Functions in TEX and Elsewhere

Jean-Michel Hufflen

Abstract After a short survey about the notion of function in Mathematics and Computer Science, we look into the relationships between (LA)TEX's commands and features associated with functions: processing arguments, possible side effects, … Then we explore the relationships between the notion of types and the different kinds of objects handled by (LA)TEX's commands.

Sommario Dopo una breve panoramica sulla nozione di funzione in Matematica e in Scienza dell'Informazione, esaminiamo la relazione tra i comandi (LA)TEX e le proprietà associate alle funzioni: elaborazione degli argomenti, possibili effetti collaterali, ecc. Quindi esploriamo la relazione tra la nozione di tipo e le differenti specie di oggetti gestite dai comandi (LA)TEX.

1. Introduction

The starting idea for this present article originates from Gregorio (2020a), presented at last GUIT[1] meeting, an English version also existing as Gregorio (2020b). At the beginning of this article, Enrico Gregorio explains that within LATEX3's terminology, there is a clear separation between *variables* and *functions*: the former are used as *containers* whereas the latter perform some *action* when they are called. Let us recall that the first step towards LATEX3's implementation is the expl3 package, providing a new programming interface for LATEX. In particular, new syntactic conventions can be put into action between the two commands `\ExplSyntaxOn` and `\ExplSyntaxOff` (The LaTeX3 project, 2021). Let us go back to variables and functions within expl3, this separation only affects internal definitions, not user-level commands, as explained in Gregorio (2020b). In fact, this distinction between containers and action-performers already existed within TEX's commands, but was not obvious, since identical notations are used for both.

Hereafter we propose a short survey of this notion of function in Mathematics and Computer Science, then we explore the relationships between functions and definitions expressed using constructs of (LA)TEX's language, such as `\def` or `\newcommand`. Often our examples from more 'classic' programming originate from functional languages, where the notion of function is obviously central. Besides, many programming languages use *types* within function definitions, we also sketch this notion and its relation with TEX's commands. Reading this article only requires basic knowledge in Mathematics and Computer Science. (LA)TEX's commands mentioned throughout this article are described in Knuth (1986); Mittelbach *et al.* (2004).

1. ***G**ruppo **U**tilizzatori **I**taliani di TEX*.

2. What is a function?

2.1. In Mathematics

Let us assume that readers are familiar with the notion of *sets* in Mathematics[2]. A *binary relation* $\mathscr{R}$ between two sets S_0 and S_1 is a subset of the cartesian product $S_0 \times S_1$. If for all element x_0 of S_0:

(i) either there is no element of S_1 associated with x_0,

(ii) or only one element of S_1 is,

then $\mathscr{R}$ is a **function** from S_0 to S_1, S_0 (resp. S_1) being the *set of departure* (resp. *destination*) of $\mathscr{R}$. Formally, this definition can be expressed by:

$$\forall (x_0, x_1), (y_0, y_1) \in \mathscr{R}, x_0 = y_0 \Rightarrow x_1 = y_1$$

Let us notice that this definition is the most widespread, but is not universal. For example, Grätzer (1979) and Mac Lane (1971) consider only the (ii) condition, that is, *applications* according to the most widespread terminology in Mathematics. So we can notice that this notion of function is subject to variation.

A function of *several* arguments is a function whose domain is a cartesian product. A zero-argument function $f_0 : \longrightarrow S_1$ may be viewed as $f_0 : \{\varnothing\} \longrightarrow S_1$, as mentioned by Grätzer (1979)[3].

Let us recall that 'classic' Mathematics are based on sets in the sense that basic mathematical objects are introduced as sets. For example, an (x_0, x_1) couple is formally defined as the set $\{\{x_0\}, \{x_0, x_1\}\}$. With this definition, we can show that two couples are equal if and only if their elements are equal and in the same order[4]. As another example, a real number can be defined as the set of its minoring rational numbers. A different approach, more based on a notion of *computation process*, has been coined with the *λ-calculus* (Church, 1941). Without going thoroughly into the basic definitions of this formalism, let us mention that a function such as:

$$\begin{array}{rcl} f : \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \mapsto & x + 1 \end{array}$$

would be denoted by $f = \lambda x : \mathbb{R} \,.\, x + 1$ according to conventions closely related to *typed* λ-calculus, and $f = \lambda x \,.\, x + 1$ in 'simple' λ-calculus, without additional type information. This 'λ' notation, allowing a function to be wholly specified by an expression, has intensively been used by *functional programming languages*, such as Lisp dialects (McCarthy, 1960) or more modern languages such as Standard ML (Paulson, 1996) or Haskell (Peyton Jones, 2003). Such languages allow a function to be an argument or result of a subprogram. For example, we can easily program a sort procedure parameterised by the relation order used for elements. In addition, modern versions of general-purpose programming languages such

2. Readers interested in going thoroughly into this notion of *sets* can report to Halmos (1987), very didactic.
3. Let us remark that this definition has been *exactly* implemented in some programming languages—e.g., Haskell (Peyton Jones, 2003)—: such a f_0 function is of type $() \rightarrow S_1$, where '()' denotes the *unit* type, containing only the () value.
4. The proof is quite tedious but not really difficult.

as C++ (Stroustrup, 1991), Python (Lutz, 1996), and Java (Java, 1997) have included such functional expressions[5].

2.2. In Computer Science

2.2.1. Generalities

Calling a *function*, within programming languages used in Computer Science, is related to a *process*, that is, a fragment of a computer program *being executed*. More precisely, a function is a subprogram that *produces* a *result*, whereas other kinds of subprograms—procedures—just perform some effects. Let us notice that the C programming language (Kernighan and Ritchie, 1988) has reversed this view since a C program is a sequence of functions returning results, particular cases are functions that returns a result being the `void` type: they just return to the caller function, as procedures do within earlier programming languages such as Algol (Naur, 1960) or Pascal (Wirth, 1971). Functions used within Computer Science may be very different from mathematical functions, since they can *modify* their environment. As a very simple example, the following C function counts the number of its calls—the `nb_of_times` variable being defined at the top level—:

```
int nb_of_times = 0 ;

int plusone_plus(int x) {
  nb_of_times++ ; return x + 1 ;
}
```

Let us mention that functional programming languages aim to put into action functions in the mathematical sense of this word, without *side effects*—unlike *imperative languages* such as Algol, Pascal or C—, but in practice, most of functional programming languages allow assignments and physical change of structures, that is, *side effects*.

The previous example has a great drawback: since the `nb_of_times` variable is global, *any* subprogram can modify it with its own way. A better version, related to the same functionality, is given in Scheme by the following function, returning a linear list with the successor of a number, followed by the number of calls of this function. Within this version, the `nb-of-times` variable is *local*, installed at the *definition* of this `plusone-plus-s` function, and incremented at each call, then saved. Only this `plusone-plus-s` function can access to the `nb-of-times` variable, so this variable is *protected* against other access.

```
(define plusone-plus-s
  (let ((nb-of-times 0))
    (lambda (x)
      (set! nb-of-times
            (+ nb-of-times 1))
      (list (+ x 1) nb-of-times))))
```

For example, the first two calls of this function could be:

```
(plusone-plus-s 2021) ⟹ (2022 1)
(plusone-plus-s 82)   ⟹ (83 2)
```

5. A didactic introduction to these *functional expressions* used in Java 8 can be found in Liguori and Liguori (2014).

This example allows us to introduce the notion of *closure*. The `plusone-plus-s` function deals with a *local environment*, retaining the local variable `nb-of-times`. This local variable must not be cleaned by a garbage collector between successive applications of the `plusone-plus-p` function, and this function should be able to retrieve this variable whenever it is called, that is possible by means of a mechanism so-called *lexical closure*. Some languages do not provide closures, but modern ones—e.g., Python or Java—do it. Sometimes a lexical closure allows a local variable to be read, but not written: this is the case in Java; Python's modern versions defaults to this behaviour, even if additional declarations allow such local variables to be updated.

2.2.2. Functions vs other objects

On another point, the Scheme functional programming language allows us to emphasise a great difference, between *functions* and *special forms*, already sketched in Hufflen (2020). Let us recall that Scheme systematically uses prefixed forms, that is:

```
(* 16 10 (+ 1918 103))      ⟹ 323360
(if (= 2020 2021) 'ok 'ko) ⟹ ko
```

In the first example, a function—the product, denoted by `*`—is applied to numbers, which may be directly provided or be the result of a computation, e.g., `(+ 1918 103)` ⟹ `2021`. In such a case, Scheme begins with the evaluation of all the arguments of a function before applying it, that is, the strategy is based on *calls by value*[6]. In the second example, a condition is evaluated, and depending on this result, the second or third argument of the `if` special form is evaluated and gives the result of this special form. Let us remark that `if` cannot be a function in Scheme; otherwise, it would evaluate all its arguments, which would be catastrophic in an example such as:

```
(if (zero? j) #f (/ i j))
```

where '`#f`' denotes the *false* logical value on Scheme. We can intuit that this expression implements a protection against a division by zero. But if all the arguments of `if` were evaluated, the `(/ i j)` expression would be evaluated even if `j` was equal to zero. So, it is impossible, for a programming language, to express any construct by means of a function, if calls by value are used for functions. More generally, this notion of special form applies to the constructs predefined in a programming language, such as conditional or iterative constructs, etc.[7] Some languages—including Scheme— allows the definition of *macros*, in which case the arguments of a macro are taken *verbatim*, without evaluation, as mentioned in **?**h2020:

```
(define-syntax twice-m
  (syntax-rules ()
    ((twice-m x)
```

6. Other strategies, possibly based on variables' *addresses*, were used within the first programming languages, or are still in use nowadays, especially within object-oriented languages. This point is outside this article's scope.
7. The difference between functions and other constructs exists in languages other than Scheme, but within 'traditional' languages such as C, the latter are expressed by syntactic markers—e.g., '`if (. . . ) . . . ; else . . . ;`'—so the difference may appear more clearly. On the contrary, the same syntax is used for all commands provided by TEX, as in Lisp dialects (including Scheme).

```
    (list (quote x) (quote x)))))

(twice-m (+ 2020 1)) ⟹
       ((+ 2020 1) (+ 2020 1))
```

2.2.3. Call by value or by need

Several recent functional programming languages —including Haskell—no longer use a call-by-value strategy, but *lazy evaluation* (or *call by need*). An argument of a function is evaluated only if need be, and is evaluated once in such a case. The main interest of this *modus operandi* is that it avoids the evaluation of a program that loops if its result is not needed. Let us consider these two examples in C:

```
int endlessly() {
  back : goto back ; (* Loops endlessly! *)
  return 2020 ;      (* Never reached. *)
}

int thisyearf(int x) { return 2021 ; }
```

We can observe that the result returned by the `thisyearf` function is independent of its argument's value, so evaluating it is not needed *stricto sensu*. However, the evaluation of the expression:

```
thisyearf(endlessly())
```

loops: C's call-by-value strategy causes the subexpression `endlessly()` to be evaluated… without returning any value. Now let us look at an analagous definition using Haskell:

```
thisyearf x = 2021
```

The expression `thisyearf (1 / 0)` returns `2021` and the division by zero has not been performed since the argument has not been evaluated. We do not want the reading of this article to be slowed down, so some complements about the lazy evaluation have been put at Appendix A. Hereafter we just mention that:

- the lazy evaluation allows the definition of infinite objects: if only a subpart is of interest, the rest can remain untouched (cf. Appendix A);
- according to such a strategy, a conditional construct '`if ... then ... else ...`' can be viewed as a function: the first argument is evaluated, and according to the result, either the second or the third is evaluated.

2.3. Functions and (LA)TEX

In practice, we can consider that the (LA)TEX's commands introduced by constructs such as `\def` in *Plain TEX* or `\newcommand` in LATEX are functions returning strings when expanded fully. Other data types are used within TEX—e.g., counters, dimensions, etc.—but they are defined by other constructs. Some commands may return an empty string, in particular the commands that perform *side effects*, e.g.:

```
\def\skipfootnote{%
 \addtocounter{footnote}{1}%
}
```

which skips a footnote number. This behaviour includes commands that performs new definitions on the fly, e.g.:

```
\def\title#1{\gdef\@title{#1}}
```

After running this `title` command, the `\@title` command[8] will be a container for the document's title. (LA)TEX's commands without argument introduced by `\def` or `\newcommand`—e.g., `\skipfootnote`—are closer to zero-argument functions than constants ('single' elements of a set), since they are evaluated as many times as they are called. They are closer to *processes* than *associations* of variables with values.

There exists some significant similarity between TEX's language and functional programming languages: the latter allows functions to build new functions, the former allows commands to run commands dynamically built. If the generated command's name is the result of some computation, the construct '`\csname`…`\endcsname`' allows this name to be specified.

As mentioned in Hufflen (2020), TEX is a *dynamic* language. There is no way to use lexical closures. Let us consider this example, close to what we proposed in our previous article—let us recall that if `C` is a counter, '`\theC`' yields its value—:

```
\newcounter{lastyearc}
\setcounter{lastyearc}{2020}
\edef\firstsentence{%
 We ain't in \thelastyearc, are we?\par}
\def\secondsentence{%
 We are in \thelastyearc, ain't we?\par}
\addtocounter{lastyearc}{1}
```

The `\secondsentence` command yields the paragraph:

We are in 2021, ain't we?

whereas the `\firstsentence` command's body is expanded as far as possible at definition-time:

We ain't in 2020, are we?

So the `\edef` command allows a lexical scope to be simulated—as we show in Hufflen (2020)—and prevents against a command's redefinition. But the *updates* of a single command are ignored, too. In addition, if an (LA)TEX command uses a local environment, changes on it are not saved when this command exits. In TEX, persistent updatings can be applied only on global definitions, as we do in our first example `plusone_plus` in `C` (cf. § 2.2.1).

As mentioned in Hufflen (2020), TEX's commands are closer to macros than functions since the arguments are taken *verbatim* and processed when the command's body is expanded. Such a *modus operandi* allows a command to be deferred until it is applied. Let us consider the following definition:

8. Let us recall that this `\@title` command is both internal—its name contains the '`@`' character—and global—it has been introduced by the `\gdef` construct, which is a shorthand for '`\global\def`'.

```
\def\apply#1#2{#1{#2}}
```

If the first argument of this `\apply` command is also a command, it is applied to the second argument: '`\apply{\uppercase}{ok}`' returns 'OK'. Applying a command by giving its name as a string, is possible, too:

```
\def\applywrtname#1#2{%
 \csname#1\endcsname{#2}%
}
```

in which case an example could be:

```
\applywrtname{uppercase}{guit}
```

which produces 'GUIT'. As in a macro, TEX command's arguments are evaluated as many times as they occur. Some workarounds allow multiple evaluations to be avoided, and a kind of call by value can sometimes be simulated by using the `\expandafter` command (Knuth, 1986, p. 213), which expands the first token after the second. A 'classic' example is:

```
\uppercase\expandafter{\romannumeral 753}
```

where the `\romannumeral` command is expanded before the `\uppercase` command is applied, so this last command correctly processes Roman numerals and the result is 'DCCLIII', as expected. If the `\expandafter` is removed, then the `\uppercase` command processes the group between braces *verbatim*—which returns this group unchanged—and the `\romannumeral` command is applied, so the result is 'dccliii'.

The use of a *box* as a container (Knuth, 1986, pp. 120–122) can allow multiple evaluations of the same argument to be avoided, as shown in Hufflen (2020):

```
\newbox\tmpbox
\def\twicemversiontwo#1{%
 \setbox\tmpbox\hbox{#1}%
 [\unhcopy\tmpbox,\unhbox\tmpbox]%
}
```

It is more related to a call by need than a call by value. A command's arguments are not evaluated and the *decision* of making a box is taken inside the command's body, in which case an argument may be evaluated only once. In other words, we are very close to a lazy evaluation.

2.4. In LATEX3

As abovementioned, the first step towards the implementation of LATEX3 has consisted in designing a new programming interface for LATEX by means of the expl3 package, documented in The LaTeX3 project (2021). Let us go back to our purpose, this package provides interesting and useful *abstraction barriers* between user-level commands and auxiliary definitions. However, since *all* the definitions are expanded into 'basic' TEX's language, this expl3 package cannot add features impossible to put into action in 'basic' TEX. So LATEX3 remains a dynamic language—as TEX—even if it possible to force expansions at definition-time. The scope of variables is clearly specified, but there is no local remanent variables which would allow closures.

Concerning the management of functions' arguments, the expl3 package proposes *type specifiers* that allow expansion to be controlled. As in TEX, this system is more related to macros than functions, since the expansion level can be specified: one-level or recursively[9]; in other words, the expansion can be *limited*. As an interesting innovation, specifying calls by value is easier, since these specifiers can apply to any argument. In §2.3, we showed that the `\expandafter` command can simulate it, but in practice this strategy is suitable for a command's first argument. For the others, we may have to put an impressive number of occurrences of this command. From our point of view, the finer control of expanding arguments is actually a great contribution of LATEX3.

3. Types

3.1. In Computer Science

Within this short survey, we do not go thoroughly into theoretical aspects about type theory[10]; according to a basic approach, we only consider a type as a characterisation of possible values for expressions used throughout programs. Regarding how types are handled, programming languages can be divided into three categories:

typed any value belongs to a type;

strongly typed in addition, variables are given a type;

untyped all the possible values belong to one type.

These definitions are the most commonly used, although some variations may be observed according to literature sources. In addition, we make precise that:

- in some languages, a *type inference* mechanism allows end users to be discharged from mentioning types; for example, Haskell acknowledges the definition of a function by making precise the arguments' types and result's; nevertheless, types are omnipresent within such languages: if the type inference fails, the corresponding definition or expression is rejected[11];
- some languages—including Scheme—manage types *dynamically*: functions allow users to be informed about any value's type;
- *types* and type *errors* should not be confused: for example, some people consider that Scheme is untyped because it is typed dynamically, but some type errors may occur—e.g., if an arithmetic operation is applied to non-numeric values—; as another example, PL/1 (IBM System 360, 1968) used type declarations, but a rich collection of conversion functions allowed a subprogram to be applied to any value (!), so type clashes were *impossible* within this language.

9. … which is close to calls by value.
10. Interested readers can consult Collins (2012), recent and very well documented about historical aspects of this part of Theoretical Computer Science.
11. For example, if the `thisyearf` function—cf. § 2.2.3—is processed by Standard ML, it acknowledges this function by giving its type: '`'a -> int`', where '`'a`' stands for 'any type, denoted by `a`'.

3.2. Types and (LA)TEX

Obviously, TEX's kernel is not a strongly typed language. Defining new types is not allowed, either. Is TEX an untyped language? We do not think so. In addition to characters and strings, it handles numbers, counters, dimensions, registers, and tables. A command crashes if it is applied to an object being a wrong type.

Is TEX a language dynamically typed? Sometimes, but not always. There is no test functions as in Scheme—`number?`, `string?`, etc.—but in some cases, a workaround may allow us to guess an object's type. Any TEXpert knows how to check if a `\c` command exists:

```
\expandafter\ifx\csname c\endcsname\relax%
  … % If the \c command is undefined, define it
    % as the \relax command and expand
    % this first part.
  \else… % Expand this second part if the \c
         % command is already defined.
\fi
```

by using the construct '`\csname`…`\endcsname`', mentioned at § 2.3. This *modus operandi* has been abbreviated in LATEX by:

```
\@ifundefined\c{. . . }{. . . }
```

and improved in e-TEX by the constructs:

```
\ifcsname c\endcsname… % If \c is defined.
  \else… % If not, it remains undefined.
\fi
```

or `\ifdefined\c{. . . }{. . . }`, equivalent to the previous expression.

Let us now go back to something close to type-checking and as an example, let us assume that we are wondering if a LATEX command can be applied to a counter defined by means of the `\newcounter` command. Let `ct` be the first argument: if it is actually such a counter, the `\thect` command allows its value to be displayed, so we can check if this command exists. Let us notice that such a test is a kind of *false-biased Monte Carlo algorithm*[12]: it is very quick; if it returns *false*, we are sure that the `ct` counter does not exist; if it returns *true*, the `\thect` command may exist without connection with a `ct` counter, even if such a case rarely occurs in practice. Besides, this *modus operandi* works with LATEX's counters, not with TEX's counters introduced by the `\newcount` command.

In fact, checking non-string objects in (LA)TEX could be easier if TEX's kernel included an error-handling mechanism, because all the non-string objects handled by TEX can be accessed by the `\the` command, which returns a string representation of such an object. But the 'trick' we have recalled applies only to a command name, we cannot check the validity of a complete expression.

12. In Computing, a *Monte Carlo algorithm* is an algorithm—often randomised, but always quick—whose output may be incorrect with a certain probability, quite small in practice (Metropolis, 1987).

3.3. In LaTeX3

Obviously the designers of LaTeX3 have wanted to put into action a *strongly typed* language. As explained in Gregorio (2020b), this decision does not apply to user-level commands, but to the implementation of such commands. On another point, it is preferable for LaTeX3's syntax to be close to LaTeX's. So the types of variables are not given as annotations—like in C—but belongs to the names of variables. For example, `\g_example_title_tl`, where '`tl`' stands for 'token list'. More generally, the format for a variable name is:

$$\backslash\langle \textit{scope}\rangle\_\langle \textit{prefix}\rangle\_\langle \textit{proper name}\rangle\_\langle \textit{type}\rangle$$

where:

- ⟨*scope*⟩ is '`c`' for a constant, '`g`' for a global variable, '`l`' for a local one;
- ⟨*prefix*⟩ is a package name;
- ⟨*proper name*⟩ is the 'actual' name of the variable;
- ⟨*type*⟩ denotes its type.

Function names are similar:

$$\backslash\langle \textit{prefix}\rangle\_\langle \textit{proper name}\rangle\_\langle \textit{signature}\rangle$$

where ⟨*signature*⟩ is a sequence of characters denoting the type of the function's successive arguments. For example, the arguments of the `\seq_seq_slit:Nnn` function are respectively of types `N`, `n`, and `n`; that is, a single token and two braced lists of tokens. The type of a function's result is not given within its signature.

Type specifications are different for variables and functions. A variable can retain any object of TeX, that is why types such as '`int`'—for integers—or '`box`' are allowed. The letters used within function signatures are related either to the look of the corresponding argument—e.g., '`c`' is for a braced argument—or its evaluation. As mentioned in §2.4, the expansion of an argument can be controlled by using:

- '`e`' for consuming an expansion's result;
- '`f`' for a recursive expansion, ending as soon as an unexpandable token is found;
- '`o`' for a one-level expansion;
- '`x`' for a compile-time expansion, as performed by TeX's `\edef` construct.

The expl3 package provides a rich collection of predefined types. We personally regret that there is no way to define new type of containers in order to retain more structured information. But maybe it is planned for future versions… The information provided by signatures is very precise, provided that the expansion mechanism of TeX is mastered by programmers. Checking the type of a result is both easy and tedious: there is no functions checking types, we have to retain a result into a variable whose type is given.

4. Conclusion

The purpose of this article is twofold. The first point is that there are some variations about terms frequently used within programming. The list of the programming languages we have cited throughout this article is obviously non-limitative, it shows how different the implementations of comparable notions are. As a consequence, the writers of a documentation should be very careful and precise about terminology. The second point, started in Hufflen (2020), consists of studying the programming in TEX, since its *modus operandi* is quite apart from the other programming paradigms. We have showed that about the notions of functions and types. We can be told that this language is very specialised and has been recognised very suitable for typesetting texts (!), but the need for functionalities more related to 'classic' programming has led to the coexistence of TEX and another programming language, the best example being LuaTEX (Hagen, 2006). In addition, some features very specific to (LA)TEX may be difficult to understand, so it may be useful to compare them with analogous functionalities within other languages. That is true about TEX's kernel language, but will probably be true for LATEX3's future language. The mechanisms put into action with this project are interesting and promising, especially if we consider the clear separation between the interface of commands and their implementation. But undoubtedly the behaviour put into action within LATEX3 will be very different to the other programming paradigms, too.

A. Lazy evaluation

As mentioned in §2.2.3, if a lazy-evaluation strategy is used, an argument of a function is evaluated only if need be. For example, let us consider the following definition:

```
succ2nd x y = y + 1
```

Evaluating '`succ2nd (1 / 0) 2020`' yields 2021 and the first argument of the `succ2nd` function—the expression '`(1 / 0)`'—has been untouched, whereas '`succ2nd 2020 (1 / 0)`' fails because the second argument needs to be evaluated in order to perform the addition.

We also mentioned that this approach allows the specification of *infinite* objects. Hereafter, we show the easiest way to see that:

- a structure may be evaluated only partially;
- a part of such a structure, if it is evaluated, is evaluated once.

A very simple example of an infinite object in Haskell is:

```
naturalnumberlist =
  let from n = n : from (n + 1)
  in from 0
```

The internal function `from` returns a list of all the natural numbers from `n`—where $n \in \mathbb{N}$—that is, `n` followed[13] by the list of all the natural numbers from `n + 1`. Obviously, the lists returned by the internal function `from`—including the `naturalnumberlist` variable's value—are infinite objects. Let us use the GHCi[14] compiler of Haskell and its `:sprint` tool, typing the

13. In Haskell, the '`:`' infix operator separates the first element of a list and the following ones.
14. **G**lasgow **H**askell **C**ompiler **I**nteractive. A didactic introduction to it can be found in O'Sullivan *et al.* (2010).

command '`:sprint naturalnumberlist`' causes the following output to be displayed:

```
naturalnumberlist = _
```

where '_' means that the expression has not been evaluated yet. Now let us access the elements Nos. 0 and 2 of this list[15]:

```
naturalnumberlist !! 0 ⟹ 0
naturalnumberlist !! 2 ⟹ 2
```

After these two evaluations, let us type again the GHCi command—'`:sprint naturalnumberlist`'—and the result is:

```
naturalnumberlist = 0 : 1 : 2 : _
```

When we access to a particular element—or more generally a finite part—of this list, we need to compute all the elements before it, but the elements located after it can remain unevaluated: that is expressed by the '_' notation for the part of the list that is not evaluated yet. Some elements could be evaluated in the future if we move forward in this list. Of course, evaluating `naturalnumberlist` itself would cause an infinite loop to occur.

Acknowledgements

I am very grateful to the anonymous referees of this article's first version, since I was given constructive comments and an Italian translation of my abstract. I also thank Denis Bitouzé for his valuable comments.

References

Church, Alonzo (1941). *The Calculi of Lambda-Conversion*. Princeton University Press.

Collins, Jordan E. (2012). *A History of the Theory of Types: Developments After the Second Edition of 'Principia Mathematica'*. Lambert Academic Publishing.

Grätzer, George (1979). *Universal Algebra*. Springer-Verlag, 2nd edition.

Gregorio, Enrico (2020a). «Funzioni e expl3». *ArsTEXnica*, **30**, pp. 38–50. In Proc. GUIT 2020 meeting.

— (2020b). «Functions and expl3». *TUGBoat*, **41** (3), pp. 299–307.

Hagen, Hans (2006). «LuaTEX: Howling to the moon». *Biuletyn Polskiej Grupy Użytkowników Systemu TEX*, **23**, pp. 63–68.

Halmos, Paul Richard (1987). *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag.

Hufflen, Jean-Michel (2020). «Which success for TEX as an old program?» *ArsTEXnica*, **30**, pp. 24–30. In Proc. GUIT 2020 meeting.

15. In Haskell, the positions inside a list are numbered from zero, as in C.

IBM System 360 (1968). *PL/1 Reference Manual*.

Java (1997). *The Source for Java™ Technology*. Documentation available at: http://java.sun.com.

Kernighan, Brian W. and Dennis M. Ritchie (1988). *The C Programming Language*. Prentice Hall, 2nd edition.

Knuth, Donald Ervin (1986). *Computers & Typesetting. Vol. A: The TEXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts.

The LaTeX3 project (2021). *The LATEX3 Interfaces*. http://ctan.math.illinois.edu/macros/latex/contrib/l3kernel/interface3.pdf.

Liguori, Robert and Patricia Liguori (2014). *Java 8 Pocket Guide. Instant Help for Java Programmers*. O'Reilly.

Lutz, Mark (1996). *Programming Python*. O'Reilly & Associates.

Mac Lane, Saunders (1971). *Categories for the Working Mathematician*. Numero 5 in Graduate Texts in Mathematics. Springer-Verlag.

McCarthy, John (1960). «Recursive functions of symbolic expressions and their computation by machine, part I». *Communications of the ACM*, **3** (4), pp. 184–195.

Metropolis, N. (1987). «The beginning of the Monte Carlo method». *Los Alamos Science*, **15**, p. 125–130.

Mittelbach, Frank and Michel Goossens, with Johannes Braams, David Carlisle, Chris A. Rowley, Christine Detig and Joachim Schrod (2004). *The LATEX Companion*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition.

Naur, Peter (1960). «Report on the algorithmic language Algol 60». *Communications of the ACM*, **3** (5), pp. 299–314.

O'Sullivan, Bryan, John Goerzen and Don Stewart (2010). *Real World Haskell*. O'Reilly.

Paulson, Lawrence C. (1996). *ML for the Working Programmer*. Cambridge University Press, 2ª edizione.

Peyton Jones, Simon (ed.) (2003). *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press.

Stroustrup, Bjarne (1991). *The C++ Programming Language*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition.

Wirth, Niklaus (1971). «The programming language Pascal». *Acta Informatica*, **1** (1), pp. 35–63.

Jean-Michel Hufflen
FEMTO-ST (UMR CNRS 6174) & University of Bourgogne Franche-Comté,
16, route de Gray,
25030 BESANÇON CEDEX
FRANCE
jmhuffle@femto-st.fr