

Ω : The Power of Visual Simplicity

Benoît Sonntag¹ and Dominique Colnet²

¹ Université de Strasbourg, Strasbourg, France

² Université de Lorraine - LORIA, Nancy, France

Dominique.Colnet@loria.fr, Benoit.sonntag@lisaac.org

Abstract. We are currently developing an innovative and visually-driven programming language called Ω (Omega). Although the Ω code is stored in text files, these files are not intended for manual editing or traditional printing. Furthermore, parsing these files using a context-free grammar is not possible. The parsing of the code and the facilitation of user-friendly manual editing both necessitate a global knowledge of the code-base. Strictly speaking, code visualization is not an integral part of the Ω language; instead, this task is delegated to the editing tools. Thanks to the global knowledge of the code, the editing process becomes remarkably straightforward, with numerous automatic completion features that enhance usability. Ω leverages a visual-oriented approach to encompass all fundamental aspects of software engineering. It offers robust features, including safe static typing, design by contracts, rules for accessing slots, operator definitions, and more, all presented in an intuitively and visually comprehensible manner, eliminating the need for obscure syntax.

Keywords: Visual object-oriented, Graphical operators, No context free grammar, Visual programming, Software engineering, Design by contracts

1 Introduction

The thirty glorious years of computer languages from 1960 to 1990 gave rise to most of the major concepts of today's programming. This flourishing period has touched every major issue in computer programming. First of all, memory management, with the notion of execution stack and the appearance of the first garbage collector (**Lisp** [15, 16]). During the same period, different directions concerning code management have emerged, with the fusion of code and data (again **Lisp**), or on the contrary a more formal approach with functional languages (**ML** [19], **Caml** [4], **Ocaml** [13]). Then a more pragmatic approach, with a grouping of code and data present in object languages (**Simula 67** [5], **Smalltalk** [8]) and the notion of inheritance which reaches its apogee with the prototype model (**SELF** [25]) and its dynamic inheritance. In this context, the semantics of languages was the battle horse to the detriment of form, i.e. notation and syntax, which took a back seat.

For historical reasons that we can well understand, the almost systematic use of context-free grammars, of either kind *LL* or kind *LR*, has never been really

questioned. We can only observe the bogging down of textual syntaxes and the appearance of real religious wars concerning syntactic choices. We believe that traditional context-free grammar has reached an evolutionary dead end and does not benefit from technological advances. Today's computers or phones offers us much more sophisticated representation possibilities. Also, we see appearing little by little an approach richer of communication between the man and the machine than the simple keyboard / mouse. It is still difficult today to predict the impact and interest of the appearance of multiple sensors, tactile, visual, auditory, .. that we already have in our phones or computers. But, it seems obvious to us that there is a growing gap between the hardware and the programming environments we use.

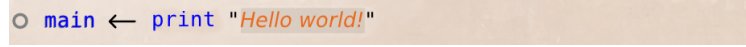
Furthermore, in a way too often decorrelated to expressiveness, for example for the optimization of *GPU*, compilers have evolved in terms of performance, but mostly for low-level languages. This almost immutable image in the collective mind of programmers, of the impossibility of combining performance and expressiveness must end. Let's note however some efforts to marry performance and high level language like the OCaml compiler [13], the SmartEiffel compiler [26], or more recently with Rust [11] and Lisaac [22, 23].

We aim to continue these efforts and to break with these various limitations and prejudices. Ω is intended to be a high-level language with a graphic visual while maintaining a rigorous terminology. Here we are talking about denotational terminology related to the field of application. The goal is to get rid of a rigid syntax to adopt the usual notations of a domain, as for example mathematics, physics, etc.

2 The Ω Language at a Glance Using ELIT Eyes

2.1 Visualized Code Blocks and Auto-Indentation

The most significant innovation in the Ω language is the concept of a visual approach that opens up a wider range of possibilities and notations. In contrast to its predecessor, the Lisaac language, a Ω program is not constructed as a text file bound by established and rigid syntax. Instead, a Ω program is represented as an editable figure using a specialized tool called, which allows you to visualize and edit the figures and pieces of text that comprise a program.



```
○ main ← print "Hello world!"
```

Fig. 1. The usual *Hello world!* program

Fig. 1 shows an image capture inside our development environment. This is the traditional *Hello World!* program in Ω . All the code visualization and, in particular, all the colors used for rendering are not part of the language definition. The choices for displaying and editing programs can be redefined and are entirely

up to the editing tool. Note that is an integrated development environment for Ω , itself written in Ω ¹.

Fig. 2 shows a variant of the previous program that uses two `print` instructions to perform the same display. The block that groups the two instructions is represented by an opening curly brace on the left. Without any explicit intervention on the part of the programmer, automatically selects the code representation. Its choice of representation in a given context is based on statistics carried out on existing code. In all cases, the display tool guarantees indentation and visualization in accordance with the code's semantics. When two instructions are displayed on the same line, may utilize a semicolon as a separator or employ other types of visual effects. Nothing is frozen by language rules.



```

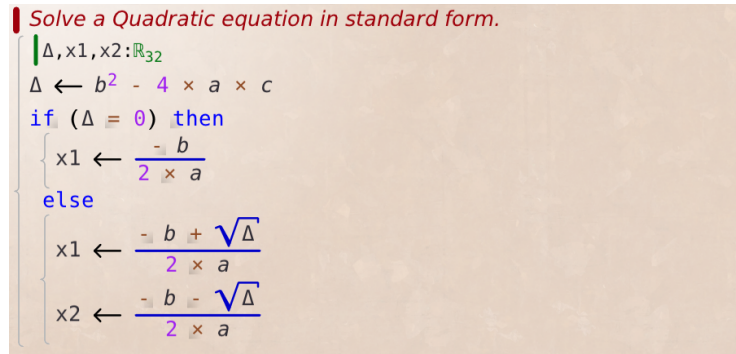
main ←
{
  print "Hello"
  print "world!"
}

```

Fig. 2. Two `print` statements within a multi-line block

2.2 Bringing Math and Computer Notations Together

Fig. 3 example starts with a comment, which can be identified by its red color, the default color of for displaying comments. The Ω language also makes it possible to add images, links and videos within comments. The enclosing block comprises three local variables Δ , $x1$ and $x2$, all of which are of type \mathbb{R}_{32} , a 32-bit floating-point number. Note that variable or type names can be composed of any unicode characters (in UTF-8 format), with the added possibility of subscript or superscript formatting. We've also resolved the debate over CamelCase *vs.* snake_case conventions by allowing spaces within any type of identifier.



```

Solve a Quadratic equation in standard form.
Δ, x1, x2: ℝ32
Δ ← b2 - 4 × a × c
if (Δ = 0) then
  { x1 ←  $\frac{-b}{2 \times a}$  }
else
  {
    x1 ←  $\frac{-b + \sqrt{\Delta}}{2 \times a}$ 
    x2 ←  $\frac{-b - \sqrt{\Delta}}{2 \times a}$ 
  }

```

Fig. 3. The default is to visualize **comments** in red and **types** using green

¹ As most video games, relies directly on the GPU thanks to the Ω library. The same Ω code runs on most platforms / OSes.

2.4 Extra Built-in Literal for Two Dimentional Notations

In addition to literals for strings or numeric constants, Ω also has literals for arrays, tuples of types or sequences of values. Thanks to its graphic and visual approach, Ω provides a natural way of visualizing a two-dimensional sequence of elements. We felt it was essential to add a literal for a pleasant representation of matrices (see Fig. 6). Unlike all the other graphic operators that can be defined in libraries, this two-dimensional literal is a built-in part of the language.

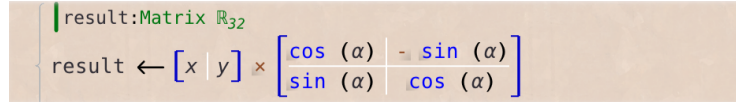


Fig. 6. The Ω language also incorporates a two-dimensional built-in literal

3 Signature for Slots

As previously mentioned, the language is statically typed, and all operations, whether they are textual or graphical, have a precise signature². Before we address how the signatures of graphical operators are specified, let's start with what is more common: textual signatures.

3.1 Textual Method or Attribute Signature

In the example shown in Fig. 7, we indicate both that the argument n of the `factorial` function is of type \mathbb{Z} and that the result type of this function is also \mathbb{Z} . The result returned by the `factorial` function is the expression that ends the function block. In this case, the result is the reading of the local variable `res`, also of type \mathbb{Z} . The `main` procedure shows an example of how to call the `factorial` function. Note that with the chosen signature for the definition of `factorial`, the use of parentheses is not required in the calling form.

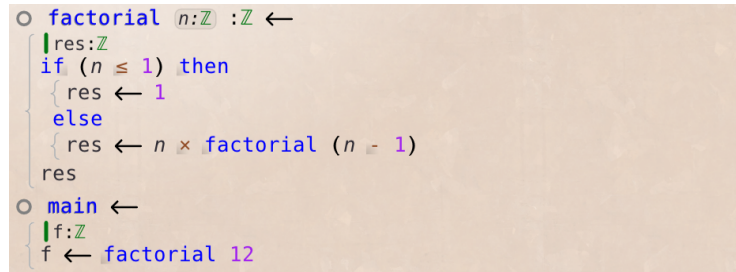


Fig. 7. Strong static and explicit typing everywhere

² As Ω is a prototype-based language, we prefer to use the term *signature* to provide information about the function parameters and return type.

If required, it is also possible to define a tuple of types in the case of a function with several results. In the example of the Fig. 8, the function `fiborec` returns a pair of values, each of type \mathbb{Z} . So the pair $(f0, f1)$ is a valid result for this function. The `fibonacci` function is implemented by a call to the recursive `fiborec` function. Note that this possibility of easily returning several results, without allocation in the heap, makes it possible here to obtain a recursive version in $O(n)$. This notion of tuple is generalized to all type declarations. In this way, a result tuple can be directly injected into an argument tuple.

```

○ fiborec n:ℤ : (ℤ, ℤ) ←
  | f0, f1:ℤ
  | if (n ≤ 1) then
  |   (f0, f1) ← (0, 1)
  | else
  |   (f0, f1) ← fiborec (n - 1)
  |   (f0, f1) ← (f1, f0 + f1)
  | (f0, f1)
○ fibonacci n:ℤ : ℤ ←
  | f1, f2:ℤ
  | (f1, f2) ← fiborec n
  | f2

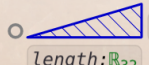

```

Fig. 8. The `fiborec` function return a pair of \mathbb{Z} integers

3.2 Graphical Operator Signature

The visual display for defining graphical operators is identical to that used when using them. Fig. 9 shows an example of the signature for a triangle-shaped graphical operator. This operator has two parameters, *height* and *length*, both of type \mathbb{R}_{32} . The result type of this operator is also \mathbb{R}_{32} . In the example, the surface area of the triangle is the result. The main procedure below provides an example of its usage.

```

○  height:ℝ32 : ℝ32 ← {  $\frac{length \times height}{2}$  }
  length:ℝ32
○ main ←
  | Ω:ℝ32
  | Ω ←  26.2
    64.414

```

Fig. 9. A graphical operator with 2 parameters

As we will see in more detail in Section 6.3, each graphical operator is identified by a textual signature. This signature serves both to facilitate code editing and to determine the order of evaluation. For example, regarding the square root, the corresponding function name is simply `sqrt`.

4 Object Oriented Programming is Back

Ω is a pure prototype-based programming language where every entity corresponds to a prototype. This rule applies to all objects, from the most complex to the most basic. For example, when it comes to numeric types, like signed integers, the Ω language allows you to use the \mathbb{Z} type to indicate a number of any size, with no limit other than the size of RAM memory. As Ω also aims to program any type of device with the best possible performance, it is also possible to explicitly use \mathbb{Z}_8 , \mathbb{Z}_{16} , \mathbb{Z}_{32} or \mathbb{Z}_{64} . One can indicate that the memory representation of some prototype must be directly mapped onto some machine word. Here, \mathbb{Z}_8 is mapped on a 8-bit signed integer, \mathbb{Z}_{16} is mapped on 16-bit signed integer, and so on. In case of direct mapping, there is no indirection, i.e. no pointer, and passing a \mathbb{Z}_{32} prototype is identical to passing an `int32_t` value in the C language. The implementation of the general type \mathbb{Z} relies on attributes used to store arrays of elementary values, enabling the manipulation of large values. Regardless of whether an object is elementary and mapped or defined by its attributes, it is necessary to be able to visualize or designate the receiver as well as its type.

4.1 Graphical Visualization of the Receiver and its Type

To simplify the presentation, in the preceding examples, we intentionally omitted specifying that the \circ disk indicates the receiver's position in the signature. Any instruction or expression is always within a prototype, and this prototype is always passed implicitly if necessary. Thus, there are no functions or procedures without a receiver. For example, in the code from Fig. 7, all calls to the `factorial` method pertain to the prototype in which this code is written. In this case, as in all the others seen previously, the called method only uses the data provided by the arguments. The current prototype is implicitly passed but not used.

Let's now explore the `at` method as defined in the `String` prototype and illustrated in Fig. 10. The purpose of this function is to access a character at a given index in the receiving string. Outside the context of the `String` prototype, it is necessary to specify which string of characters you want to work with. Therefore, the expression to access the character at index `i` in a character string `s` is simply written as: `s.at i`. The leftmost circle \circ indicates the position of the

```
 $\circ$  at index: $\mathbb{Z}$  :Character  $\leftarrow$  {storage item index}
```

Fig. 10. Definition of method `at` in `String`

receiver just before the `at` keyword within the signature. Please be aware that using a period in the calling form is not required. In this case, the calling form, `(s.at i)`, is identical to the notation used in `Smalltalk` or `SELF`.

Within the body of a method, the \bullet symbol is used to designate the receiver of the message. It's the equivalent of `self` in `Smalltalk` or `this` in `Java`. The

symbol \bigcirc , allows you to note the exact type of receiver using the idea presented in [3]. This choice ensures safe typing while at the same time avoiding the criticisms made to Eiffel on this subject [3]. Fig. 11 presents the signature of the `clone` method for which the exact type of the receiver is particularly well-suited.

```
○ clone:○ ← {[memcpy (malloc(@object_size),●)]`:○}
```

Fig. 11. The green circle indicates that the result matches the receiver type

4.2 Multiple Inheritance, Polymorphism and Visual Selectors

Inheritance in Ω is multiple, and each inheritance link indicates whether polymorphism is authorized or not. Furthermore, to facilitate the understanding of inheritance choices, one can decide whether to visualize inherited methods or not. Of course, in Ω , all these choices are made graphically. First example of Fig. 12 presents the inheritance tree of the `Boolean` prototype. A dotted line indicates implementation inheritance: all slots are inherited, but without permission for assignment. All descendant prototypes are also displayed.

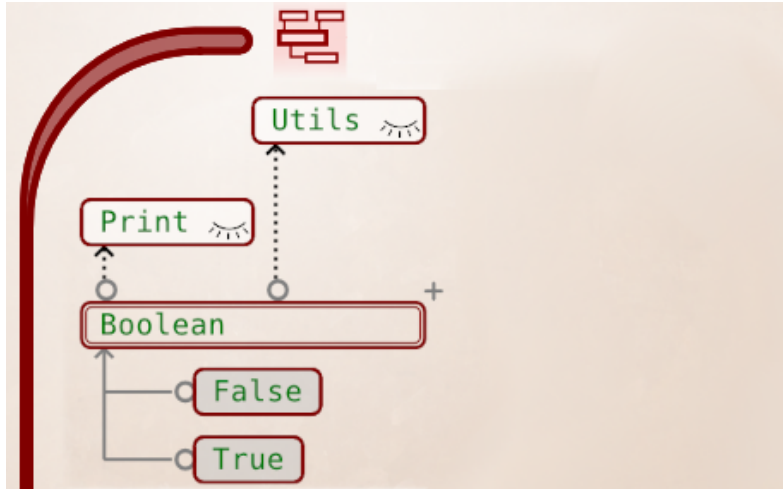


Fig. 12. No syntax for inheritance. The graphic is directly editable.

So in the example of `Boolean` we see that there are two descendant prototypes, `False` and `True`, each of which can be the object of an assignment in `Boolean`. Type `Boolean` inherits `Print` and then `Utils`, each time with a dotted line. These two prototypes, `Print` and `Utils`, are in fact simple method tanks [7]. Note that this inheritance visualization made by is also the one used to navigate or to modify inheritance. Once again, thanks to the visual approach, there's no need to learn any special syntax. Unlike a standard textual approach, we have a global representation of the inheritance tree, including the descendants.

Let's take a look at the **String** inheritance graph shown in Fig. 13. According to the selected inheritance order, **Hashable** before **Comparable**, the visualization shows the lookup algorithm's scanning order from bottom to top. Thus, if the lookup of some slot starts from **String**, the order of search is as follows: **Hashable** (1), **Clone** (2), **Print** (3), **Utils** (4), **Comparable** (5). If you need to modify the order of inheritance or the nature of an inheritance link (solid or dotted line), you can do so graphically via or by using keyboard shortcuts. It's essential that the editing tool is sufficiently easy and intuitive to use. To achieve this, we drew inspiration from video games as well as GPU capabilities.

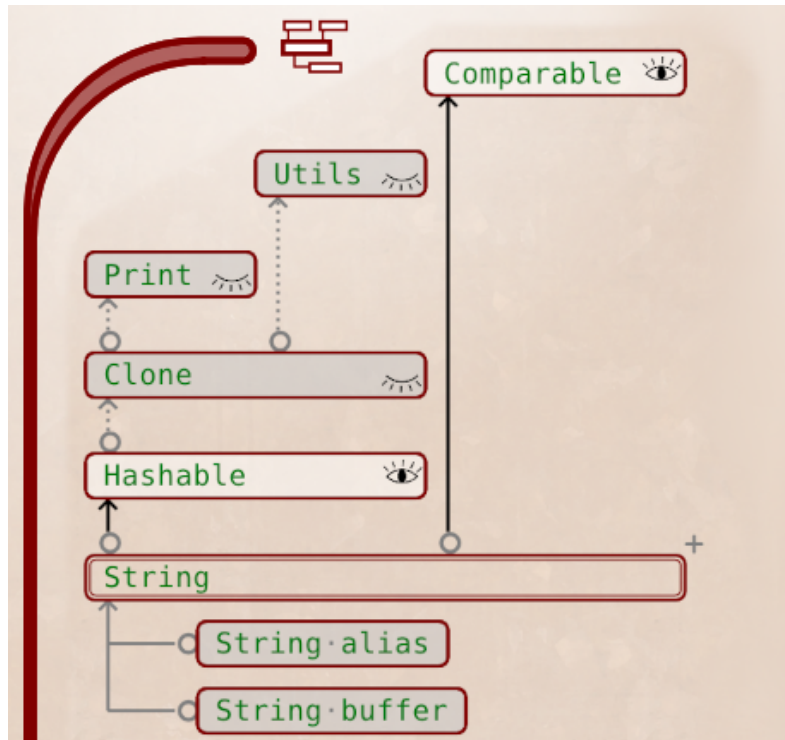


Fig. 13. Open eyes are used solely to assist users in browsing the code

In the Ω prototype library, there are two types of **String**: immutable and aliased strings of type **String alias**, and extensible and modifiable strings of type **String buffer**. Of course, all UTF-8 characters can be used with natural indexing. One of the problems of object-oriented programming is the lack of visibility of inherited slots when going through the code of a prototype. In Fig. 13, the open eye on **Hashable** and **Comparable** allows you to view their inherited slots directly within the **String** edition. A closed or open eye does not affect the semantic of the code. This is purely a visual artefact for the convenience of the code designer. Thus still on the example in Fig. 13, the eyes are close on **Clone** and **Print** because those

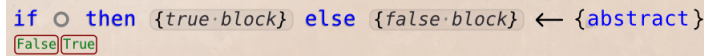
prototypes are very familiar. As we are well aware of the inherited slots from `Clone` and `Print`, we thereby prevent their visualization in the child prototypes.

To further discuss Ω inheritance, it's important to note that there is no default root for the inheritance tree. It is entirely possible to create a new prototype that inherits nothing, a practice often beneficial, especially when modeling singletons [7]. In order for a prototype to be cloneable, it must inherit, either directly or indirectly, from the `Clone` prototype, which contains the `clone` method – the sole method responsible for generating an object by duplicating an existing prototype. As `True` and `False` prototypes do not inherit from `Clone`, they remain unique objects.

4.3 More Flexibility to Place the Receiver in the Signature

As the Ω language is above all perfectly object-oriented, it's thanks to dynamic binding that conditional instructions are implemented. Fig. 14 shows an extract from the code of `Boolean`, which corresponds to the definition of the `if then else` construct used in the previous examples. The small round \circ in the signature indicates the receiver's position.

So the `if then else` method is a method whose first argument (also receiver) is of type `Boolean`. The second and third arguments after `then` and `else` are delayed evaluation blocks. Note that, compared to `Smalltalk` or `SELF`, the great novelty of Ω is that you can put a keyword *before* the receiver, but also after the last argument. Thanks to this flexibility and simplicity when it comes to place keywords, we avoid the many criticisms of `Smalltalk` notation. Finally, it's also possible to enclose the receiver in braces `{}` for delayed evaluation. In this way, methods with a delayed evaluation block as receiver are arranged according to their block return type context. No more out-of-context *Block* class as in `Smalltalk` or `SELF`.



```
if o then {true_block} else {false_block} ← {abstract}
False True
```

Fig. 14. The receiver between `if` and `then` mimics the traditional notation

Still in Fig. 14, the two small red rectangles at the bottom of the method are added by to indicate that there are two redefinitions of this method, one in `False` and one in `True`. By browsing these definitions, you will check that the evaluation of the `then` and `else` blocks respects the expected semantics. It is particularly easy for the compiler to detect that there are only two possibilities for dynamic binding with a variable of type `Boolean` and thus generate the optimal code as described in [21].

4.4 Properties for Operators

All operators are library-defined. Unlike `Smalltalk`, the Ω language lets you choose the priority, right-associativity or left-associativity and type commutativity for each operator. Fig. 15 displays the definition of the `&&` operator as it

is defined in the **Boolean** prototype. Thanks to the graphical approach, again, there's no need to invent any special syntax for defining operators. The signature looks like any other textual method, except for the red color which is used for operators. The right-hand side features three visual controls to select priority level, associativity, and type commutativity.

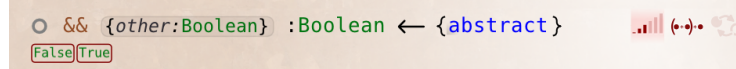


Fig. 15. Low priority level, left-associativity, and no commutativity

As another example, Fig. 16 presents the definition of the **+** operator. The type commutativity button is enabled. Furthermore the *other* argument is typed using **o** to indicate that we must have exactly the same type for the argument and the receiver.

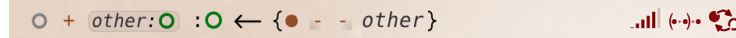


Fig. 16. High priority level, left-associativity and type commutativity

Returning to the definition of the **&&** operator in Fig. 15, it's worth noting that *other* is enclosed in curly braces. This indicates the presence of a code block. Its evaluation may be delayed and, as a result, depend on the exact nature of the receiver. In such a situation the call form must be perfectly compatible with the signature. So, only a call of the form $x \&\& \{y\}$ is valid. The aim is to fully visualize the signature choices at call site level. This helps avoid potential errors by highlighting the possible lazy evaluation. What's more, we've removed a special case that has no place in the elegance of a high-level visual language.

4.5 Visualizing Clonable Slots and the Self Symbol

As we have seen, the symbolism to indicate the position of the receiver in a signature is **o**. This choice also makes it possible to introduce the language to beginners without having to go too quickly into the details of object-oriented programming. At first glance, the **o** symbol might appear to be a simple routine separator (Fig. 1, 2, 7, 8). We have also seen that within the body of a method, the **●** symbol is used to designate the receiver and that its exact type is **o**. To go a step further, given that the context of the Ω language is prototype-based and strongly typed, there is one last important symbols we'll now introduce. In a slot signature, symbol **o** can be replaced by symbol **⓪** to indicate the slot's behavior in the case of cloning. Thus, the **⓪** symbol indicates that the corresponding slot, usually an attribute, is duplicated in the case of cloning.

Fig. 17 shows an example of defining a **Pixel** as being represented using two attributes **x** and **y**.

Note the use of the symbol **⓪** in front of **x** and **y**, indicating the duplication of these attributes in the case of cloning. Ω guarantees encapsulation by

accepting attribute assignments only within the prototype that holds them (or its descendants). In addition, when reading a slot from the outside, the code for reading an attribute or for a function call is identical. This respects the uniform reference principle [18].

The “ $x \text{ } px \text{ } y \text{ } py$ ” method assigns the two attributes x and y . These two attributes are assigned in a single instruction, and the method returns \bullet to enable a cascading call. Still in Fig. 17, next comes the definition of the `new` function designed to build a new copy of `Pixel`. Note here that the `new` keyword precedes the \circ receiver, to get the usual creation notation. As with the previous method, the result type is exactly that of the receiver. The definition block only contains a clone call on the receiver, which is implicit (\bullet is not required). The figure ends with the calculation of the `distance` and an example of its use in `main`.

```

①  $x:\mathbb{R}_{64}$ 
①  $y:\mathbb{R}_{64}$ 
○  $x \text{ } px:\mathbb{R}_{64} \text{ } y \text{ } py:\mathbb{R}_{64} : \circ \leftarrow$ 
  {
     $(x,y) \leftarrow (px,py)$ 
  }
   $\bullet$ 
new ○  $: \circ \leftarrow \{\text{clone}\}$ 
○ distance  $other:\text{Pixel} : \mathbb{R}_{64} \leftarrow$ 
  {
     $dx,dy:\mathbb{R}_{32}$ 
     $dx \leftarrow other \text{ } x - x$ 
     $dy \leftarrow other \text{ } y - y$ 
     $\sqrt{dx^2 + dy^2}$ 
  }
○ main  $\leftarrow$ 
  {
     $p:\text{Pixel}$ 
     $valx,dist:\mathbb{R}_{64}$ 
     $p \leftarrow \text{new Pixel } x \text{ } 2 \text{ } y \text{ } 3$ 
     $valx \leftarrow p \text{ } x$ 
     $dist \leftarrow p \text{ } \text{distance} \text{ } (\text{new Pixel } x \text{ } 4 \text{ } y \text{ } 5)$ 
  }
```

Fig. 17. The `Pixel` prototype example

4.6 Order of Evaluation for Graphical Operators

As the Ω language is purely object-oriented, the definition of the Σ graphical operator naturally finds its place in `Numeric` (Fig. 18).

The position of the receiver is below the Σ symbol, then the *up* argument takes its place above the Σ symbol. The parameter *a* is a block that returns a result of a generic type parameter `T`. Here as well, thanks to the visual effects, the generic parameters are effectively highlighted.

The body of the method is implemented with a loop method `while do` defined in `Boolean`. Fig. 19 shows an example of using the Σ operator to calculate

an arithmetic mean. The mathematical formula is as beautiful as on a school chalkboard.

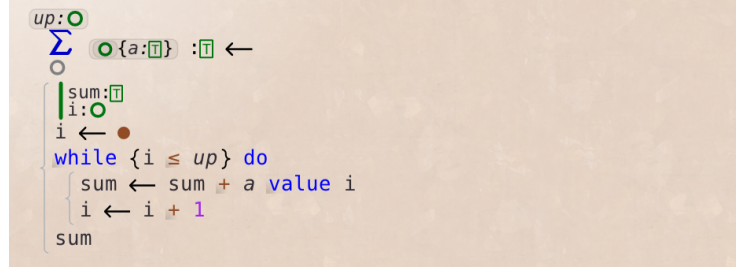


Fig. 18. Graphical Σ operator definition

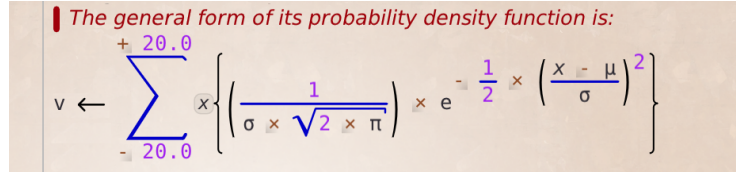


Fig. 19. Using the graphical symbol Σ defined in Fig. 18

Regarding the evaluation order of graphical operators, the \bigcirc receiver is always evaluated first, just as it is the case with textual forms. The evaluation order of the other arguments is arbitrarily determined at the same time as the definition of the vectorial shape of the operator. Since the same graphical operator is globally defined and can appear in different contexts, the goal is to uniformize the visually perceived semantics.

5 Visualizing Software Engineering Aspects

An obvious impact of the visualization layer concerns the comments placed within the code. There is no need to choose start and end markers, such as `/*` and `*/`, as is often the case. We won't provide additional examples with comments in this article because it's easy to imagine that selecting typography, shading, or boxing can distinguish comment sections from code sections. Nevertheless, you can also incorporate animations, short videos, and links to URLs within the comments. Additionally, you can refer to variables or types without the need to learn a separate sub-language, similar to Javadoc, for beautifying the comments.

5.1 Fine Control of Slot Access Rights Also Made Visual

The Ω language is designed for writing large projects and therefore requires a powerful mechanism for managing slot access permissions. Here again, a graphic

effect is used to select and visualize access permission for each slot. Fig. 20 repeats the example of the Fibonacci function, revealing the slot access rules with an enlarged view on the left. The green margin on the left, enclosing the `fibonacci` function, indicates that this slot is public. Conversely, the margin surrounding the `fiborec` function is dark, indicating that this slot can only be accessed by the \bullet receiver. It is also possible to authorize a list of prototypes by name, or to designate a directory that factorizes access for all the prototypes it contains. Again a visual artifact is used. No need for reserved keywords.

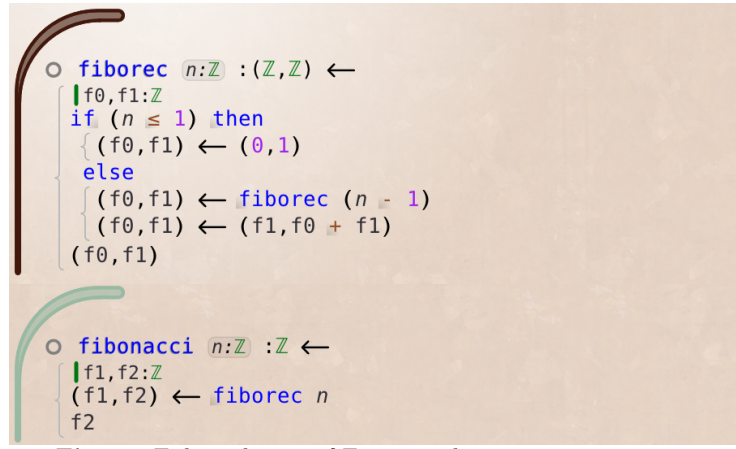


Fig. 20. Enlarged view of Fig. 8 to show access permissions

5.2 Programming by Contract on the Stage

The Ω language offers full support for design-by-contract programming.

Code zones corresponding to assertions are outlined in yellow hatching³. Here, the `at put` method is used to modify a collection by replacing a v value at a given i index. The precondition indicates that the index used must be valid within the index bounds before the call. The postcondition indicates that, after execution, the value has been written and that the number of elements in the collection, given by the `count` function, has not changed. When assertions are enabled, code is added to check that `count` gives the same result before and after the call.

Fig. 21 shows the abstract definition of the `at put` method in `Collection` [V]. The precondition and postcondition are inherited and visible in the various descendants. This direct visibility in descendants is a major advantage over other languages. It is also possible, in a descendant, to add new assertions or even to ignore an inherited assertion. Note also that allows quick navigation to the original location where an assertion is written. Even comments are subject to a

³ Not very visible here on paper but much clearer on a real screen.

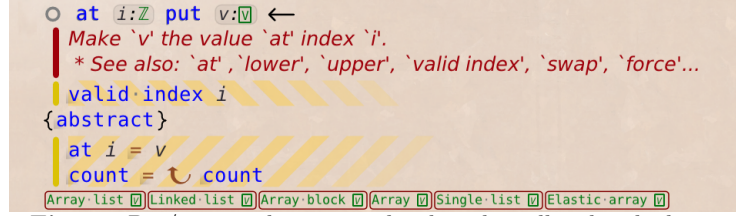


Fig. 21. Pre/postcondition visualized in the yellow hatched area

similar inheritance mechanism. Here again, the approach is visual and not, as in Eiffel [17], constrained by reserved keywords.

6 Storage Format, Parsing, and Editing

Each prototype is stored in a file with the same name in a format compatible with the UTF-8 encoding system. The entire prototype is described in this single file, but we will not delve into a detailed description of its format here. Indeed, some aspects, such as the format for comments or the format for storing all choices related to inheritance, pose no particular difficulty. We will focus on the storage format of slot signatures, both graphical and textual, and, of course, on the format for storing method bodies. It is in these parts that all the difficulties are concentrated. Regarding the format of method bodies, it is important, for example, to facilitate the use of copy and paste between different parts of the code.

The goal is to reduce the size of these files as much as possible, and give the code-editing tool maximum freedom for visualization. Our editing tool, itself written in Ω , uses a graphics library also written in Ω , enabling direct use of the GPU without dependence on an external library. So all our tools are cross-platform and operating system independent. By design, is a demonstration of Ω 's ability to handle low-level programming and large applications, while offering optimal performance without the need for C libraries to speed it up.

6.1 File Format and Compact Source Code Encoding

Being a programming language halfway between purely textual and graphical languages, the Ω language requires the encoding of many meta-information: blocks on one or more lines, indexing or exponentiation for a specific part of a name, multi-line or mono-line function calls, differentiation of types, local variables, and so on. Storing information on disk must adhere to both the semantics and the choices of presentation or formatting. In order to facilitate readability while avoiding a binary format incompatible with tools like *Git* or *Subversion*, we have chosen to use tags within the source files.

An XML-type markup was quickly discarded due to the obvious file size overhead. A lighter markup system in the style of LaTeX was attempted, but eventually, we settled on a markup using special UTF-8 characters. The UTF-8 encoding principle allows defining 2^{21} characters in its normalized form, limited

to 4 bytes⁴. Our idea is simply to reserve a small range of UTF-8 characters to define our tags in a minimal number of bytes. Within the two-byte UTF-8 character set, we chose the following range, $[0xCB80..0xCBBF]$, which represents 64 special characters reserved for Ω . These are compact characters, all occupying two bytes and having no direct usage corresponding to any human language. This lightweight approach is completely transparent to tools like *Git/Subversion*, which are capable of handling standard UTF-8 encoded files.

We will not explain here the meaning of each character within this range. It is worth noting that at present time, we utilize less than a quarter of the 64 reserved characters, thereby allowing space for future developments of Ω . Fig. 22 illustrates the encoding of some basic building blocks and style tags of Ω .

Symb	UTF-8 Range	Description
●	CB99	The Receiver <i>aka</i> 'Self'
○	CB91	Receiver Shared / signature
⊙	CB90	Receiver Cloned / signature
◯	CB9A	Exact Type of the Receiver
(←)→	CBB1+['0'..'9']	Left Associativity + Priority
→(←)	CBB2+['0'..'9']	Right Associativity + Priority
↻	CBA0	Commutativity Switch On
	CB84/CB86	Start/Stop Superscript
	CB85/CB87	Start/Stop Subscript
←	CBBF	Assignment Arrow

Fig. 22. The Ω language only reserves 64 UTF-8 codes

Initially, we opted for systematic tag usage to resolve all potential ambiguities within the code of a method. However, this approach proved to be too rigid and required challenging maintenance of tag consistency within the source file. In fact, the validation of this markup is no longer assured when updating from an external source to the current edition, such as through *Git/Subversion* updates, for example. Furthermore, it was particularly problematic to perform copy/paste operations between different code blocks.

For all these reasons, when it comes to method bodies, we have opted for a plain representation without any tags. This approach also facilitates the initial code typing during editing. Consequently, with each environment loading, such as when starts, the method bodies need to be reanalyzed. In other words, the use of formatting tags is only present in the prototype header, for a type, or when defining a slot signature. There is no markup within the method bodies.

It should be noted that in Ω , formatting an element as a subscript or superscript is not part of the namespace for identifying a slot or a type. For example, the type \mathbb{N}_{32} conflicts with a type defined as \mathbb{N}^{32} or $\mathbb{N}32$. In practice, this slight

⁴ Its principle could be extended up to eight bytes for a single code point, thereby increasing the count to 2^{42} characters. However, for compatibility reasons, we chose to ignore this potential extension.

limitation is particularly useful in avoiding names that are too similar and could lead to confusion. The same limitation is applied when it comes to method selectors.

In the definition of slot signatures, we distinguish between two types of slots: *Operator* and *Standard*. As the name suggests, an *Operator* slot should be writable with a single symbol or unique keyword, such as `+` or `&&` or `||`. An *Operator* slot can be unary, either pre- or post-fixed, or binary with associativity, priority, and potentially type commutativity. A slot in the *Standard* category can have any number of arguments and is left-associative with the highest priority. The type commutativity property applies, of course, only to slots in the *Operator* category.

Finally, each slot, whether in the *Standard* or *Operator* category, may be matched with a graphical slot. If such a correspondence exists, it is not mentioned in the method bodies. The corresponding textual representation should be used instead (detailed later in 6.3).

6.2 Parsing and Building of the Abstract Syntax Tree

During the design of our language, we set significant requirements regarding the flexibility of writing source code. These requirements led to significant challenges during syntax analysis. Aspects such as the ability to use spaces in identifiers, the absence of an explicit dot to denote dynamic binding, and the ability to add user-defined operators through libraries made it impossible to use an LALR-type parser.

To address the inherent ambiguities in the grammar, an upward parsing approach with dynamic programming CYK [9, 20] can effectively handle context-free ambiguous grammars. However, in the context of a strongly typed computer language, a purely context-free approach is suboptimal. It's worth noting that the complexity of this algorithm is $O(|m|^3 \cdot |G|)$, where $|m|$ is the length of the word to be analyzed, and $|G|$ is the size of the grammar. For more precise control and reduced computational cost, we have chosen an approach that utilizes semantic knowledge to make informed rule reductions. The parsing process occurs in two distinct phases. The first phase, aided by the markup system, is dedicated to acquiring semantic knowledge about the entire environment (see 6.2). It primarily focuses on prototype headers and slot signatures. Once this knowledge is acquired for all prototypes, a second phase of parsing the plain text of slot bodies, without markup, can begin (see 6.2).

First Step: Reading the Prototype Header and Slot Signatures. The first phase is quite straightforward and very fast as it involves reading a pre-positioned markup system in all source files. This allows us to locate and analyze all semantic information without ambiguity, facilitating the second parsing phase. The parsing of the marked portion enables the rapid collection of the following information:

Complete list of prototypes with or without generality. Formatting instructions for each prototype name, subscript or superscript for each prototype name.

Directed acyclic inheritance graph, nature of links, polymorphic or not, open eyes, etc.

Signature of all Standard Slots: sequence of identifiers and formatting, presence of exponents or subscripts, receiver position, argument types, and return value types if any.

Signature of Operator Slots: similar data collection as for Standard slots, with additional information on priority, unary pre or post-fix, or binary; priority, right or left associativity, and commutativity type if applicable.

For Operator Slots, a global dictionary is constructed. It will facilitate, in the second phase, the segmentation of a complex expression into multiple sub-expressions separated by operators. Regarding Standard Slots, we build a single syntax derivation tree for all the signatures in the environment (all libraries together with the current application). For example, for slots with identifiers such as `while`, `if then`, `if then else`, `add first`, `add last`, `add to`, and `append`, we construct the tree shown in Fig. 23. Thanks to this tree, we are now able to proceed to the second phase of parsing: the analysis of the plain text that makes up the slot bodies.

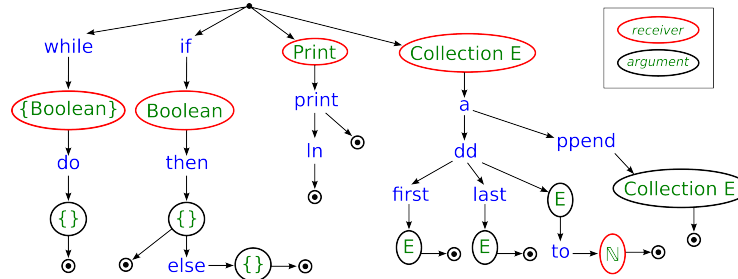


Fig. 23. Example of a syntax tree including global type information

Second Step: Parsing the Slot Bodies. The ambiguous grammar of a slot's body is presented in Fig. 24. The first four rules of the grammar identify assignments, declarations of local variables, blocks, constants, and tuples. Furthermore, Operator Slots are easily recognizable thanks to the dictionary built in the previous phase. This allows complex expressions to be divided into interleaved sub-expressions with invocations of Slot Operators. After analyzing each sub-expression, their static type precisely identifies the invoked Operator Slots. Building the evaluation tree, taking into account associativities and priorities, becomes possible. However, it is important to note that a semantic error at this

level of analysis is still possible. For instance, three interleaved sub-expressions of a left-associative operator and a right-associative operator, both with the same priority, make the construction indeterministic. In this case, the use of parentheses becomes essential to enforce the order of evaluation.

```

Body → {(idf_slot|idf_local) '←'} [op] Exp {op Exp} [op]
Exp  → Atom | Msg
Atom → ● | number | string | external | Type
      | '(' Grp ')' | '[' Loc ']' | '{' Grp '}'
Grp  → { Loc '\n' } [Body { (',' | '\n') Body }]
Loc  → { idf_local [ ':' Type ] ',' } idf_local ':' Type
Type → ○ | idf_type [ Prm { idf_type Prm } ] ⚠
Prm  → '(' Type { ',' Type } ')' | idf_type ⚠
Msg  → idf_slot [ Arg { idf_slot Arg } ] ⚠ | idf_local [ Arg ] ⚠
Arg  → Atom | idf_slot ⚠ | idf_local ⚠

```

Fig. 24. The ambiguous grammar of a slot body (EBNF representation)

The most challenging part of the analysis lies in the terminals *idf_local*, *idf_slot*, and *idf_type*. Each use of one of these terminals corresponds to a potential ambiguity, indicated in Fig. 24 by the ⚠ symbol. The three terminals that give rise to ambiguity are: *idf_local* for local variables, *idf_slot* for Standard Slots, and *idf_type* for types. It's important to note that in Ω , these three categories share the same namespace.

For parsing the content of an *idf_**, we use a CYK-type algorithm, utilizing the syntax tree built in the first step. The types of parameters help reduce complexity by making cuts in the tree of possibilities. Finally, among the different interpretations, we choose the longest recognized phrase.

In theory, it's easy to create a code example with two valid interpretations of the same size. However, in practice, thanks to the naming convention of types starting with an uppercase letter, in contrast to lowercase for slots, and by adhering to typing constraints, we haven't encountered any ambiguity in all the code currently written in Ω . However, if such ambiguity were to arise, the simple addition of parentheses would be sufficient to enforce the evaluation in one direction or another, thus eliminating the ambiguity.

6.3 Integration and Definition of Graphical Operators

In general, thanks to the knowledge of the formatting of types and slots gathered during the first phase (see 6.2), it is the responsibility of the editor to provide a correct visualization for each use of these elements. In the example of the type \mathbb{N}_{32} , the raw written version, $\mathbb{N}32$, must be displayed correctly, as \mathbb{N}_{32} .

All graphical slots (fraction bar, logic gate, square root, etc.) have a textual signature that identifies them. Each graphical symbol is defined in an external file that contains not only this textual signature but also instructions for the placement of arguments and vector drawing instructions (see Fig. 25).

```

A0 div A1                                // Textual corresponding form.
v0 <- SP * 0.46
v1 <- 2
Push                                     // Draw arg #A0 and push.
Push                                     // Draw arg #A1 and push.
v2 <- Max(A0.Width,A1.Width)
dx <- (A0.Width - A1.Width) / 2
dy <- A1.Ascent - A0.Descent + v1 * 2
Pop (dx, dy)                             // Pop and translate #A1.
dx <- (v2 - A0.Width) / 2 + 5
dy <- A0.Descent-v1-v0
Pop (dx, dy)                             // Pop and translate #A0.
Move (0x + 3, 0y - v0)                   // Trace the fraction line.
Line (0x + v2 + 7, 0y - v0)
Stroke 3
// Return new Ascent & new Descent.
Ascent <- A0.Ascent - A0.Descent + v1 + v0
Descent <- -A1.Ascent + A1.Descent - v1 + v0
Width <- v2 + 10                         // Return global Width.

```

Fig. 25. Vectorial operator description file for fraction

Without going into too much detail, we have a dictionary of graphical symbols with keys corresponding to the identifiers of a slot’s signature without a type. For example, the keys “**A0 sqrt**” and “**A0 div A1**” represent the symbols for the square root and fraction bar, respectively. The visual application of the graphical symbol within a program is simply the use of a slot with a profile corresponding to the key of the symbol in question. For example, the presence of the message in the source **42 sqrt** will be directly interpreted graphically as follows: $\sqrt{42}$. This solution, both simple and flexible, even allows for improving the graphical representation of old code without any modification to it. For editing, entering **1 div 2** is sufficient to obtain $\frac{1}{2}$.

6.4 Global Knowledge and Static Typing for Easy Editing

To provide the most enjoyable user experience, it is essential that the programmer can enter their code without worrying too much about formatting (e.g. subscripts, superscripts, graphics, and indentation). When entering source code, we avoid the use of complex keyboard commands or mouse-driven graphical interventions. It is crucial that the programmer feels comfortable and can type their code with, at the very least, the same ease as in a purely textual language.

It is essential to emphasize that Ω ’s choice of being a strongly statically typed language not only provides valuable parsing assistance but is also a powerful tool for code auto-completion. More than a constraint and somewhat paradoxically, explicit typing makes the code entry process easier and even shorter. It should be noted that there is a slight constraint on type names, which are the only ones

that can start with a capital letter. Thus, entering an existing type name always triggers auto-completion, and a type name is most often obtained in just a few keystrokes. Since all prototypes are preloaded by, the absence of auto-completion indicates an error while typing the code.

For auto-completing slot calls, the syntax derivation tree built for parsing is also utilized by. For example, at the current time, within all the prototypes, the only slot that starts with **if** corresponds to methods of **Boolean**. Simply typing **if** initiates auto-completion. Another example is if you enter **new**, you are presented with a list of types that have this method selector before the receiver. Finally, by entering the initial letters of **while**, auto-completion is unique and straightforward because the only possible current receiver is a block of type **Boolean**. Auto-completion is semantic and is based on global knowledge of prototypes and possible signatures.

Regarding the basic elements of Ω , they are entered using user-redefinable shortcuts. For example, for the assignment arrow \leftarrow , the default shortcut is ' \leftarrow ', but someone familiar with the **Eiffel** language might opt for ' $:=$ '. Many shortcuts are obvious and almost natural, such as the shortcut ' \leq ' for ' \leq ', for example. The default shortcut for \bullet is currently '**self**'. All shortcuts are redefinable, and they can even be associated with function keys or other specific devices. Users simply need to make their choice according to their preferences while avoiding conflicts.

Furthermore, the syntax tree of possible signatures (similar to the one shown in Fig. 23) is annotated with probabilities of single-line or multi-line indications. In this tree, each node corresponding to either a receiver or an argument has this information calculated based on the code encountered so far. We use these probabilities to represent blocks during their initial creation. For example, for the slot **while** { ... } **do** { ... }, the first block has a high probability of being on a single line, while the second block has a high probability of being multi-line. During auto-completion, we use this information to make the initial display choice. Of course, the programmer has complete freedom to correct this choice. His preference will then be stored as is and become part of the input statistics for future usage.

In the end, intuitive editing, acting directly on the visual representation of Ω code, is very straightforward for the user. This is made possible in part by the minimalist nature of the Ω language and also by the global knowledge of the entire existing code. What was achieved for the efficient compilation of the code presented in [21] also proves to be usable to facilitate editing. The benefits of abandoning the archaic approach of editing a single source file and separate compilation are clearly evident in an era where the capabilities of random-access memory easily support this comprehensive approach. We have not discussed memory consumption regarding the consideration of all existing prototypes which are all loaded because it has remained, so far, remarkably small.

7 Related Languages or Environments

In the history of programming languages, the first language that started using visual effects for programming is most certainly **Smalltalk** [8]. Indeed, already in 1980, all the indications concerning the inheritance are done via an interface with the mouse, using the famous browser of **Smalltalk**. No real syntax for the classes either and only the methods are defined using a text that has to respect a fixed syntax, but otherwise very pleasant.

In **Smalltalk** the definition of control structures and conditionals is already part of the library. Note that the operators are also redefinable, but they are all left associative with the same priority. Huge progress, **Smalltalk** invents the keyword notation and generalizes the concept of object for all manipulated data, even the simplest. Thus, Ω is in the direct line of Smalltalk's heirs, with a new, more flexible notation, statically typed and compiled language, which swaps the classes of **Smalltalk** for the prototypes of **SELF**.

In 1983, the **Prograph** language [1] [14] is completely graphic and avoids the use of text for programs. The underlying language is class-based and data flow driven. The most successful implementation is the one that works on Macintosh. The same implementation, apparently unchanged for a very long time, is still available today on MacOS X. The greatest fault of **Prograph** is probably to have been too far ahead of his time. Indeed, the cheap computers of that time were not powerful enough and, in any case, did not even have a mouse!

In the right line of the **Logo** language [6], itself non-graphical, **Scratch** [2] is probably one of the most successful visual programming environments. **Scratch** is a simple coding language with a visual interface that allows young children to create digital stories, games and animations. The name of the language, **Scratch**, is supposed to evoke the sound of DJs on their turntables. The craze for **Scratch** is palpable with many users and a wide diffusion as well as multiple translations of this language. Many videos on the web show its use and not only by children, showing a certain interest for a less austere and more graphic programming language.

Even if **Scratch** allows for example to program the equivalent of loops and conditionals, the language remains however rather limited. All graphical constructs are frozen and the language is essentially event driven and block assembly driven.

The *Snap!* language⁵ [10] is largely inspired by **Scratch**. The name is meant to evoke the noise that blocks of code make when they are assembled with each other. *Snap!* is a language that allows the definition of its own blocks, with the use of anonymous functions. The concept of *First class sprites* corresponds to that of the prototype and *Snap!* integrates all aspects of object-oriented programming. The purely functional style can also be used with the possibility of having higher order functions. Thus, *Snap!* is a significant advance in terms of a more general-

⁵ Do not confuse *Snap!*, note the use of the exclamation mark, with Snap which is a language from the 1960s and which is dedicated to the teaching of computer science to students of humanities.

purpose visual programming language, and not only dedicated to programming initiation for beginners or children.

Fig. 26 shows the *Snap!* program for the `fibonacci` function of Fig. 8 following a very similar approach. The definition is recursive but remains of linear complexity thanks to the use of a result in the form of a pair of values. Therefore, the ability to return multiple values at once is not merely a syntactic gimmick. Without going into details, this possibility is offered in Ω without ever having to allocate a container to group the different results. The *Snap!* language is very similar to Ω in that these two visual languages each provide full access to a real underlying programming language. In both cases, it is no longer possible to use an ordinary text editor to edit your code. A *Snap!* block name also uses a keyword system very similar to that used for a slot name in Ω . This being said, *Snap!*, with its Lego-like visuals, is resolutely oriented towards the discovery of programming. Ω retains a more traditional visual style that remains quite textual and integrates many aspects specific to software engineering (e.g. programming by contracts, exportation rules, static typing, etc.).

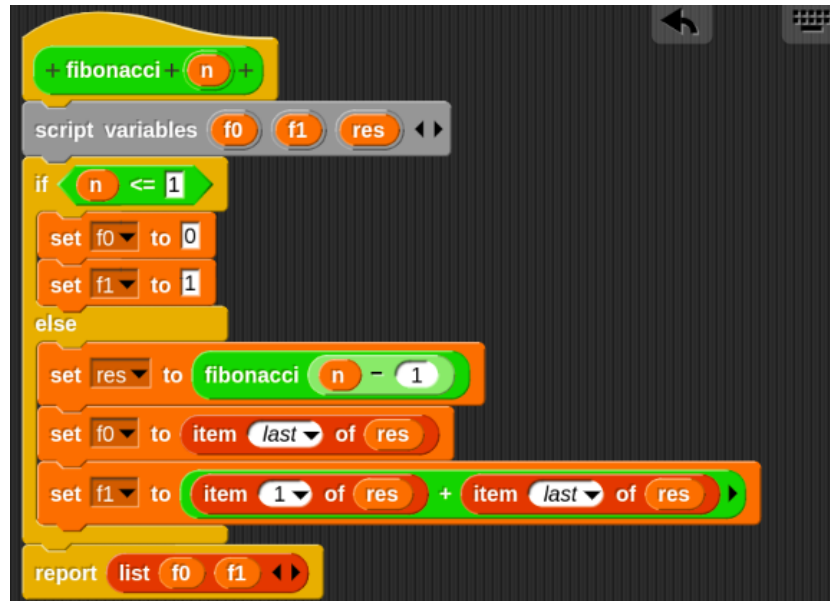


Fig. 26. The `fiborec` function of Fig. 8 in *Snap!*

For integrating visual aspects into the source code of programs, Jupyter Lab [12] is often cited as a reference. This approach allows users to incorporate graphics, diagrams, images, and other visual elements alongside the code, thus providing an interactive and immersive experience. However, within the scope of Ω , we aim to push the boundaries of this approach by introducing a new language that offers even deeper integration of visual elements. This seamless fusion between code and graphical elements represents a significant advancement in the communication

and presentation of source code, fostering deeper exploration and a more intuitive understanding of complex concepts or calculations.

8 Conclusion

We have presented Ω , a statically typed language designed to facilitate the development of large-scale software. In addition to ensuring safety, static typing also promotes achieving the best runtime performance [21]. Like **Eiffel** [18], Ω integrates all programming-by-contract tools into a fully object-oriented environment.

The major innovation of the Ω language is to eliminate syntactic constraints (LALR grammar and parser, reserved keywords, etc.), through the use of a visual approach. For example, key software engineering concepts such as multiple inheritance, access rights management, pre- and post-conditions are represented by non-fixed visual artifacts in the language. The visualizations presented in this article (from Fig. 1 to Fig. 21) come from our specialized editor for Ω , named, itself entirely developed in Ω .

In order to benefit from tools like *Git* or *Subversion*, the source code of programs written in Ω is stored in ordinary UTF-8 text files. Thanks to static typing and the global understanding of the entire source code, the Ω language parser can easily resolve ambiguities inherent in the language's flexibility. Furthermore, global knowledge of the source code also allows for very intuitive editing of the source code, minimizing keyboard typing as well as menu usage.

Graphical operators are not limited to mathematical operators alone, and it is quite easy to add new ones. The definition of operators is completely vectorial, and our visualizer/editor does not use any external libraries to directly access the GPU. All libraries used in the implementation of are written in Ω . Thus, is the first example of a large program written with Ω .

References

1. Prograph. Acadia University / Andescotia Software, 1983. Still available at andescotia.com.
2. Scratch. Scratch Foundation, 2003.
3. Dominique Colnet and Luigi Liquori. Match-O, a dialect of Eiffel with match-types. In IEEE, editor, *37th International Conference on Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings.*, pages 190 – 201, Sydney, Australia, November 2000.
4. Guy Cousineau. *Caml*. Université Paris-Diderot (Paris VII) et INRIA, 1985.
5. Ole-Johan Dahl and Kristen Nygaard. *Simula 67*. Norwegian Computing Center of Oslo, 1967.
6. Wally Feurzeig, Cynthia Solomon, and Seymour Papaert. *Logo*. In *Bolt Benarek and Newman*. Cambridge Massachusetts, 1967.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

8. Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983. ISBN 0201113716, 1366 pages.
9. Dick Grune. *Parsing Techniques: A Practical Guide (2nd ed.)*. Springer, New York, 2008.
10. Brian Harvey and Jens Mönig. Snap! Build Your Own Blocks. In *snap.berkeley.edu*, 2011.
11. Graydon Hoare. *Rust*. In *www.rust-lang.org*. Mozilla Research, 2006.
12. Jupyter Lab. <https://jupyter.org/>, 2014.
13. Xavier Leroy. *OCaml*. www.ocaml.org, 1990.
14. S. Matwin and T. Pietrzykowski. Prograph: A preliminary report. *Computer Languages*, 10(2):91–126, 1985.
15. John McCarthy. *Lisp*. Massachusetts Institute of Technology, 1958.
16. John McCarthy. Réursive-function of symbolic expression and their computation by machine, part i. Communications of the ACM (Association for Computing Machinery), 1960.
17. B. Meyer. *Eiffel, The Language*. Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-247925-7.
18. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
19. Robin Milner. *ML*. University of Edinburgh, 1970.
20. Itiroo Sakai. Syntax in universal translation. In *1961 International Conference on Machine Translation of Languages and Applied Language Analysis, Teddington, England*, volume II, pages 593–608, London, 1962. Her Majesty’s Stationery Office.
21. Benoit Sonntag and Dominique Colnet. Efficient compilation strategy for object-oriented languages under the closed-world assumption. *Software: Practice and Experience*, 44(5):565–592, 2013.
22. Benoit Sonntag and Dominique Colnet. Lisaac: the power of simplicity at work for operating system. In *In 40th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific’2002)*, pages 45–52. Australian Computer Society, 2002.
23. Benoit Sonntag, Dominique Colnet, and Olivier Zendra. Dynamic inheritance: a powerful mechanism for operating system design. In *5th ECOOP Workshop on Object-Oriented Programming and Operating Systems - ECOOP-OOOSWS’2002*, Lecture Notes in Computer Science, page 5 p, Malaga, Espagne, June 2002. Springer Verlag. Colloque avec actes et comité de lecture. internationale.
24. Till Tantau. The TikZ and PGF Packages. In <https://ctan.org/pkg/pgf>. Institut für Theoretische Informatik, Universität zu Lübeck, 2020.
25. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA ’87, page 227–242, New York, NY, USA, 1987. Association for Computing Machinery.
26. Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The smalleiffel compiler. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’97, page 125–141, New York, NY, USA, 1997. Association for Computing Machinery.