A Language Based on Two Relations between Symbols

Agustín Rafael Martínez
Universidad de Buenos Aires
Argentina
agustin@dc.uba.ar

Abstract

We present a language with all the power of abstraction and the simplicity of two fundamental relations: substitution and categorization. With a graphic symbol representing each one of them, we created a playful visual programming environment aimed at teaching with high expressive power. This environment includes tools to inspect the program execution and a console to try visual expressions. This is achieved without resorting to text, since the symbols are user-defined drawings. To address complex problems, the language offers another set of tools to define text-based programs. Here we show a functional prototype of our rule-based, general-purpose declarative programming language.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics → Computing education**.

Keywords: general-purpose programming languages, visual programming, computer science education

ACM Reference Format:
Agustín Rafael Martínez. 2022. A Language Based on Two Relations between Symbols. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '22), December 8–10, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3563835.3567660

1 Introduction

In teaching programming, an important issue is the limitation of the expressive power of visual languages [32]. Another well-known problem is the gap in the teaching process from simple graphically defined programs to complex programs written in conventional languages [7] [6] [25]. Both problems are related, since it is the expressive limitation of visual languages that forces a change when advancing towards the programming of complex models.

The approaches to these problems are oriented in two identifiable directions. There are projects that decide to maintain simplicity at the cost of having a limited power of abstraction [27] [14] [7]. On the other hand, there are projects that decide to increase the expressive power of visual languages [10] [32], and what is gained in expressiveness is lost in complexity. Faced with this situation, teachers must choose the tool to introduce their students to programming and then identify the instance or instances in which the language change is necessary.

In this work we present a different solution. The proposal is not to change the language, but only to change the tools. Our language can work in a visual way with a high power of abstraction. We can define categories and parameterize any sequence of symbols, no matter what kind they are. When we need to move towards models where text is more appropriate to express them, we can change the tools but continue programming with the same language.

The name of the language is *Representar*, which in Spanish means 'to represent'. The name highlights the vision of the project: programming is creating "epistemic representations" [4, p. 1]. This point of view is constitutive of the language, which is based on a representationalist theory of knowledge [21] [22]. From this theory comes the idea that human symbolic language is based on two fundamental relationships between symbols: substitution and categorization.

The structure of the paper is as follows. First, we show different scenarios addressed by graphical programming with our language. Then we present the syntax of the language and we analyze its fundamental characteristics. Subsequently, we describe some aspects of the implementation that help to understand the evaluation mechanisms. After that, we present text-based implementations with the language, some of them considered complex in programming. Later, we review related works in both dimensions of the project: visual and textual. Finally, we discuss future works and make some final comments by way of conclusion.

2 User-Defined Drawings as the Symbols Building the Program

We present a declarative alternative to graphical programming that dispenses with text completely, and encourages programming through drawings made by the user. These drawings become the symbols that the learner can drag and

Onward! '22, December 8–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9909-8/22/12…$15.00
https://doi.org/10.1145/3563835.3567660

drop at will, living an early experience about the essence of computing: the manipulation of symbols.

We show four versions of our visual programming environment, adapted to different activities. The intention is to highlight the variety of topics that can be addressed by programming with drawings: from socio-environmental issues, robot behaviors, to simple representations of natural numbers and arithmetic operations.

2.1 The Drawing-Based Programming Environment

Here we present the basic programming and drawing tool and the fundamental symbols that allow programming with user-defined drawings.

Table 1. Basic cards of the drawing environment

	Points	Defines a substitution
	Arround	Defines a categorization
	When	Defines a condition on a substitution
	Wildcard	Categorizes any sequence of symbols

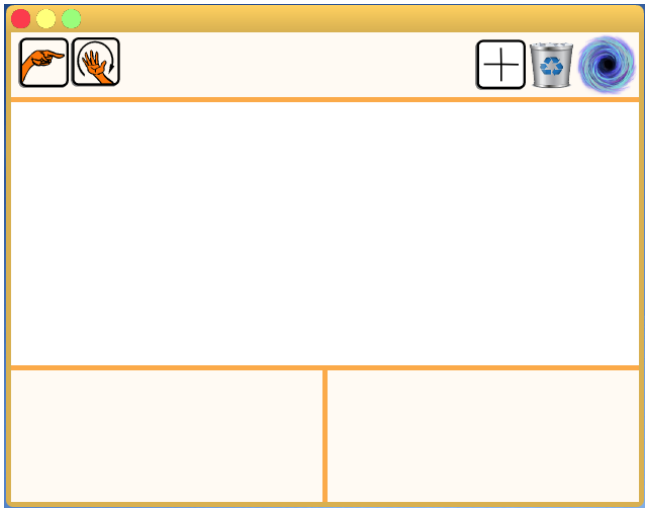


Figure 1. The minimal version of the visual programming environment with the two cards that represent the two fundamental relations.

On Table 1 we present a description of the four basic cards of the visual environment and in Figure 1 we show what users have initially available. This is a playful environment that already has two fundamental cards of this programming game. These cards are available at the top left. The first card represents the act of pointing or referencing: any sequence of symbols can point to any other sequence. The second card represents the other fundamental relationship of the language: categorization. Every symbol can be categorized by any other. The drawings chosen to represent these relationships are inspired by two symbols of the American Sign Language that mean 'pointing' and 'around' respectively. At the top right of the environment there are three elements: a button to draw new cards, a trash can to get rid of created drawings, and a black hole to get rid of copies of the cards that have been unused. In the center of the window is the board where the cards are played to define the program. Below is a kind of console or REPL, that lets us throw cards on the left and see the results of their evaluation on the right. When the cards in the central board change, the results of the REPL are updated live.

The first step to program in this environment is to draw new symbols by pressing the button to create a new card. This is how the drawing tool shown in Figure 2 opens. We can take colors from the palette that is around the card frame and draw inside the established space. In this way, the different symbols that will be manipulated in the representation are created.

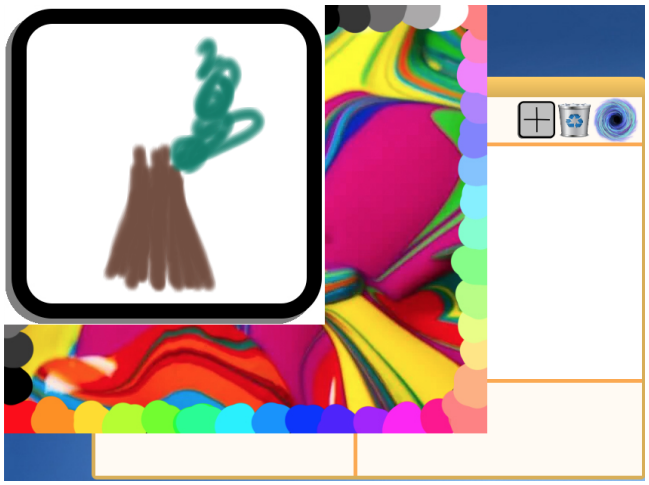


Figure 2. Drawing a card: the tree.

2.2 Representing a Socio-Environmental Problem

We want to show that with our language and tools we can deal with any topic without the need to go beyond the manipulation of symbols.

Let's suppose the case in which the region where the children live is being crossed by a socio-environmental problem. Teachers can decide to address the issue in class. In such a case, is there a possibility that programming can participate in an experience of reflection on a socio-environmental problem? Let's assume the problem is forest fires. Why not draw and program around this problem? It is not proposed to represent the problem with scientific rigor but rather to attend to an expression need through programming and drawing.

In the face of forest fires, the figure of the tree, the fire and the ash can be represented. At the top of Figure 3, on the second line, all the cards drawn are located from where unlimited copies of them can be obtained. A first rule to define with these cards is the one where the fire ignites the tree leaving ash in its advance. This is represented in the first line of the central board: tree and fire are replaced by fire and ashes. The other aspect addressed is the fight against the fire by means of a helicopter firefighter who turns the fire into ashes, leaving a rainbow in its wake and keeping the tree intact. This is the second substitution rule of the implemented program.

Figure 3. A program representing a socio-environmental problem.

Since fire can originate from different sources, not just from a bonfire, it is possible to categorize other cards as other possible forms of fire. This is expressed in the third line, the cigarette behaves like any other fire in this representation. In the same way, flowers and bushes can be equally victims of the flames, in that sense the tree is a "prototype-based category" [17, p. 62] of the rest of the plants. This is expressed in the last two lines of the central board.

At the bottom of Figure 3 we can see how the defined program works. Three lines are evaluated separately, the first and the second are identical except for the presence of the firefighter. In the first case the result is that the plants are preserved, while in the absence of the firefighter everything turns to ashes and fire.

It is usually considered something advanced to have access to debug or inspect in some way the execution. In this project, by simply clicking on a result, a window opens that shows the transformation of the initial sequence until the result is reached.

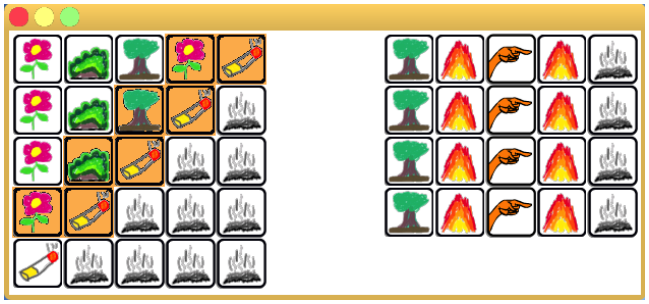


Figure 4. Inspection of substitutions in an evaluation.

If we click on the place where the mouse pointer is in Figure 3, the window of Figure 4 opens. In this way, the execution is inspected. On the left side of Figure 4, the transformation of the evaluated sequence is seen, with the symbols that are being replaced in each step highlighted in orange. To the right of each line is the substitution relation applied in each transformation, which in the example is always the same.

In short, the evaluation works as follows: given a sequence of symbols, the interpreter looks for relations to replace the entire sequence. If it is not possible to replace the entire sequence, it is attempted to transform it by parts. Once a part is transformed, the entire sequence is attempted to be transformed again. The execution ends when there are no substitutions that can transform the sequence. In this sense, every sequence of symbols is syntactically valid and therefore executable. The language is fully permissive, that is one aspect of the flexibility of its syntax.

2.3 Programming the Behavior of a Robot

With this example we want to show that with our language it is possible to represent typical behaviors in educational robotics.

This second scenario provides specific cards for robot control that we describe in Table 2, some of them referring to basic robot behaviors and other to its parts: sensors and actuators. The robot in question is driven by two independent wheels, and has two proximity sensors and two light sensors at the front. This is a fairly standard configuration of a teaching-oriented robot, which allows defining typical behaviors in educational robotics: avoiding obstacles and moving in relation to the ambient light [20] [2].

An educational sequence could probably begin by exploring the simulator by throwing predefined symbols into the REPL console and seeing how the robot moves. Later we can try to define an infinite loop with a simple behavior, like going back and forth continuously. Then introduce the behaviors of going towards the light and avoiding obstacles

Figure 5. The drawing-based programming environment including a robot simulator.

separately and finally the combination of both in a single program as it appears in Figure 5 and is explained in Table 3.

In the center of Figure 5 we can see: on the left, the board with a program already defined and on the right the simulation. We can add obstacles or light sources and see how the robot performs in different situations. To define the program we draw three new cards: one representing an infinite loop, another representing the robot's behavior in general, and another that represents a particular behavior that the robot can adopt, go towards the light. The meaning of the cards is expressed in its drawings but the cards make sense, for the program, in the substitution rules defined on the board. In the example, we defined five substitution explained in Table 3. There we can see that the behavior of the robot has 3 definitions, the last line defines a behavior without conditions, where the robot goes towards the light. This particular behavior is defined in the second line of the board following the classical proposal of Braitenberg [2, p. 7 vehicle 2b]. The order in which the lines are defined is not relevant for the interpreter. The third and fourth substitutions in the board of Figure 5 are applied when the left or right proximity sensor perceives the proximity of something. In those cases, the robot turns in the opposite direction to the sensed obstacle. In this way, the behavior of going towards the light and the one of avoiding obstacles are combined in a simple graphical program defined without any text.

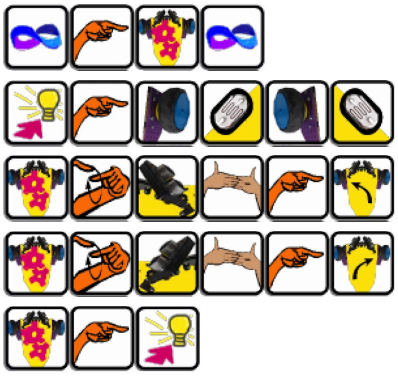
The behavior of avoiding obstacles has priority over going towards the light, given that the substitutions with conditions have priority of application over the ones without conditions. The decision of which substitution to apply is up to the interpreter, simplifying the programming. Here we see the contribution of the declarative nature of the language.

Table 2. Provided cards to define robot behaviors

	Robot goes forward
	Robot goes backwards
	Robot turns right
	Robot turns left
	Robot stops
	Right wheel
	Left wheel
	Right light sensor
	Left light sensor
	Right proximity sensor
	Left proximity sensor
	Near an obstacle
	Execution waits a while

To get the robot running, we throw the loop card into the REPL console. The ellipsis points at the bottom left of Figure 5 mean that the evaluation of the expression on the right is not yet complete, and it won't be since it's an infinite loop. This does not prevent us from inspecting the execution.

Table 3. Description of each line of the program of Figure 5

	Infinite loop user-defined drawing substituted by: robot behavior drawing and again infinite loop drawing Go to the light substituted by: left wheel takes right light sensor value and right wheel takes left light sensor value Robot behavior, when right proximity sensor is near an obstacle, substituted by: robot turns left Robot behavior, when left proximity sensor is near an obstacle, substituted by: robot turns right Robot behavior user-defined drawing substituted by: go to the light user-defined drawing

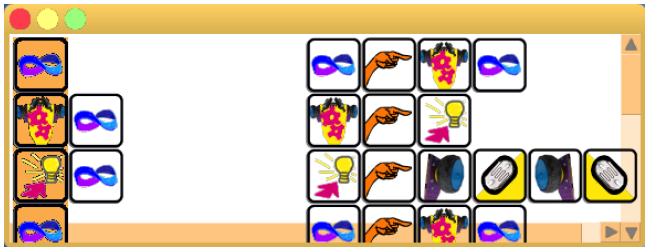


Figure 6. One iteration of the infinite loop that updates the robot's behavior.

Figure 6 shows the execution of one iteration of the loop when the proximity sensors do not find obstacles so that the behavior is to go towards the light.

New relationships can be defined through predefined cards or new cards drawn. Changing relationships affects live the behavior of the simulated robot, without the need to interrupt its movement.

2.4 Building a Control Flow Notation

With this example we want to show that with our language it is possible to fully represent a simple abstract model such as Booleans and with it emulate syntactic structures of conventional languages.

In Figure 7 we show a representation of booleans and a definition of an 'if… then… else' notation based on them. We draw symbols to represent truth and falsehood: 'T' and 'F' respectively. Then we draw symbols for the logical operations 'and', 'or', 'not', and a symbol for the abstract notion of boolean. We categorize 'T' and 'F' by the notion of boolean in the first two lines of the board. The next six lines complete the representation of booleans.

We defined the 'if… then… else' statement in the last two lines of the board. To do it, we had to play the wildcard. Like in card games, the wildcard can take the place of any card. In our game, the wildcard is a primitive symbol that

Figure 7. Definition and use of booleans and 'if… then… else' notation.

categorizes any sequence of symbols. We used it here in a particular way: we placed it 'around' other symbols that we draw to represent any phrase. These newly drawn symbols now categorize the wildcard, so they categorize the symbols included by the wildcard, thay way becoming user-defined categories that include any sequence of symbols.

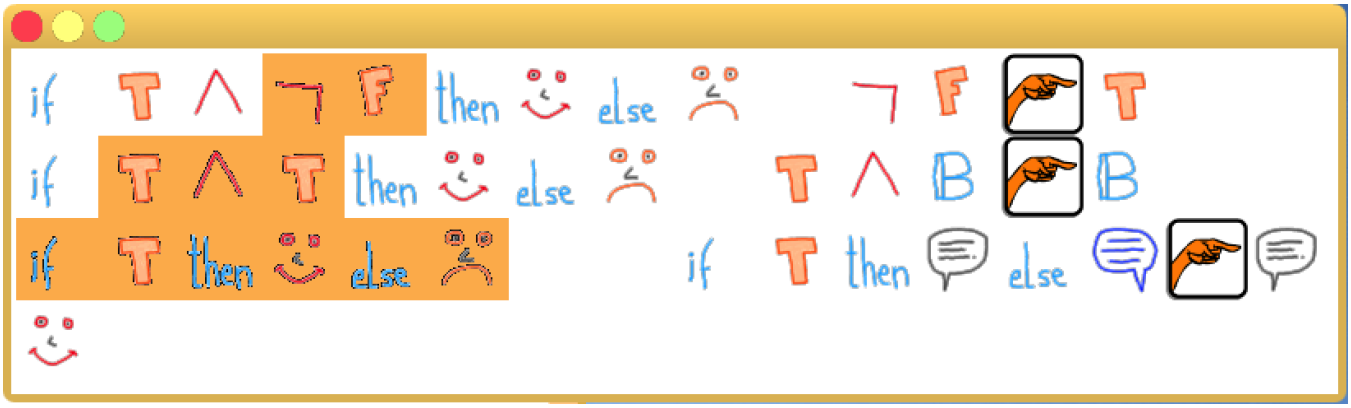


Figure 8. Inspection of an execution where the control flow notation is applied for the case where the condition is true.

In the case that the condition is true, the whole statement is replaced by the phrase after the 'then'. When the condition is false, the resulting phrase is the one after the 'else'. Since evaluation ends when there are no more substitutions to apply, the resulting phrase of the 'if… then… else' will continue to be evaluated while the other phrase will be ignored and not executed.

Figure 8 shows the substitution process until the truth value of the condition is obtained and then the corresponding 'if… then… else' substitution is applied.

2.5 Modeling the Natural Numbers

With this example we want to show that it is possible to visually represent some elements of complex abstract models such as the arithmetic of natural numbers.

In Figure 9 we show some drawn digits. By means of next and previous relationships between them we define addition. The number two works as an exemplar of the numbers greater than one, it is a prototype-based category. On the other hand, the symbol 'ℕ' represents the abstract notion of number, in this sense it is a class-based category. From the point of view of the language there is no difference between both categories, this distinction is an interpretation of the programmer. These categories make it possible to define the addition recursively, with the base case being when 1 is added to another number. We can see these definitions in the first and second line of the center of Figure 9. In Figure 10 we can se the execution of the expression '2+2'.

The last expressions at the bottom of Figure 9 does not achieve the expected result because there are undefined relations. Nevertheless, evaluation does not fail, on the contrary, it expresses where to continue the development of the model. The sum of '3+2' returns 'the next of 4', making it explicit where the program should be expanded to include that calculation. We should draw the number five and define it as the successor of four to complete it.

With drawings and the two fundamental language relationships, we represented something as abstract as natural number and an arithmetic operations. However, a complete representation of the natural numbers requires making the jump to text-based programming.

Figure 9. A simplified representation of the sum of natural numbers.

Table 4. Drawing and textual notation of the *Representar* language

... ...	... → ...	Defines a substitution rule
... ...	... ↷ ...	Defines a categorization
... ...	... | ... → ...	Defines a substitution with conditions
	... ↷ ... → ...	Defines a substitution, declaring local parameter names at the beginning
	... ↷ ... | ... → ...[1]	Defines a substitution with conditions, declaring local parameter names

[1] See Figure 19 for an example of use of this notation.

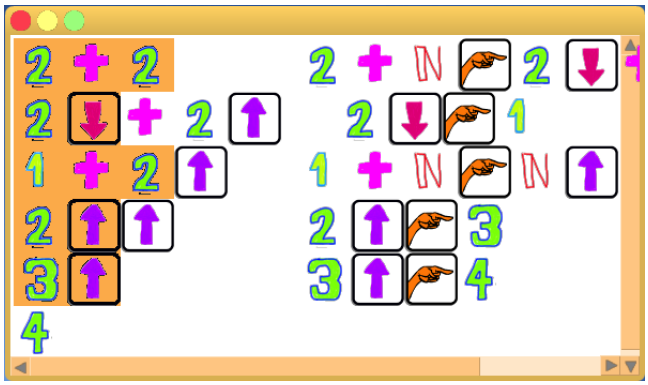


Figure 10. Substitutions that solve the sum of 2+2.

3 The Language Characteristics

The previous activities have offered an approximation to the language in its simplified graphical version. Now we will analyze the fundamental characteristics of our programming language.

3.1 Syntax

The language *Representar* is intended to be as flexible in its syntax as natural language and writing are. Keeping this analogy in mind, in its textual version, the language's elements are words organized into sentences. In general, the elements from which a program is built are symbols organized into sequences. In the current implementation, the symbols are graphical or textual. In future implementations, they could be of another nature: auditory, tactile, or another sensory alternative. In this language, every sentence, or sequence of symbols, 'compiles' and runs. There are no syntax errors that prevent the evaluation of a sentence. What can happen is that there is no substitution of symbols that is applied to a given sequence. In this case the execution will result in the same expression originally evaluated.

To define substitution rules or categories between symbols, we have to do it by evaluating sentences with particular formats. In Table 4 we present the notation to define these relationships. The substitution and categorization relations previously defined with these notations constitute the programs.

A distinguishing feature is that given a sentence to be evaluated, some of its symbols may or may not function as parameters, depending on the substitutions that are applied at runtime. That is, the parameters of an expression are not determined a priori by the language syntax but are resolved dynamically. Depending on what substitution is applied, a term may turn out to be either a parameter or the equivalent to the function or message name in other paradigms. This is an aspect by which we say that the syntax of the language is dynamic.

3.2 Text-Based Environment Specific Syntax

In textual mode we can tell the interpreter that a part of the sentence must be evaluated before evaluating the entire sentence. We call this early evaluation syntax. Alternatively we can tell the interpreter that a part of the sentence should be parsed as a block, as if it were one big indivisible symbol. This is analogous to the use of closures in other languages and we call it late evaluation syntax. Also we need a character to declare that a sentence ends. Table 5 presents these syntactic characters that are specific to the textual version of the language. Furthermore, whitespace plays an important role in the textual version. They separate the words, constituting them as symbols analogous to the cards of the graphical environment.

Table 5. Syntactic characters of the textual version

(	Early evaluation opening term
)	Early evaluation closing term
:	Late evaluation opening term
,	Late evaluation closing term
.	End of sentence term

3.3 Substitutions by Condition

There is a variant of the basic substitution rule. It is a rule that includes a condition of application.

As the fundamental relations of substitution and categorization, the substitution by condition is not a syntactic structure of flow control but a primitive relationship. As we showed in the boolean activity, control flow notation can be built from the two simplest relations of substitution and categorization. However, the usefulness of substitution rules with condition lies in simplifying the definition of some programs as shown in the robot activity.

3.4 Categories by Construction

In its textual version, the language allows to define categories with infinite elements by means of a specific syntax. In this way it was possible to define the category of natural numbers as lists of digits that cannot start with zero. Thus, fractions could also be defined as natural numbers joined by a slash '/' in a same word. These definitions are found in Supplemental Material.

3.5 Recursion

Since this general-purpose language does not provide flow control syntax for iteration, what it offers is recursion. In the activity of the numbers a simple definition of the sum has been presented in recursive form.

In Figure 10 we can see how a sum is resolved first by applying the recursive definition and then the base case.

3.6 Parameter Inference instead of Type Inference

In the rule defined by the sentence '1 + ℕ ☞ ℕ↑', the symbol 'ℕ' is functioning as the category on the left side and as the parameter name on the right side. While in functional programming the use of type inference has been extended in order to abbreviate writing, here it is proposed to dispense with naming parameters and to use their category directly. In the textual version, if one wanted to explicitly name a parameter, one could do so as shown in Table 4.

3.7 Alias for Categories

To gain declarativity, sometimes we can have different symbols that represent the same category. In the definition of 'if ... then ... else' two categories were drawn as comic book dialogue bubbles to represent any sentence. When they were defined, the categorization relationship was used in reverse. We placed the wildcard, which already categorized any sequence of symbols, 'inside' these new symbols. In this way, two new categories were created that also include any sequence of symbols. In this sense, we can say that the dialogue bubbles became aliases of the wildcard.

4 Implementation Remarks

This section includes brief remarks about implementation decisions that determine the behavior of the interpreter. The intention of the section is to deepen the explanation of the evaluation mechanism.

4.1 Substitution Rule Lookup

Given a concrete evaluation, a search is made for all the substitution relations that match with the sentence. At present, all substitution relations are global, that is, the scope of the search is always the same for all evaluations.

If there is no relation that applies to a sentence, it is evaluated by parts: first larger parts and, if there is no possible substitution, smaller parts. When this occurs, a word is first removed from the sentence and relations are sought. In this case there are two subordinates to try: without the first word or without the last, since the sequentiality of the symbols is respected. If there is no possible substitution, two words are removed, resulting in 3 variants: without the first two words, without the last two, or without the first and last. So successively, every sentence is attacked from the general to the particular. If any part is transformed, the whole is re-evaluated.

4.2 Matching of an Expression with a Substitution Rule

Although the combinations in which a relation can apply to a sentence are exponential, a polynomial algorithm is currently used. This algorithm identifies a variety of combinations, which is sufficient for the language to work. A more complex algorithm would offer even more flexibility in how programs are defined, promoting much less use of early and late evaluation syntax.

4.3 Selection of which Substitution Rule to Apply

When the interpreter evaluates an expression, it may happen that more than one substitution rule matches. To determine which rule to apply, the interpreter follows the steps described in Algorithm 1.

Algorithm 1 Select relation from: *matchingRelations* for sentence: *S*

```
R ← matchingRelations
R ←select relations with user defined high priority: R
R ←select relations from: R with more exact terms in: S
R ←select relations with less abstract categories from: R
R ←select relations with more conditions from: R
r ←detect first relation in alphabetical order from: R
return r
```

The first thing is to verify if any of these substitutions were defined as high priority by the user. The priority level of a substitution can be high, medium or low. By default

all substitutions have medium priority. Then the interpreter analizes the exact terms of coincidence between the relations and the sentence, the degree of abstraction of the categories that define the relations and, if any, the number of conditions in the definition of the relations. In each step there are fewer and fewer substitution rules left. In the end, if there is more than one substitution left, the first in alphabetical order is chosen, so the selection criterion is deterministic.

4.4 Articulation of the Two Fundamental Relations: Substitution and Categorization

The notion of 'substitution' refers directly to the evaluation mechanism of the language. An expression is substituted for others until there is nothing more to substitute, at which point the evaluation ends. For example, when the expression '1 + 2' is evaluated, the interpreter looks for substitution relations that match this expression. If there were a substitution relation for those exact terms, that would be the relation applied. In the last activity presented, between two defined relations that match, the one applied is the substitution relation that takes any expression of the form '1 + ℕ' and transforms it into 'ℕ↑'. Thus the expression '1 + 2' is transformed into '2↑'. In this case only the symbol 2 turned out to be a parameter of the evaluation. Either the symbols match exactly, that is, a symbol matches itself, or they can do so by category. Therefore, for the symbol '2' to match the symbol 'ℕ', there must be categorization relations that define that '2' is included in the category 'ℕ'. In this way the two fundamental relations between symbols are complemented.

Categorization and substitution relations are created by evaluating sentences. For example, the execution of the expression '1 + ℕ ☞ ℕ↑' creates the substitution relation applied in the previous example. Through the category 'ℕ' it was possible to match the expression '1 + 2' to that substitution relation.

The fundamental relationships of this language do not need to be syntactically reserved symbols to work. What makes them possible are substitution rules defined beyond *Representar* language. These are wired relationships, primitives, that are defined in the Smalltalk environment [9] on which our language is being implemented.

5 Written Words as the Symbols Defining the Program

In this section we present text-based programming tools for our language. We show that the relations of the graphical environment are implemented over textual relations. We also repeat the example of the booleans with the intention to show the graduality of the transition from visual to textual. Then we present three possible advanced activities. The purpose of this section is to show the textual version of the language, while continuing to analyze its potential, even in complex topics currently reserved for advanced programmers.

5.1 The Text-Based Programming Environment

Here we present the text programming editor and the substitution browser that goes with it.

The 'Representar Editor' allows us to program through text. It also supports symbols drawn in the graphical environment, allowing them to be copied and pasted, and interspersed with text. This helps in a gradual transition from the graphical to the textual experience.

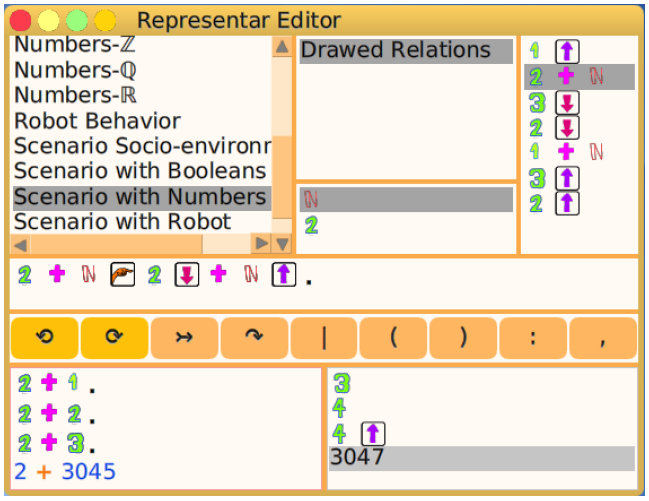


Figure 11. 'Representar Editor', the main programming tool.

The editor in Figure 11 has the same semantics as the graphical environment of the first part. Below it offers a REPL console, where the expressions on the left are evaluated. Results are displayed on the right and refreshed whenever the program changes. In the center is where the substitution and categorization relations are written. Above, these relations are ordered by topic. When choosing a topic (in Figure 11 the selected topic is 'Scenario with Numbers'), the sections and categories defined within that topic are displayed in the center. When choosing a section (in Figure 11 the selected section is 'Drawn Relations'), the substitution relations within that section are displayed on the right. When selecting a substitution of categorization relation, its definition is displayed in the center part to be read and modified.

In the center of the editor there is a bar with nine buttons. The first two on the left allow us to undo and redo changes in the substitution and categorization relations, as well as their organization in topics and sections. The rest of the buttons work as a keyboard, providing and also emphasizing the main characters of the language. Among these buttons, the three on the left are equivalent to the three fundamental cards of the drawing environment: point, categorize and the pipe to define substitutions by condition. The last four buttons on the right are the proper syntactic characters (see Table 5).

In the same way as in the graphical environment, when selecting a result in the lower right part of the editor, it is

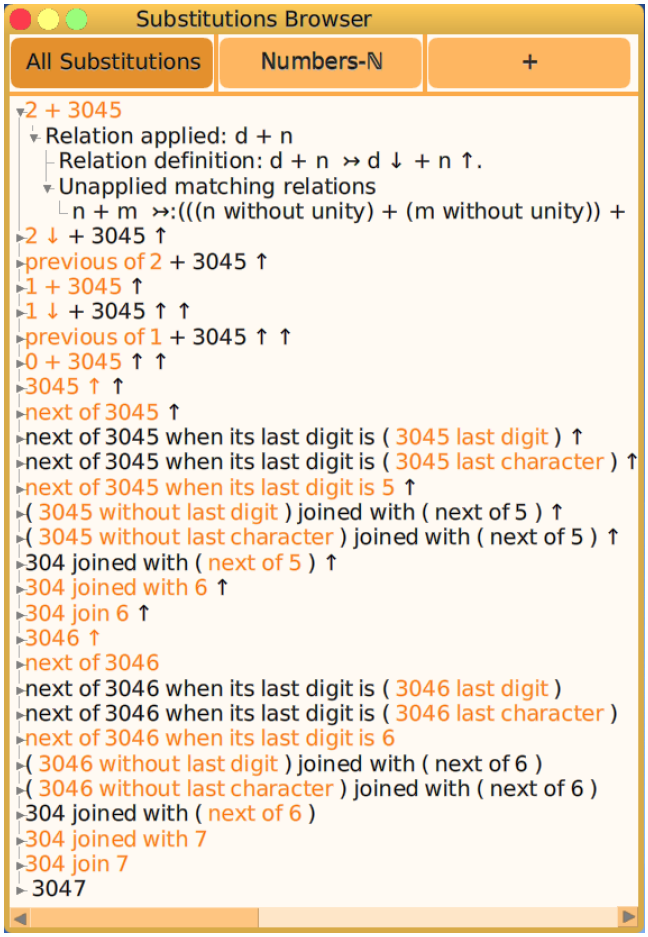


Figure 12. Substitutions Browser.

possible to inspect the substitutions that took place in its evaluation. In Figure 12 we can see the substitutions made to solve the sum of '2 + 3045'.

The 'Substitutions Browser' allows us to see all the transformations of the initial expression. For each substitution it is possible to hierarchically browse the applied relation, its definition, as well as the relations that matched with the expression but that were considered of lower priority by the interpreter's Algorithm 1.

In the textual environment we implemented a representation of integer and fractional numbers based on substitution and categorization relations. However, for didactic reasons it may be possible to start from an editor that has only the basic relations. This could be useful, for example, if we wanted to implement a representation of the natural numbers from scratch, continuing the last activity of drawing the numbers in a textual representation. Within the paper we will only repeat the activity of booleans and conditionals in the text-based environment. But any of the graphical activities presented can be repeated using text.

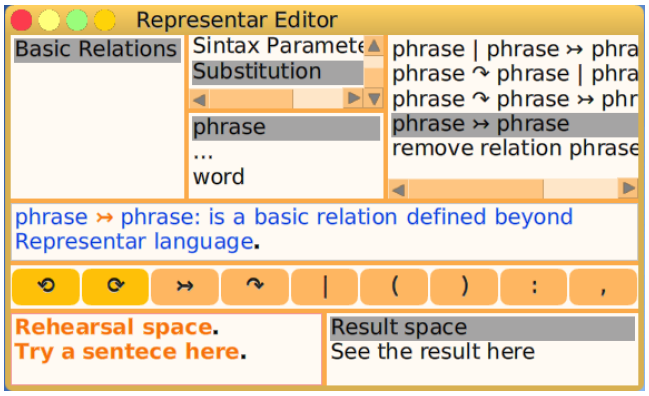


Figure 13. Editor with basic relations.

In Figure 13 we can see an editor that only has basic relations. Above, the most used basic substitution relation is selected. This relation allows any phrase to be replaced by any other phrase. The coloring of the texts located in the regions where one can write does not express the syntax of the language but the evaluation that text would have. So it is not syntactic highlighting but evaluation highlighting. Orange indicates words that match exactly with the first substitution rule that would be applied in case of evaluating the expression. Blue parts of the sentence indicate words that match by category, that is, parameters. Black indicates words that will not be transformed in the first substitution that applies.

In the center of the upper part of the editor we can see that the primitive categories are three: 'phrase' that matches with any non-empty sequence of symbols, '…' that matches with any sequence, whether it is empty or with symbols, and 'word' that matches with any individual symbol.

5.2 Graphical Notation Implemented over Textual Notation

The graphical notation is not only semantically the same as the textual one, but it is implemented on top of it and can be modified from the textual environment.

With the integrated execution of the graphical and textual environments in the same interpreter, we could define the basic relations for programming with drawings through textual relations. In other words we were able to build the basic relations of the graphic environment without leaving our language, which facilitated the development. In the same way, users can give semantics to their drawings through text when they feel limited by graphical tools.

In Figure 14 we can see how the substitution relation by condition is implemented with a simple substitution rule that replaces the graphical symbols with the textual notation.

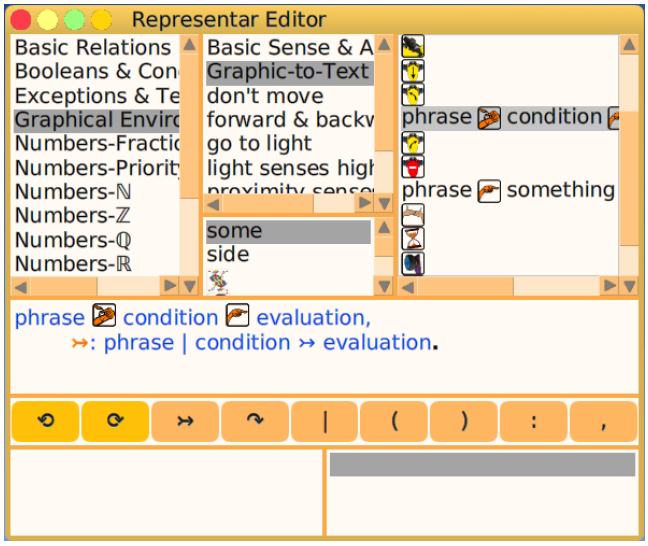


Figure 14. Implementation of a fundamental graphical relation of substitution over a textual relation.

5.3 Building Booleans and the Control Flow Notation with Text

The porpuse is to show the similarity between the graphical and textual implementation of the same model.

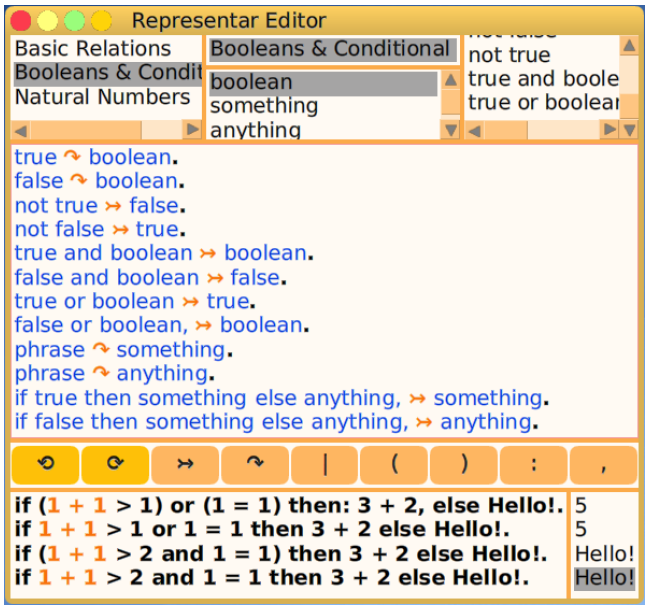


Figure 15. Implementation of booleans and 'if ... then ... else' with text

In the center of Figure 15 we can see exactly the same model of the board of Figure 7 expressed through text. Instead of drawn symbols we use words. This shows the continuity between the graphic and textual environment that the language facilitates. In the REPL console we define some expressions that show how the conditional and booleans work. This is implemented in a clean environment, which includes a simple natural number model. We can see that the newly created 'if ... then ... else' notation can work with or without parentheses thanks to the matching algorithms discussed in the previous section.

The implementation of the basic graphical relations on the basis of the textual primitives, as well as the implementation of the 'if ... then ... else', are demonstrations of the potential of the language in containing different domain-specific notations. This allows the language to host domain-specific sublanguages within the same environment.

5.4 Programming a Simple Model of Exceptions

The intention of this example is to show that the language allows to represent a complex topic in a few lines.

The teaching and use of exceptions is currently reserved for advanced programmers. How simple can an exception model be? In this language, a one-line exception model is presented. This simplicity would allow students to implement everything necessary for the language to support exceptions and thus easily understand the fundamentals of their use.

The fundamental characteristic of exceptions is that execution must be interrupted when an error occurs. Nothing else should be evaluated unless an exception is handled. If the exception is not caught, the evaluation must end. In this language, evaluation ends when no substitution is applied, so when an exception occurs, no more substitutions should be made. With the exception model presented, what happens when an exception is raised is that only substitutions that correspond to the exception model or that handle the raised exception take place.

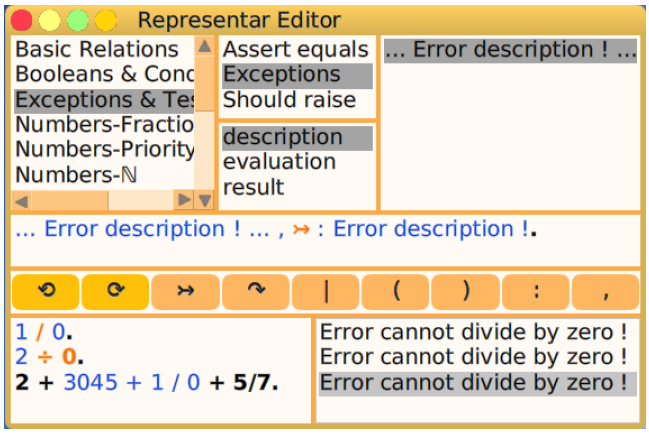


Figure 16. The model of exceptions.

Figure 16 shows the implementation of exceptions. The word 'description' is an alias for 'phrase', that is, any sequence of words matches with 'description'. According to this model, every exception must start with the word 'Error', then a description and finally an exclamation mark. That is

a convention established by this one-line implementation of exceptions. The written line is very easy to interpret, any expression of the form 'Error description !', no matter what is before or after, reduces itself by ignoring everything else around.

"There is a huge semantic gap between what the programmer knows about his program and the way he has to express this knowledge to a system for reasoning about that program. Languages which can narrow this gap are sorely needed" [28, p. 28]. If this need is still relevant, this example of an exception model shows the potential of *Representar*. A single line is enough to express that everything around an expression must be ignored. In comparison, the exception model of the Smalltalk environment where we implement the language [9] consists of 629 lines of code. This open model can be modified from the language itself but is much more complex.

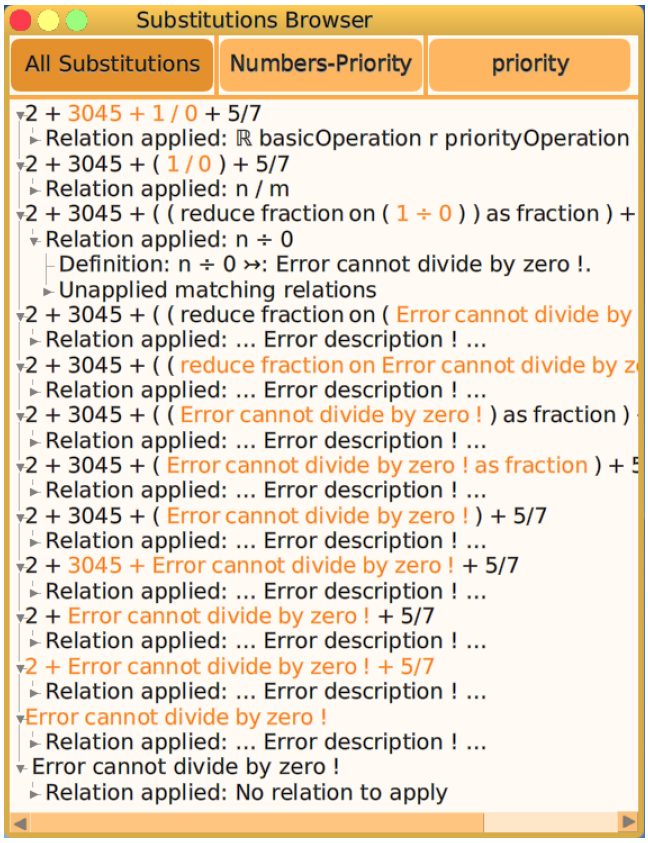


Figure 17. Evaluation that raises an exception.

In the REPL in Figure 16 we can see that different evaluations that include a division by zero result in the same exception. In Figure 17 we show that no other substitution relation is applied once the exception occurs. When the exception defined in the 'n ÷ 0' relation occurs, a process of successive evaluations is given where the substitution applied is always the one corresponding to the model of exceptions, until finally the execution ends signaling the error.

5.5 Defining a Simple Testing Model

The porpuse is to show the simplicity in using a model to build another model, in this case exceptions to build a testing model.

There are two essential functionalities to define tests, one that enables us to test that the result of an expression is the expected one, and another that allows testing that the expected error occurs. The first case can have a form similar to "assert expression equals result", the second a form of "should expression raise error description".

With two substitutions it's possible to implement the two aspects of the first functionality: 1) the one that returns the expected result and 2) the one that doesn't. If the result is not what is expected, the usual behavior of a testing framework is to throw an exception. Similarly, with two substitution relations we can implement both situations in which the expected error should occur.

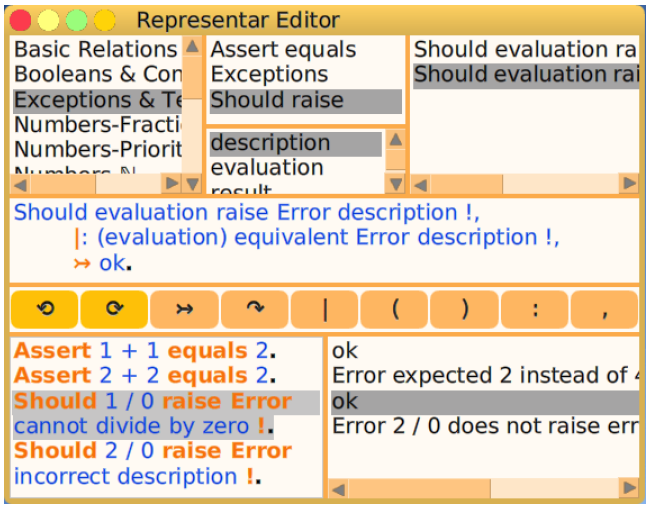


Figure 18. The testing model.

Figure 18 shows one of these four relations: the one that allows testing if an expression fails with the correct description. The definition shown in the center has an applicability condition that is declared between the pipe and the arrow. The complete model of testing can be seen in Supplemental Material.

Figure 19 shows the evaluation of the expression highlighted in gray in Figure 18. This is the test of an evaluation that fails as it should: raising an exception with the correct description.

5.6 Representing the Addition and Subtraction of Amounts of the Same Thing

The intention of this example is to show that the language allows to solve in a simple way a problem expressed colloquially, mixing words and mathematical symbols.

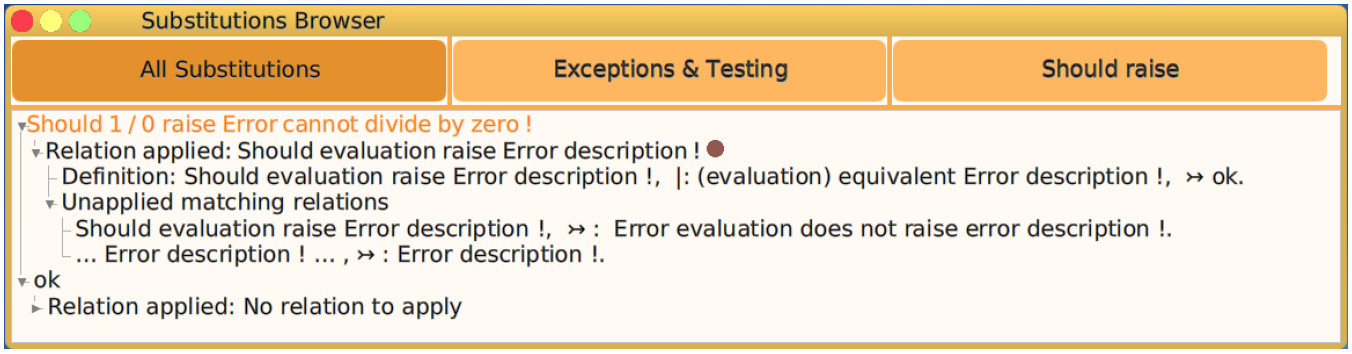


Figure 19. Substitutions in the testing of an expression that fails when it has to do so with the correct description.

We will represent the sum and subtraction of quantities of the same thing. This is a general case that admits a generic solution. When two quantities of the same thing are added or subtracted, the quantities must be added or subtracted directly and the result is obtained. It is a simple idea with a single substitution to express it (see Figure 20).

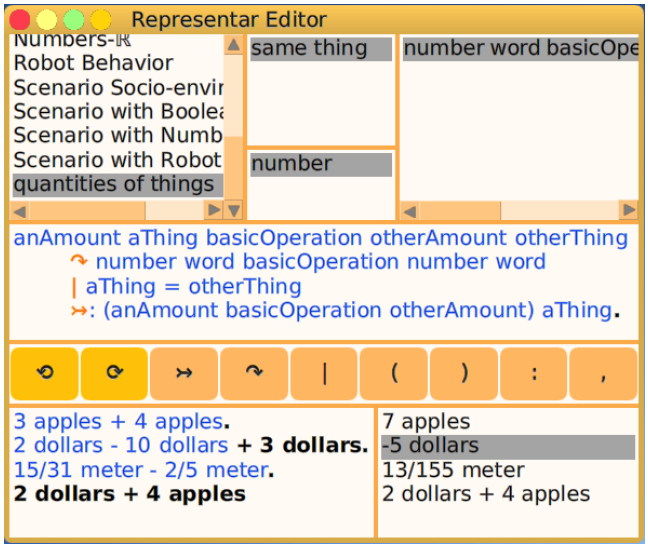


Figure 20. A single substitution relationship in order to add or subtract amounts of the same thing.

The substitution rule of Figure 20 is a conditional relation that declares particular names for each parameter at the beginning. These names are local, the scope does not go beyond the substitution rule. In the first line we declared the five parameters, in the second line, the categories of each parameter respectively, in the third line is the application condition, and in the fourth line we established how the new expression that replaces the matching phrase must be assembled. In Figure 21 we can see the sequence of substitutions for the second sentence evaluated in the REPL console of Figure 20.

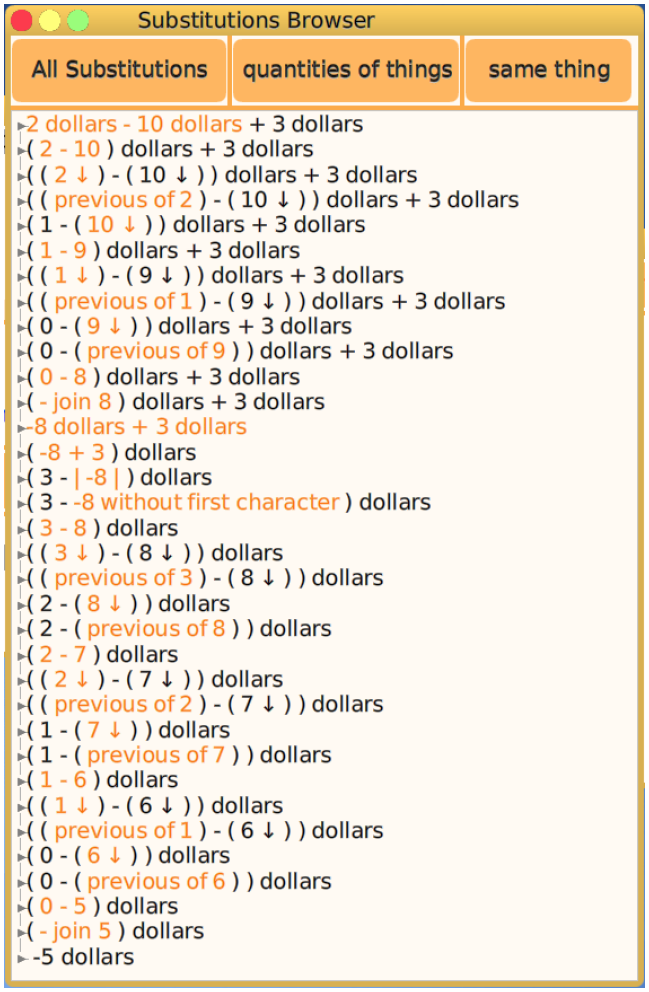


Figure 21. Substitutions that solve a sequence with operations of adding and subtracting amounts of the same thing.

The creation of a single substitution relation was enough to solve the sum and subtraction of quantities of the same thing, whatever it is. In this simple activity, the potential for reducing the semantic gap of the *Representar* language is

shown again. This happens both in the simplicity of the implemented model and the closeness to the natural language writing of the expressions that can be evaluated.

The same relation handles expressions with several terms. In the execution of Figure 21, the first and thirteenth line correspond to the application of the substitution rule defined in Figure 20. The addition and subtraction of expressions of different things was not altered, it remained something irreducible as shown in the last expression of the REPL console in Figure 19.

Having a simple model that is already functional is a great motivation to expand it in different directions. We can add relations to allow the addition and subtraction of different things that have a conversion factor, for example units of measure of the same type. We can incorporate relations of multiplication and division, applying arithmetic operations on quantities and on the things themselves.

6 Related Works

In this section we review related works in both dimensions of the project. First, we analyze current approaches of visual programming, where we compare our proposal with other approaches in computer science education. Then, we discuss related general purpose programming languages and analyze similarities and differences with ours.

6.1 Related Works in Visual Programming Oriented to Education

We will make an analysis and comparison with one of the most widespread tools in computer science education: Scratch [27] and other languages based on it, such as Snap! [10] and mBlock [18] which present themselves as improvements or adaptations for certain purposes. Also, we will analyze another relevant tool: Etoys [14] and derived works [32] [7].

The first thing to note is that both Scratch and Etoys as well as our project were initially implemented in Smalltalk. While Etoys remains only implemented in this way, Scratch has also been developed in javascript and can be used from a web browser. This is a strategic aspect that distinguishes Scratch project.

To make a comparison at the language level, it is pertinent to show a specific example. We will return to the case of programming a robot that must go towards the light and at the same time avoid obstacles. Our program consists of 5 lines shown in Figure 22 and already described in Table 3. In Figure 23 we can see a similar behavior of going towards the light combined with avoiding obstacles, defined with a Scratch version oriented to robot control: the mBlock Editor [18] for the mBot robot [26].

A common element of the compared approaches is that they all use drag and drop as a fundamental mechanism for composing programs. The other common aspect is that there are no error messages in any of them. All seek to

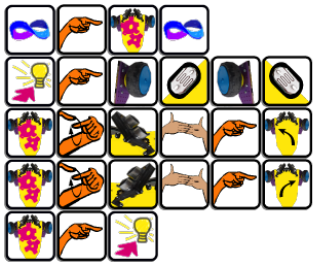


Figure 22. Robot behavior defined in our language and described in Table 3.

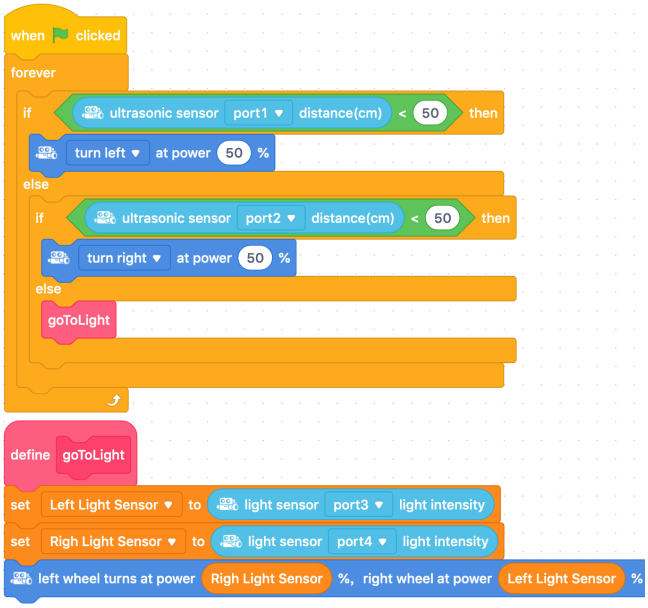


Figure 23. A robot behavior analogous to that in Figure 22 defined with Scratch-style visual blocks.

minimize the basic building blocks that allow expressions to be combined. And all have 'liveness' as a design principle [19], which means there is no compilation step or edit/run mode distinction.

A difference at first sight is that our visual language does not use any text, while Scatch, and also Etoys, need text to give meaning to their visual structures. Around Etoys there has been an attempt to address this issue in a project called V-Toys [7]. This tool is equally expressive to Etoys but purely visual. Other Etoys-derived project called TileScript [32] proposes a transition from Etoys scripts to purely textual scripts. It tries to power up Etoys at the cost of increasing its complexity. There is only one article that mentions each of these projects and there is no prototype available to try them.

In our project, the lack of text does not prevent us from a central quality in programming languages, which is being able to define abstractions. In our language, like Scratch and

Etoys, we were able to define a simple abstraction, with no parameters, that represents the behavior of going towards the light. It was not necessary to resort to text for this, it was enough to draw the concept as a card.

Regarding parameterization, Scratch and Etoys abstractions allow a limited set of parameter types, in the case of Scratch only three (booleans, strings, and numbers). In our language we can parameterize any sequence of symbols without distinction between code and data. The visual language that has deepened in this sense is Snap! [10] allowing to dynamically parameterize even portions of its own code.

Another abstraction mechanism of our language are user-defined categories. In Scratch and Etoys we cannot create new types. In Snap! it is possible to define new data types through lists but at the cost of complexity. Our visual language offers great power of abstraction while keeping it simple.

In our proposal the user does not have to go beyond the symbols to test the language. We offer a fully graphical REPL console for this. We also offer the possibility of inspecting the execution of the program without resorting to any textual representation. These are novelties in visual languages designed for teaching, along with the possibility of defining categories with a single combinator. And another element that stands out is that while in Scratch and Etoys the control of the flow is explicit in the program, in our language this is implicit and is the interpreter's responsibility.

In Scratch and Etoys it is possible to draw the cartoon character that the program controls, while in our prototype this is not yet possible. Both Scratch and Etoys allows to have unused modules in the programming environment, which makes it easy to roll back or test changes. Our prototype does not offer this yet, so deleting relationships requires rewriting them to roll back. This reveals that our project has not yet been used outside the laboratory.

6.2 Related Works on General-Purpose Programming Languages

The history of programming languages shows a breadth and variability in the possibilities of high-level languages. Recent examples include Super-Glue [23] or Subtext [8]. Cases like Forth [24] or APL [13] early showed the possibility of really different variants. In the case of Forth, its conceptual unit is the word, understood in the same way as in *Representar*: a sequence of characters delimited by spaces. Thus, the flexibility of Forth's syntax is a common feature with *Representar*. On the other hand, Forth, based on stacks, achieved a high-level approach to symbol manipulation combined with a very close relationship to machine language and, therefore, performance. Unlike *Representar*, Forth did not aim to capture the way humans represent knowledge or manipulate symbols beyond the computer. On the contrary, it has been presented as a particular way of "thinking about software problems" [3, p. 119].

A branch of languages that seeks to emulate the human decision process are rule-based programming languages. Among them Prolog, CLIPS and OP5 stand out. These languages are associated with the development of expert systems: "[t]he rule-based approach has often been chosen for programming expert systems because the pattern-action model resembles human decision processes" [11, p. 1]. Since they have a rule-based input/output transformation mechanism, they are related to our language. However, rule-based languages and systems, between other shared properties, "[t]hey incorporate practical human knowledge in conditional if-then rules" [16, p. 921]. While in *Representar* substitution rules can eventually present boolean conditions, in rule-based programming all rules are defined based on conditions. In the case of *Representar*, the distinctive aspect of its substitution rules is that they match by category with the input, a substantially different kind of rule. Rule-based languages also allow to define 'facts' which, combined with inference rules, they enables to solve queries. Our language, although it is based on substitution rules, does not solve the evaluation with an inferential mechanism.

The substitution mechanism of our language is inspired by the 'Substitution Model' studied by Abelson and Sussman [1]. This specific aspect of our language is related to functional programming. In Figure 24 we can see that in their book the substitution mechanism is presented as a programmer's deduction. Instead, in our language, through the Substitution Browser tool, it becomes the responsibility of the computer, freeing the programmer from the work of having to mentally reconstruct the substitution process. Another relationship with languages such as Lisp and Scheme, is that *Representar* is homoiconic: the same structures allow to represent both code and data. In the case of *Representar*, these structures are simply the words delimited by blanks.

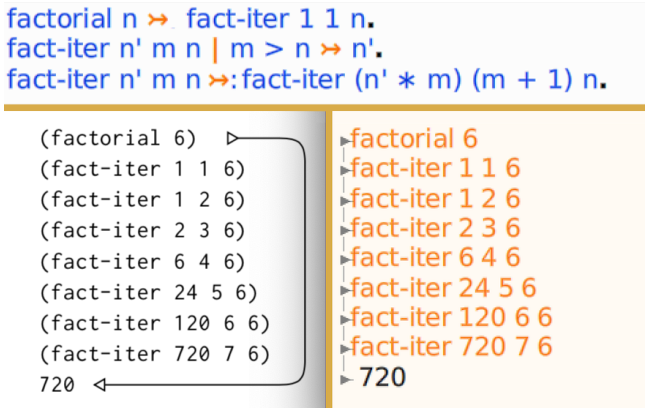


Figure 24. Above our factorial function implemented 'iteratively' (following [1]). Below left a representation of the substitution mechanism extracted from the book by Abelson and Sussman[1, p. 43]. Below right is a representation auto-generated by our tools of this substitution process.

The branch of languages that is historically linked to a theory of knowledge is object-oriented programming. Simula-67 incorporated the notion of class and object [5]. Smalltalk kept the same classical approach with two different fundamental relationships for sharing behavior: subclassification and instantiation. Later Self [31] adopted a modern approach, establishing a single sharing mechanism, the parent delegation between objects. Through this relationship one object can be an exemplar of another, or else represent an abstract notion under which another object is categorized. Which of them is established is not declared in the language but is left to the programmer's interpretation. In this sense, our language is related to Self, since it offers a unique way of defining flexible categories, based on prototypes or more abstract notions. The clear difference is that our categories are between symbols, not objects. And we cannot say that they are a way of sharing behavior but rather a way of expanding the range of application of the substitution rules. Like Self and Smalltalk, our language is part of a tradition of dynamic languages, which prioritizes flexibility and freedom of expression over preventing errors through static checks.

Simula-67 and its successors are based on a classical theory of knowledge. Self and its successors are inspired by modern approaches from cognitive psychology [29] . Our language is based on a theory developed by ourselves and presented in previous works [21] [22]. In these articles we argue that human symbolic representation is not innate, it is constructed through experience (unlike the honeybee that also collaborates through symbolic communication [15]). Nevertheless, we consider that there is a natural basis for human symbolic communication, and it is given by two relationships between symbols: substitution and categorization. The substitution relationship in particular has an extra quality: it can reference beyond symbols, to the world. That is, a word or a set of words can substitute for another set of words or reference an entity or process beyond symbols. The *Representar* language is an implementation and a complement of this theory of knowledge.

7 Future Works

The immediate future work is to test the use of the system built by children guided by a teacher in a didactic context. By doing so, future activities we design could allow users to draw a character in the simulator, program its behavior and interaction with mouse and keyboard events, allowing to build interactive animations. Another feature that could be added is to allow the user to quantify visually. We could incorporate a card that allows adjusting the amount of what another card expresses. This new card would be like a speedometer whose needle can be manipulated by the user.

In this project we have reduced the problem of changing the language when going from simpler to more complex programs. This problem was replaced by a two-mode scheme, one visual and one textual. Each mode has its own distinct set of tools. Following Larry Tesler [30], a future work is to advance in the integration of the tools, in order to offer a modeless solution. This is a design challenge now before us.

However, the vision of the project goes much further than what has been built. The graphical environment has the conditions to be implemented as a Tangible User Interface [12]. It could work as a magnetic construction game combined with drawing on paper with crayons. Figure 25 shows a staging of how this programming environment might look like. The program is the same as the robot activity in the first section. There would be a camera and a video projector on the ceiling to project "copies" of drawings made on paper onto the blank magnetic cards.

Figure 25. Photomontage of how programming with drawings could look as a Tangible User Interface.

8 Final Words

Representar constitutes a new paradigm of programming that combines both simplicity and expressiveness. The original contribution lies in the fact that it is based on two fundamental relations between symbols: substitution and categorization. Its versatility allows it to work in a visual mode, with great power of abstraction and without the use of text to make sense of its components. The execution is based on a model of substitutions that can be visualized automatically, freeing the programmer from the task of mentally reproducing it. This debugging tool is available in visual mode, constituting a novelty in languages oriented to education. The other original contribution to visual languages is a console that allows us to execute purely graphical expressions. In text mode it has all the power of a conventional language

and it allows complex problems to be defined concisely and declaratively. It has a very simple syntax, which goes little further than delimiting words with white space.

Acknowledgments

This material is based upon work partially funded by the *Régimen de Participación Cultural de la Ciudad Autónoma de Buenos Aires* under Grant No. EX-2021-18985508-GCABA-DGDCC and Grant No. EX-2022-29729661-GCABA-DGDCC. The author would like to thank Hernán Wilkinson, Juan Burella, Javier Legris, Guillermo Folguera, David Ungar, Dionisio Martínez and *Fundación Argentina de Smalltalk*.

References

[1] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and interpretation of computer programs.* The MIT Press.

[2] Valentino Braitenberg. 1986. *Vehicles: Experiments in synthetic psychology.* The MIT Press.

[3] Leo Brodie. 2004. *Thinking forth.* Punchy Pub.

[4] Gabriele Contessa. 2013. *Models and Maps: An Essay on Epistemic Representation.* unpublished manuscript, Carleton University, Ottawa, ON. Retrieved July 6, 2022 from https://philarchive.org/archive/CONMAM-9

[5] Ole-Johan Dahl. 2004. The birth of object orientation: the simula languages. *From Object-Orientation to Formal Methods. Lecture Notes in Computer Science.* 2635 (2004), 15–25. https://doi.org/110.1007/978-3-540-39993-3_3

[6] Mark Dorling and Dave White. 2015. Scratch: A Way to Logo and Python. *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), 191–196. https://doi.org/10.1145/2676723.2677256

[7] Pierre-André Dreyfuss and Serge Stinckwich. 2008. V-Toys: An Experiment in Adding Visual Tiles to EToys. In *Sixth International Conference on Creating, Connecting and Collaborating through Computing (C5 2008).* 165–171. https://doi.org/10.1109/C5.2008.22

[8] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), 505–518. https://doi.org/10.1145/1094811.1094851

[9] Hilaire Fernandes, Ken Dickey, and Juan Vuletich. 2020. *The Cuis-Smalltalk book.* Retrieved November 4, 2021 from https://github.com/Cuis-Smalltalk/TheCuisBook

[10] Brian Harvey and Jens Mönig. 2015. Lambda in blocks languages: Lessons learned. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* 35–38. https://doi.org/10.1109/BLOCKS.2015.7368997

[11] Frederick Hayes-Roth. 1985. Rule-Based Systems. *Commun. ACM* 28, 9 (sep 1985), 921–932. https://doi.org/10.1145/4284.4286

[12] Hiroshi Ishii. 2008. The Tangible User Interface and Its Evolution. *Commun. ACM* 51, 6 (jun 2008), 32–36. https://doi.org/10.1145/1349026.1349034

[13] Kenneth E. Iverson. 1965. A programming language. *AIEE-IRE '62 (Spring): Proceedings of the May 1-3, 1962, spring joint computer conference* (1965), 345–351. https://doi.org/10.1145/1460833.1460872

[14] Alan Kay. 2005. Squeak etoys, children and learning. *Viewpoints Research Institute* (2005). Retrieved August 31, 2022 from http://www.vpri.org/pdf/rn2005001_learning.pdf

[15] Patrick L. Kohl, Neethu Thulasi, Benjamin Rutschmann, Ebi A. George, Ingolf Steffan-Dewenter, and Axel Brockmann. 2020. Adaptive evolution of honeybee dance dialects. *Proceedings of the Royal Society B* 287, 20200190 (2020), 1–9. https://doi.org/10.1098/rspb.2020.0190

[16] Thaddeus J. Kowalski and Leon S. Levy. 1996. *Rule-based programming.* AT&T, USA.

[17] George Lakoff. 1987. *Women, fire, and dangerous things. What categories reveal about the mind.* University of Chicago Press. https://doi.org/10.7208/chicago/9780226471013.001.0001

[18] Makeblock 2022. *mBblock Editor.* Retrieved July 1, 2022 from https://ide.mblock.cc/

[19] John H. Maloney and Randall B. Smith. 1995. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology.* 21–28. https://doi.org/10.1145/215585.215636

[20] Fred Martin. 1996. Kids learning engineering science using LEGO and the programmable brick. *Proc of AERA* (1996).

[21] Agustín Rafael Martínez. 2020. Integración del conocimiento científico y materialismo dialéctico. *Hic Rhodus. Crisis Capitalista, Polémica y Controversias* 19 (2020), 23–43. https://publicaciones.sociales.uba.ar/index.php/hicrhodus/article/download/6161/5117

[22] Agustín Rafael Martínez. 2021. Representación simbólica y materialismo dialéctico. De la comunicación simbólica a la programación de computadoras. *Hic Rhodus. Crisis Capitalista, Polémica y Controversias* 20 (2021), 59–78. https://publicaciones.sociales.uba.ar/index.php/hicrhodus/article/download/6644/5553

[23] Sean McDirmid. 2007. Living it up with a live programming language. *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications* (2007), 623–638. https://doi.org/10.1145/1297027.1297073

[24] Charles H. Moore. 1974. FORTH: a new way to program a mini computer. *Astronomy and Astrophysics Supplement Series* 15, 497 (1974).

[25] Mark Noone and Aidan Mooney. 2018. Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education* 5, 2 (2018), 149–174. https://doi.org/10.1007/s40692-018-0101-5

[26] Jelena Pisarov and Gyula Mester. 2019. Programming the mbot robot in school. *Proceedings of the International Conference and Workshop Mechatronics in Practice and Education, MechEdu* (2019).

[27] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[28] Tim Sheard. 2004. Languages of the future. *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2004), 116–119. https://doi.org/10.1145/1028664.1028711

[29] Antero Kyösti P Taivalsaari. 1997. Classes vs. Prototypes – Some Philosophical and Historical Observations. *Journal of Object-Oriented Programming* 10, 7 (Nov. 1997), 44–50.

[30] Larry Tesler. 2012. A Personal History of Modeless Text Editing and Cut/Copy-Paste. *Interactions* 19, 4 (jul 2012), 70–75. https://doi.org/10.1145/2212877.2212896

[31] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. *SIGPLAN Not.* 22, 12 (dec 1987), 227–242. https://doi.org/10.1145/38807.38828

[32] Alessandro Warth, Takashi Yamamiya, Yoshiki Ohshima, and Scott Wallace. 2008. Toward A More Scalable End-User Scripting Language. In *Sixth International Conference on Creating, Connecting and Collaborating through Computing (C5 2008).* 172–178. https://doi.org/10.1109/C5.2008.33

A Supplemental Material

This appendix provides a file-out of the models implemented in the *Representar* language that have been shown in the paper: the model of exceptions and testing and the model of natural and fractional numbers. Also, in Figure 27, is a list of the basic or primitive relations of the *Representar* language.

In Figure 28, the category of natural numbers is partially defined by extension by declaring that 'digit' is within the category 'ℕ'. But it is also defined by "construction" by declaring how a word must be constructed to be considered a natural number: it must start with a digit that is not zero and the rest of its characters must be a list of digits.

As can be seen in Figure 28, the last categories in the list have been defined using a syntax similar to mathematics rather than using the language notation. This has been possible through the substitution relations of Figure 29. This is an example of how the language allows itself to be syntactically adapted to different domains.

At the end of Figure 35, a relation was included that allow parentheses to be dispensed with, in order to ensure the priority of multiplication and division over addition and subtraction in an expression. This is another example of the adaptability of the language to the syntactic conventions of a specific domain.

The evaluation of the text of all the figures, allows us to load the models in their correct topic and section within an empty 'Representar Editor' like the one in Figure 13.

```
Topic name is Exceptions & Testing.
description is an alias of phrase.
phrase ↷ evaluation.
result is an alias of phrase.


Section name is Assert equals.
Assert evaluation equals result, ↛:
        Error expected result instead of
        (evaluation) on expression: evaluation !.
Assert evaluation equals result,
    |: (evaluation) = result,
    ↛ ok.

Section name is Exceptions.
... Error description ! ... , ↛ : Error description !.

Section name is Manipulation.

Section name is Should raise.
Should evaluation raise Error description !,
    |: (evaluation) equivalent Error description !,
    ↛ ok.
Should evaluation raise Error description !,
    ↛ :
    Error evaluation does not raise error description !.
Should evaluation raise Error description !,
    |: (evaluation) equivalent Error description !,
    ↛ ok.
```

Figure 26. Exception and testing models

```
Section name is Substitution.
remove relation phrase
phrase ↷ phrase | phrase ↛ phrase
phrase ↛ phrase
phrase ↷ phrase ↛ phrase
phrase | phrase ↛ phrase

Section name is Categorization.
word ∈ word ?
no more word ↷ word
word definition by construction phrase
word ↷ word

Section name is Symbol Manipulation.
word without last character
word join word
word without first character
word last character
word first character

Section name is Organization.
Topic to which belongs relation phrase
Section to which belongs relation phrase
Section name is phrase
Topic name is phrase

Section name is Performance.
ℝ < ℝ

Section name is Regularities & Invariants.
all the time phrase

Section name is Sintax Parameterization.
word is a term with its own meaning
word is an early evaluation closing term
word is an early evaluation oppening term
word now indicates the opening of a late evaluation
word now indicates the closing of a late evaluation
word is not a term with its own meaning
word now indicates the end of a sentence

Section name is File Management.
bring out all as phrase representation
bring in phrase representation
```

Figure 27. Predefined basic relation of the language.

```
Topic name is Numbers-ℕ.

1 ↷ nonZeroDigit.
2 ↷ nonZeroDigit.
3 ↷ nonZeroDigit.
4 ↷ nonZeroDigit.
5 ↷ nonZeroDigit.
6 ↷ nonZeroDigit.
7 ↷ nonZeroDigit.
8 ↷ nonZeroDigit.
9 ↷ nonZeroDigit.
nonZeroDigit ↷ digit.
0 ↷ digit.
digit ↷ ℕ.
ℕ definition by construction: nonZeroDigit [digit].
∀ n ∈ ℕ.
∀ m ∈ ℕ.
∀ n' ∈ ℕ.
∀ natural ∈ ℕ.
∀ d ∈ digit.
∀ d' ∈ digit.
∀ w' ∈ word.
∀ w ∈ word.
```

Figure 28. Categories defined in the implementation of natural numbers.

```
Section name is Next & Previous.
next of 0 ↣ 1.
next of 1 ↣ 2.
next of 2 ↣ 3.
next of 3 ↣ 4.
next of 4 ↣ 5.
next of 5 ↣ 6.
next of 6 ↣ 7.
next of 7 ↣ 8.
next of 8 ↣ 9.
previous of 1 ↣ 0.
previous of 2 ↣ 1.
previous of 3 ↣ 2.
previous of 4 ↣ 3.
previous of 5 ↣ 4.
previous of 6 ↣ 5.
previous of 7 ↣ 6.
previous of 8 ↣ 7.
next of n ↣: next of n when its last digit is (n last digit).
n ↓ | ones of n = 0
      ↣: ((n without ones) ↓) joined with 9.
n ↓ | ones of n = 0
      ↣: ((n without ones) ↓) joined with 9.
n ↓ ↣: (n without ones) joined with ((ones of n) ↓).
nonZeroDigit ↓ ↣ previous of nonZeroDigit.
10 ↓ ↣ 9.
n ↑ ↣ next of n.
next of n when its last digit is d
      ↣: (n without last digit) joined with (next of d).
next of 9 ↣ 10.
next of n when its last digit is 9
      ↣: (next of (n without last digit)) joined with 0.
```

Figure 30. Next and previous. Relationships necessary to later address arithmetic operations.

```
Section name is Set Notation.
∀ name ∈ set
      ↷    ∀ word ∈ word
      ↣: set ↷ name.
word is an alias of category
      ↷    word is an alias of word
      ↣: category ↷ word.
element ∈ set
      ↷    word ∈ word
      ↣: element ↷ set.
subset ⊂ set
      ↷    word ⊂ word
      ↣: subset ↷ set.
```

Figure 29. Notation of the domain of mathematics implemented on the language. The first relation is the one used to define the last eight categories of Figure 28.

```
Section name is +.
d + n ↣ d ↓ + n ↑.
n + m ↣:(((n without unity) + (m without unity)) +
          tens of ((unity of n) + (unity of m)))
          joined with
          (unity of ((unity of n) + (unity of m))).
0 + n ↣ n.
n + 0 ↣ n.
n + d ↣ d ↓ + n ↑.
```

Figure 31. The sum of natural numbers that takes into account the positionality of the digits as we do when we calculate by hand.

```
Section name is -.
n - d ↣: (n↓) - (d↓).
n - 0 ↣ n.
n - m | n = m ↣: 0.
0 - 0 ↣ 0.
n - m
      | n > m & ones of n >= ones of m
      ↣: ((n without ones) - (m without ones))
            joined with ((ones of n) - (ones of m)).
m - n
      | m > n & ones of m < ones of n
      ↣: (((m without ones) - 1) - (n without ones))
            joined with (((ones of m) + 10) - (ones of n)).
n - m | n = m ↣: 0.
d - n ↣: (d↓) - (n↓).
```

Figure 32. Subtraction of natural numbers.

```
Section name is ∗.
0 ∗ n ↣ 0.
n ∗ 0 ↣ 0.
n · m ↣ n ∗ m.
d ∗ n ↣: n + (n ∗ (d↓)).
n ∗ m ↣: ((n ∗ (m without ones)) joined with 0)
                  +
                  (n ∗ (ones of m)).
n ∗ d ↣ d ∗ n.
```

Figure 33. Multiplication of natural numbers.

```
Section name is ÷.
n ÷ 0 ↣: Error cannot divide by zero !.
0 divided by n ↣ 0.
n divided by m ↣ n join / join m.
m ÷ n | m = n ↣ 1.
n divided by 1 ↣ n.
n / m ↣: (reduce fraction on (n ÷ m)) as fraction.
n ÷ m | n > m
      ↣: ((first (m size + 1) digits of n) ÷ m)
            continue with rest of dividend
            (last (n size - (m size + 1)) digits of n)
            and divisor m.
n ÷ m |: n > m & (n size - 1) <= (m size),
      ↣: 1 + ((n - m) ÷ m).
m divided by n | m = n ↣ 1.
n ÷ 1 ↣ n.
n + f continue with rest of dividend w and divisor m
      ↣: (n with (w size) zeros) +
            (reduce fraction on (((f numerator)
                  · (m ÷ (f denominator))) join w) ÷ m).
n continue with rest of dividend w and divisor m
      ↣: (n with (w size) zeros) +
        (reduce fraction on (remove zeros from left of w) ÷ m).
n ÷ m ↣ n join / join m.
n divided by m
      |: n > m & (n size - 1) <= (m size),
      ↣: 1 + ((n - m) divided by m).
n divided by m | n > m
      ↣: ((first (m size + 1) digits of n) divided by m)
            continue with rest of dividend
            (last (n size - (m size + 1)) digits of n)
            and divisor m.
0 ÷ n ↣ 0.
```

Figure 34. Division of natural numbers.

```
Topic name is Numbers-ℝ.
Fraction ↷ ℝ.
ℤ ↷ ℝ.
∀ r ∈ ℝ.
∀ r' ∈ ℝ.

Section name is negation.
negate r + phrase, ↣: - r - phrase.
negate r - phrase, ↣: - r + phrase.
negate r ↣ - r.

Topic name is Numbers-Priority.
+ ↷ basicOperation.
- ↷ basicOperation.
∗ ↷ priorityOperation.
/ ↷ priorityOperation.
÷ ↷ priorityOperation.

Section name is priority.
ℝ basicOperation r priorityOperation r'
      ↣: ℝ basicOperation (r priorityOperation r').
```

Figure 35. The real numbers up to where they were implemented.

```
Section name is Comparing.
symbol to represent falsehood ↣ F.
symbol to represent truth ↣ T.
n > m | n size = m size ↣ first digit of n > first digit of m.
d > d' ↣: (d↓) > (d'↓).
w = w' | w = w' ↣ T.
n <= m ↣ n < m.
n > m ↣: (n size) > (m size).
n > m | n size = m size & first digit of n = first digit of m
      ↣:    (remove zeros from left of (n without first digit)) >
            (remove zeros from left of (m without first digit)).
n > m | n size = m size
      ↣ first digit of n > first digit of m.
w = w' | w = w' ↣ T.
n < m ↣ m > n.
0 > n ↣ F.
n > 0 ↣ T.
w = 0 ↣ F.
d > n ↣ F.
0 = 0 ↣ T.
n <= m | n = m ↣ T.
n > m | n size = m size & first digit of n = first digit of m
      ↣:    (remove zeros from left of (n without first digit)) >
            (remove zeros from left of (m without first digit)).
n >= m ↣ n > m.
n >= m | n = m ↣ T. n >= m | n = m ↣ T.
n > d ↣ T. n = m ↣: (n - m) = 0.
0 > 0 ↣ F. n = 0 ↣ F. n <= m | n = m ↣ T.
```

Figure 36. Comparison of natural numbers.

```
Section name is Construction of Numbers.
w size ↣ calculate w size.
first 1 digits of w ↣ w first character.
n with 0 zeros ↣ n.
first m digits of w
      ↣: (w first character)
            join
            (first (m - 1) digits of (w without first character)).
remove zeros from left of w
      | w first character = 0
      ↣: remove zeros from left of (w without first character).
n without ones
      ↣ n without last character.
0 joined with n, ↣ n.
n size ↣: 1 + ((n without ones) size).
+ size ↣ 1.
n joined with m ↣: n join m.
n last digit ↣ n last character.
natural without last digit ↣ natural without last character.
n without first digit ↣ n without first character.
last m digits of w ↣:
            (last (m - 1) digits of (w without last character))
            join
            (w last character).
tens of d ↣ 0.
digit size ↣ 1.
last 1 digits of w ↣ w last character.
n with m zeros ↣: (n joined with 0) with (m - 1) zeros.
n without last digit ↣ n without last character.
tens of n ↣ unity of n without last digit.
remove zeros from left of w ↣ w.
remove zeros from left of w
      | w first character = 0
      ↣: remove zeros from left of (w without first character).
unity of n ↣ n last digit.
remove zeros from left of 0 ↣ 0.
first digit of n ↣ n first character.
n without unity ↣ n without last digit.
ones of n ↣ n last character.
calculate size ↣ 0.
calculate w size
      ↣: 1 + (calculate (w without last character) size).
natural last digit ↣ natural last character.
```

Figure 37. Auxiliary relationships linked to properties of natural numbers as well as to their construction.

```
Section name is Greatest Common Divisor.
GCD n m with remainder 0 ↣ m.
GCD n m ↣: GCD n m with remainder (n \\ m).
GCD n m with remainder n'
      ↣: GCD m n' with remainder (m \\ n').
```

Figure 38. Greatest Common Divisor

```
Section name is Quotient & Remainder.
quotient of n + f ↣ n.
reminder of n + f when m is the divisor
      ↣ reminder of f when m is the divisor.
quotient of n ↣ n.
reminder of f when m is the divisor | f denominator = m
      ↣ f numerator.
quotient of f ↣ 0.
n // m ↣: quotient of (n ÷ m).
reminder of f when m is the divisor
      ↣: (f numerator) · (m ÷ (f denominator)).
n \\ m ↣: reminder of (n ÷ m) when m is the divisor.
reminder of n when m is the divisor ↣ 0.
```

Figure 39. Quotient and remainder

```
Topic name is Numbers-Fractions.
PositiveFraction definition by construction: ℕ / ℕ.
PositiveFraction ↷ f.
NegativeFraction ↷ nf'.
NegativeFraction ↷ Fraction.
PositiveFraction ↷ Fraction.
NegativeFraction ↷ nf.
PositiveFraction ↷ f'.
NegativeFraction definition by construction:
      - PositiveFraction.

Section name is +.
f + f' ↣: ((((f numerator)·(f' denominator))
      + ((f' numerator)·(f denominator)))
            / ((f denominator)·(f' denominator))) as fraction.

Section name is -.
nf - nf' ↣: nf + (negate nf').
f - f' ↣: ((((f numerator)·(f' denominator))
      - ((f' numerator)·(f denominator)))
            / ((f denominator)·(f' denominator))) as fraction.
negate nf ↣ nf without first character.

Section name is ·.
f ∗ f' ↣: ((f numerator)·(f' numerator))
      ÷ ((f denominator)·(f' denominator)).
f · f' ↣: ((f numerator)·(f' numerator))
      / ((f denominator)·(f' denominator)).

Section name is ÷.
f / f' ↣: (((f numerator)·(f' denominator))
      / ((f denominator)·(f' numerator))) as fraction.
f ÷ f' ↣: (((f numerator)·(f' denominator))
      / ((f denominator)·(f' numerator))) as fraction.

Section name is Converting.
n as fraction ↣ n.
f as fraction ↣ f.
n + f as fraction ↣: ((n · (f denominator)) +
      (f numerator)) join / join (f denominator).
z as fraction ↣ z.

Section name is Numerator & Denominator.
extract denominator from n ↣ n.
extract numerator from w ↣:
      extract numerator from (w without last character).
extract numerator from n ↣ n.
f denominator ↣ extract denominator from f.
extract denominator from w ↣:
      extract denominator from (w without first character).
f numerator ↣ extract numerator from f.
```

Figure 40. Fractions.

```
Section name is Reducing.
m ÷ n can reduce ↠: (GCD m m) > 1.
m divided by n can reduce ↠: (GCD m m) > 1.
reduce fraction on n ↠ n.
reduce n divided by m ↠:
      (n divided by (GCD n m))
            join / join (m divided by (GCD n m)).
n reduced ↠ n.
reduce n / m ↠: (n ÷ (GCD n m)) join / join (m ÷ (GCD n m)).
reduce n ÷ m ↠: (n ÷ (GCD n m)) join / join (m ÷ (GCD n m)).
reduce fraction on n / m ↠: (n ÷ (GCD n m)) / (m ÷ (GCD n m)).
reduce fraction on n ÷ m
      ↦: (n ÷ (GCD n m)) ÷ (m ÷ (GCD n m)).
phrase - f | f can reduce ↠: phrase - (f reduced).
reduce fraction on n divided by m ↠:
      (n divided by (GCD n m))
            divided by (m divided by (GCD n m)).
reduce fraction on f ↠ f reduced.
f can reduce ↠: (GCD (f numerator) (f denominator)) > 1.
phrase + f | f can reduce ↠: phrase + (f reduced).
f reduced ↠: reduce (f numerator) / (f denominator).
reduce fraction on n + phrase,
      ↠: n + reduce fraction on phrase.
phrase - f | f can reduce ↠: phrase - (f reduced).
phrase + f | f can reduce ↠: phrase + (f reduced).
```

Figure 41. Simplifying fractions.

```
Topic name is Numbers-ℤ.
ℤ ↷ z.
Negative definition by construction: - ℕ.
Negative ↷ neg.
ℤ ↷ integer.
ℤ ↷ z'.
Negative ↷ neg'.
Negative ↷ ℤ.
ℕ ↷ ℤ.

Section name is +.
neg + neg' ↠: - join (|neg| + |neg'|).
neg + n ↠ n - |neg|.
n + neg ↠ n - |neg|.

Section name is -.
n - neg ↠ n + |neg|.
0 - n ↠ - join n.
neg - neg' ↠ neg + |neg'|.
neg - n ↠: - join (|neg| + n).
m - n | m < n ↠: - join (n - m).
- neg ↠ |neg|.

Section name is ·.
neg · neg' ↠ |neg| · |neg'|.
n · neg ↠: - join (n · |neg|).
z * z' ↠ z · z'.
neg · n ↠ n · neg.

Section name is ÷.
neg / n ↠: - join (|neg| / n).
neg ÷ n ↠: negate (reduce fraction on (|neg| ÷ n)).
neg / neg' ↠ |neg| / |neg'|.
n / neg ↠: - join (n / |neg|).
neg ÷ neg' ↠ |neg| ÷ |neg'|.
n ÷ neg ↠: negate (n ÷ |neg|).

Section name is Construction.
- join neg ↠ |neg|.
- - n ↠: - (- join n).

Section name is Module.
|neg| ↠ neg without first character.
```

Figure 42. Integers