY, USA, 2001. ACM.

[9] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.