ISSN 1816-0301 (Print)
ISSN 2617-6963 (Online)

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

INFORMATION TECHNOLOGY

UDC 004.432

Received 21.05.2019
Поступила в редакцию 21.05.2019

Accepted 25.09.2019
Принята к публикации 25.09.2019

On reliability, safety, and readability of programming languages on the example of Ada language

A. V. Leont'ev

Program Systems Institute of Russian Academy of Sciences, Pereslavl-Zalessky, Yaroslavl Region, Russia
E-mail: alex@leont.botik.ru; psi@botik.ru

Abstract. A few small suggestions of language design reflect the author's views are presented. These suggestions are mainly related to the reliability and safety of simple, typical structures and statements in programming: typification, compile-time elaborations of variables, status of variables, high-level statements, etc. The programmers often spend working time with similar constructions. From mathematical point of view these suggestions do not effect seriously on the complexity and efficiency of calculations; they are designed exclusively to help a programmer to write reliable, secure and clear programs. These suggestions are illustrated on Ada language, which is very suitable for this purpose, but they can be applied in other languages also.

Keywords: programming language, reliable programming, safe programming, programming style, design of languages

Acknowledgements. The publication was prepared within the framework of the state assignment of the Ministry of Education and Science of Russia, number AA AAAA-A19-119020690042-2.

For citation. Leont'ev A. V. On reliability, safety, and readability of programming languages on the example of Ada language. *Informatics*, 2019, vol. 16, no. 4, pp. 88–98 (in Russian).

О надежности, безопасности и читабельности языков программирования на примере языка Ада

А. В. Леонтьев

Институт программных систем Российской академии наук, Переславль-Залесский, Ярославская область, Россия
E-mail: alex@leont.botik.ru; psi@botik.ru

Аннотация. Представлены несколько небольших предложений, которые относятся к проектированию языков и отражают взгляды автора на данный предмет. Эти предложения затрагивают надежность, ясность и безопасность типичных, простых конструкций языков программирования, таких как, например, типизация, начальная элаборация, статус переменных, некоторые высокоуровневые операторы и пр. Часто программисты тратят много времени на работу с подобными конструкциями. Представленные предложения мало влияют на сложность и эффективность вычислений. Они предлагаются исключительно для помощи программисту в создании надежных и безопасных программ. Предложения иллюстрируются примерами на языке Ада, который очень удобен для этого, но они могут быть применены и в других языках.

Ключевые слова: языки программирования, надежное программирование, безопасное программирование, стиль программирования, проектирование языков

Благодарность. Работа выполнена по НИР «Исследование и разработка методов интеллектуального анализа данных» в рамках государственного задания Министерства образования и науки России, номер ГР АААА-А19-119020690042-2.

Для цитирования. Леонтьев, А. В. О надежности, безопасности и читабельности языков программирования на примере языка Ада / А. В. Леонтьев // Информатика. – 2019. – Т. 16, № 4. – С. 88–98.

Introduction. The author, basing on his unassuming experience, would like to present a few small suggestions that reflect his subject vision. Though these suggestions are presented schematic enough and the manner of the describing is concise, the author hopes that they are quite clear. It goes without saying that he does not call to implement them immediately; these suggestions are rather offered for a discussion.

Themes of this note are mainly related to reliability and safety of typical, simple structures and statements in programming. Why about them? A good many programmers spend the most part of their time, working with these structures. So, some errors occur during this routine work. Programs are written by human. To err is human. The human factor in programming can hardly be overestimated and is one of the main factors determining reliability of programs.

In fact, having excepted for failures of equipments (in the broadest sense, including errors of a compiler, a run-time environment, and etc) and algorithmic miscalculations, we can attribute to this category all the other errors (that is, to programming proper). Recognition, search, corrections of mistakes of this kind have their own specific and can take long time enough. Though modern languages are more adapted to a making of changes in programs, a correction of these errors can be a separate task. In programming, in which quantity transformed in quality long ago, a loss of a control (over a process) is not a problem. On the whole, it can be said that reliability of a program is its basic characteristic (in the sense, if a program does not have sufficient degree of reliability then the result of its work is sufficiently unpredictable and its exploitation is rather hard or impossible).

Discussing of languages it is very difficult to avoid a question of readability. The precise assessment of this parameter can be difficult but this attribute is very important. In some cases a program can be remade and rewritten if it is complicated and tangled, even if its execution does not have any complaints.

In fact, this is very closely connected with a programming style. It can also be said that programmer's style and language are reflection of those conceptions a programmer had accepted. The author is sure that a programming language, a programming style, and reliability of programs are interdependent not much but very much. As many suppose it is impossible fast writing reliable and clear programs without a good style. It presents difficulty too if a programming language is not suitable. A language directly influences on a speed of writing (a productivity), debugging, reliable, and clarity of a program. In other words, a programming language must be adapted to a programming style. The thought is not new but it is not always (for different reasons) incarnate. Of course, the problems of reliability far exceed the limits of the programming languages' purview but a language, as a basic tool of a programmer, is one of the main starting points.

These proposals little affect on efficiency of computations (in a mathematical sense) or do not affect at all. They are only designed to help a programmer to write reliable and clear programs. This is especially urgent if we take into consideration the growth of software and its complexity. New safety's requirements demands new (and more complicated) tools.

Usual ways of this approach is restrictions and checks with machinery, additions of high-level statements in a language (including libraries), special clear structures of language, and etc. In other words, this shifts a part of the work from a programmer to a machine and simplifies perception of programs. As it is noted in literature, constructions of this kind can some increase *the time of a writing* (of a program). But *the total time of a development*, as a rule, can be (considerably) less since *the time of a testing* and *a debugging* is less. In addition, a program is better worked out and clearer, the number of errors is fewer, its reliability is higher, its maintenance is simpler. In some sense, a program is more self-documented.

Sometimes, maybe, elements of deduction are included in a language. But deduction (at the present level of the development) can be very expensive, complex, and tangled. In whole, at the present time, deduction is rather an experimental tool and a handwork then a extensively used language's tool. It does not mean, of course, that the author disclaims importance of deduction. He does not call for the disorderly chaotic and intuitively-naive programming. On the contrary, an elaborate design facilitates a success. However, the author met examples in which clear language's constructions were much more persuasive then complicated tangled formal proofs. These language's constructions must, obviously, be very clear, reliability, and safety. (All this, naturally, does not impede a using of deduction.) In whole, the problem of software's reliability is very urgent and insistently requires its solution. This problem is multifaceted and, oddly enough, can go far beyond the actual programming. A good help, as it is commonly believed, can be a systematic (punctual) approach being supported by a language and other equipment. In this context, the author is assured that deduction's techniques will have contributed to the solution of this problem. What forms will it be in? The future will have shown.

Let us make a note on ones' opinion here. Some programmers suppose that a programmer should trust himself (but, certainly, not others) and does not check himself with language's tools (for example, with typification, a precondition, a post-condition, and etc.). Probably, every programmer must himself choose his strategy. But it could be noted (slightly exaggerating, of course) that if this was so then programs would not have (contain) errors, all programmers would write their programs with the most effective languages (that are assemblers and machine codes), debugs and compilers would not be, probably, needed, and etc. In other words, to rely upon oneself (and others) is necessary with great prudence. In whole, according to the modern views, a language, a compiler, a development environment, parsers, a run-time environment must comprehensively help a programmer to avoid making errors and to find those that have already been made.

Following folklore, it can be said (maybe some humorously) that every one of the programmers considers himself as an expert. It is quite possible that a programmer can adhere to his own (maybe very specific) concept of programming and can have his own judgment. (In whole, this can be considered as a norm since programmers, as all people, are rather diverse. They have different attainments, knowledges, experience, aims; it would be strange if their opinions were always coincident.) But speaking more seriously, the author supposes that whatever conceptions have been accepted of a programmer, it would be excellent if a programmer could *have a possibility* to choose suitable tools for himself.

The following must be noted. It is applied to section 1. *After this note had been written, some programmers noted (to the author) that these topics had been discussed earlier. Thus, for example, the article* [1] *was submitted at* PLC'05, Las Vegas, NV, June 2005. *But, on the other hand, all these discussions have not found any noticeable embodiments. The author considers these topics being quite interesting. Therefore, after having thought a little, he decided to retain these brief section. First of all, hoping that a languages developers' attention could be turned to these questions once more. In addition, the author has slightly changed this construction. The new design is slightly different from the proposed one in* [1]. *This new design is more flexible, although perhaps more compromise; it should be noted that this construction is developed in* section 2. *The author considers it is appropriate to give an example in support of the construction under discussion, as well (In fact, this section is composed of this example.).*

All these programming maxims, the author has allowed himself to remind here, ought to clarify the motives, the subject, and the purposes of these notes. The author is also grateful for valuable critical remarks and expresses his sincere appreciation to everyone he discussed these issues with; all shortcomings are, naturally, related to the author. So, let us get started...

1. On the value null. In whole, value **`null`** is used in programming widely enough (which is interpreted as an absence of a sensible value). Nevertheless, in general-purpose languages, references' (pointers') variables mainly use this value. Maybe, it should let ordinary variables to take value **`null`**. What is this for? Not all variables can have sensible values during computations. A value can have been not received yet, can being calculated (as yet), can be absent in some situations, and etc. Sometimes it is preferably to say this clearly.

Let us consider the following simple example. Let variable *Index_Of_First_Zero_Element* contain the index of the first zero element of array *Array_A* :

```
Array_A   :   is  array( Integer ) of  Integer ;
Index_Of_First_Zero_Element   :   Integer ;
```

Everything is well if there is a zero element in array *Array_A*. But if a zero element is absent then any `Integer's` values are meaningless. Any assignments of integer type's values are methodologically wrong here. A careful programmer could only get slight irritation if he met an assignment like this. This variable will have some value, of course. Quite possibly that the type of this value will be `Integer`. But this will not be a proper content. An (accidental or deliberate) attempt of a using of this variable can quite cause errors in a program. Here an appropriate value is only **`null`**. A declaration of variables could be expressed, for example, as:

```
Index_Of_First_Zero_Element : Integer with null ;
```

This can be made for a whole type:

```
type Integer_Null is Integer with null ;
```

The syntax can be different, of course. Obviously, its start value of these variables (after of an compile-time elaboration) must be **`null`** unless otherwise has been provided. Next, if a domain has been expanded, it is necessary redefining operations for it. Obviously, if a variable has value **`null`** then this must not take part in primary basic ordinary expressions: so, for example, an attempt of an execution of the follow statements

```
Index_Of_First_Zero_Element  :=  null  ;
i  :=  Index_Of_First_Zero_Element + 1 ;
```

must call an exception handler. This is an expression of that fact that it should not use a variable if it does not have a proper content. Obviously, a variable should be allowed to take part in comparative expressions and assignment statements:

```
Index_Of_First_Zero_Element  := null ;
.....................
if     Index_Of_First_Zero_Element  =  null
then   ... ;
```

The principle "*omnia mea mecum porto*" (all that's mine I carry with me), id est a grouping logically cohesive units together, is vary appropriate for cases of this kind. In whole, it can be said that this programming is more punctual and this structure is safer. This is especially handy if there are lots of variables of this kind. A creation of an object would be somewhat cumbersome in this case. An extension of an index's range is not the best solution too; this does not promote reliability, clarity, and readability of a program. It would be (deliberate) intermixing of different entities.

The proposing design considers using this value **`null`** as *an option*. Of course, the rejection of the global use of this value **`null`** is definitely a serious compromise. But this would allow relatively easy to include this design in an existing language. Although this is a question of a more extensive theoretical research and much depends on the optimizer.

2. On an expanding of types. So, auxiliary value **`null`** can be quite useful. But, as it can easily be noted, this word **`null`** is not the most informative. It can not be said that this provokes an occupational disease but having read words "**`null`**" and "**`not null`**" in some quantity, a programmer can feel some tiredness. This is similar to the situation that was in the early assembler languages, in which values "1" and "0" were very widely used. Perhaps, this is very good for a machine logic but this is not always good for human perception. Not always this conciseness promotes readability, reliability and safety. It is better to read more sensible expressions, certainly.

As it was mentioned above, a "base" value of a variable can be absent for many reasons: a value can be absent at all (in certain cases), can still have been not received, can still being calculated, can be discarded as questionable, can have been used by this moment, and etc. All these reasons could be

reflected with auxiliary values. In other words, it is quite possible not only indicating an absence of a "base" value with **null** but indicating a reason of an absence with additional auxiliary values. All these auxiliary values could improve readability and reliability of programs and could be used in **if-**statements and **case-**statements also. In a manner it is a specification of the value **null**.

Let us continue the consideration of the example (of the previous section). By implication (of the problem) we can select three options for the values of variable *Index_Of_First_Zero_Element*:

1 . array A has not been received (and, accordingly, has not been processed);
2 . array A has been received, has been processed but a zero element was absent;
3 . array A has been received, has been processed and a zero element was present;

It can be noted that all these options are mutually exclusive. Thereby, there are two cases to use auxiliary values (in the third case the variable takes a usual integer's value). The first could be denoted as *Array_A_Has_Not_Been_Received*. The second could be denoted as *Zero_Element_Is_Absent*. It is quite possible, for clarity, that auxiliary values can have the prefix **null**:

1 . **null.***Array_A_Has_Not_Been_Received,*
2 . **null.**Zero_*Element_Is_Absent*.

It could be something like the following:

```
type                                                    --  type declaration
     Type_Index_Of_First_Zero_Element is Integer        --  basic type
with  null                                              --  auxiliary values
        Array_A_Has_Not_Been_Received,
        Zero_Element_Is_Absent

end Type_Index_Of_First_Zero_Element  ;  --  the end of this declaration

                                                        --  variable
declaration  Index_Of_First_Zero_Element                :
Type_Index_Of_First_Zero_Element;
```

Syntax can be different, of course. Here `Integer` is an initial (basic) type, the other values are auxiliary. It is a kind of some a types' union but it is not usual. (This is unusual because of the role that basic and auxiliary values play in statements.) It should be noted that basic and auxiliary types, obviously, must be considered as a single structure; otherwise weakening, softening of a type's control or types' conflicts are possible.

As earlier (in the previous section), if you design structures of this kind, a redefinition of operations and an addition of new attributes must obviously be done (an auxiliary value should be allowed to take part in comparative expressions and assignment statements but not in "usual" operations):

```
Index_Of_First_Zero_Element :=

null.Array_A_Has_Not_Been_Received ;

if Index_Of_First_Zero_Element  = null.Zero_Element_Is_Absent
then   ... ;
```

It goes without saying that this can be applied to accesses' variables too.

3. On constant initializations. As it is known, constants are convenient since it is difficult to modify them during a program execution. But it would be more convenient if facilities of their initialization were broader. In other words, usual tools of a programmer would be appropriate here (loops, if-statements, and etc). This could be a construction in the follow manner (like a procedure):

```
type Type_Array  is array( Integer ) of Integer ;

One_Array :  constant  Type_Array  :=              ---  declaration
```

```
(
      for   I  in    Integer   loop      --- and initialization
                 if       I  =  ..............
                 then     One_Array( I )  := ...
                 end if ;
      end loop ;
);
```

This would increase resources of using constants.

4. On variables' status. Continuing the previous topic, let us consider the follow situation. Let us presuppose that a programmer has decided not to update a value of variable *One_Variable* any more (after a certain point of a program), that is, to use this variable only for reading. In order not to update this value accidentally, a statement "freeze" can be offered. This can be written down, for example, as

```
constant ::   One_Variable := 5 * Other_Variable + ... ;
```

After this statement, variable *One_Variable* becomes a "constant". Thus a programmer gets a new constant during an execution of a program without any efforts. In whole, marking variables that have already been calculated is a good practice. This can also serve as an additional commentary, can discipline a programmer, and even can increase his assertiveness. Probably, a variable can have an attribute on "freezing" status.

Obviously, it should not mix statements "freeze" and assignments in disorder; otherwise conflicts are quite possible. Compiler diagnostics or an exception handler must help to find a programmer's mistake in this case.

A more radical statement can be offered to throw out a variable (as well as procedures, functions and etc) from a program (in order not to be in the way) if this variable is not needed more. For example,

```
away  (  One_Variable, Second_Variable, ... ) ;
```

Many various options and variations of this statement could also be offered. For example, statement "unfreeze". Or, for example, an effect of this statement can be bounded with a block (a provisional freezing):

```
 One_Variable  :=  500 * X + 40 ...  ;
Second_Variable :=  700 * Y - 30 ...   ;
Calculating_Block :          ---------------  the begin of  a new block
declare
                        ----------------------  the provisional freezing
constant  of  Calculating_Block ::  One_Variable,
                                          Second_Variable, ... ;
.........
end   Calculating_Block ;     ---------------  the end of a new block
```

In whole, it can be noted that similar themes appear, for example, on questions of an restriction access to data and resources in the system programming.

5. On complexity of languages, safe segments of programs, the functional programming, and modularity. In a manner (maybe remotely) this section is addressed to the functional programming, which is commonly considered as a some reliable tool. It would be good if a programmer could write a program or its parts in a functional style (or in a resembling manner).

Let us consider the following note. This is related to functions, although it could be related to other program elements (procedures, blocks, objects, packages, tasks, etc.) too. It can be noted that in spite of some similarity the concept of a function in programming covers rather different designs. For example, programmers often distinguish safe functions from unsafe ones. What are the safe functions? This question requires a separate discussion. For example, if a function behaves like a mathematical function (id est functionally) then this is, in generally, considered being safe. If a

function refers to global variables in its body, opens files, allocates a memory, etc. (does something outside itself) then this function is usually considered being unsafe. Basing on this representation, a label of a safe function can be offered:

```
safe function  One_Function ( ... ) ;
```

It should definitely be noted that perceiving these marks only as a kind of comments is not entirely true. Comments are not rigidly related (and are not identical) to programs; they represent only thoughts of a programmer. Formally, they are not connected absolutely. Programs do not always follow comments, from time to time it happens. In whole, it is not the best practice completely to rely upon comments (particularly in small details). This practice will not assist to get a profit. Generally speaking, comments can be anything to please. But, for example, if a programmer considers a using of a library and wants to be assured that side effects are absent then the labels of the library and its subroutines (supported and guaranteed with a compiler) would be a good addition to the library creator's comments. (A program having a lot of side effects is often surveyed, understood and modified with difficulty.) In other words, if a programmer wants to write of this style than he should be afforded this opportunity. (However, side effects can be very convenient in some cases. It can take place, for example, if side effects influence on data that are compact, small, self-contained, easy-surveyed, considered as a single essence (like an object). In this case they have effects but rather formality to some degree. Though, in each case, a programmer must decide himself whether it should be used or not.)

Let us emphasize that this can be attributed not only to functions but to other programming segments too. Different modes of editing, compiling, debugging, testing, pragmas' modes could be entered here. Moreover, maybe it makes sense to make a more subtle classification of functions: for example, classes of semi-safe functions and procedures (id est of new entities) and etc.

In vie of the aforesaid, let us touch a topic of functions in programming, which was discussed rather often; though sometimes it is discussed now too. It can very shortly be formulated of the following way. Should a function (in programming) be allowed that it could change its own parameters (like a procedure)? A function that changes its own parameters is not, obviously, a procedure. On the other hand, this is not a function in the usual mathematical sense and it is not safe. In the judgment of the author this is certainly the *third* essence and can be designate, for example, as “procedfuncion” or in a similar manner. (All unsafe constructions should be attributed to a separate unsafe category.) Though, this is rather a theme for more extensive theoretical researches.

This question adjoins to a more general one that the author would like to dwell on some more. He would like to preface a few remarks to this. Modern languages tend to their considerable complication. Simplification of languages has begun taking the second, third or, maybe, tenth place. This alienates some programmers. And it is not only their own personal attitude. The growth of language complexity is, probably, unavoidable. But, with the growing of language's complexity, many issues of reliability, safety, verification of a languages and applications' programs have begun revealing themselves very much. There is some additional difficulty for Ada since the designers of the language set a goal to create an uniform integrated (practically, universal general-purpose) language (for their own purposes, particularly for real-time systems), which was partly put in order to reduce a number of used languages, to unify processes, to increase reliability. On the whole, this is an attractive idea, having its own logic and necessity. But will this language be controllable, handy, transparent and manageable? Or, will this language be huge and be expanded extensionally, boundless and unmanageable?... For Ada, which was always attributed to the most complex languages, this problem was (and is) actual enough.

In whole, programmers (in a majority, at least) agree that simplicity is a very important factor of reliability. “*The price of reliability is the pursuit of the utmost simplicity.*” - the quote is attributed to C. A. R. Hoare. But, obviously, achievement of absolute simplicity is not always possible. As it is known, a simple language (a low-level language) can be implemented easily and, in this sense, it is reliable. But writing programs with this language is hard enough. If a language is large and complex (a high-level language), its implementation is, as a rule, less reliable (usually, the larger a system, the many more errors (including rough ones) it contains and the harder to find them). But programs of this language are usually much more readable, shorter and more reliable, writing them is faster and easier

(on conditions, of course, that the language is not tangle and is transparent). It is quite clear that this contradiction is fundamental enough. So, how much should complexity be pushed in programs and how much in a language?

On the other hand, modularity is a well-established conception in programming. We could define (in a big language) a few sub-languages of diverse degrees of complexity and reliability (for example, a functional sub-language; although, the degree of compatibility of the functional and object-oriented concepts is an interesting question...). These sub-languages must, obviously, be marked well. A programmer could write programs with simple sub-languages, using more complex sub-languages as necessary. The semantics of statements must not depend on a sub-language, obviously. In whole, a safe language is determined of various factors. It is particularly interesting and important for a programmer to know what kinds of constructions are allowed (and not allowed) of a language for him to use.

In whole, this part of the section can rather be considered as a suggestion-wish. Of course, this work is not simple and it can be fully comparable to a creation of a new language. Though, this is somewhat of different theme.

After these notes had been written, the author met a similar design in Internet. It was the keyword "**`unsafe`**" for modules in language `MODULA-3`, which was created in 1986-1988 years. Some sketchy comments on `MODULA-3` can be found, for example, in [3–5]. In addition, several resembling words on the functional programming were said by A. V. Klimov, A. I. Adamovich at the conference National Supercomputing Forum, Pereslavl-Zalessky, November-December, 2017 [2], though on an other occasion. Well, some coincidence of the views is good.

6. On tree's structures and statement `return`. The safe programming supposes, among other things, that a programmer must very clear envisage a program structure. In this context, a will of a programmer using simple clear structures is quite understandable. Tree's structures are reckoned as sufficient attractive here. They are considered being clearer then complex tangled graphs. They are usually simpler, contains fewer bugs, are tested and verified easier and faster. In whole, it can be said that a building of various hierarchies and orders is rather appreciable work in programming and all these *tree-like structures of data* are often used by programmers and can be very extensive. Their wholeness is usually supported with special procedures. But *tree-like statements* are usually limited enough (for example, these statements **`if`**, **`if-then-else-then-else`**, **`case`**). There is, certainly, some asymmetry here. It would not be bad if a programmer had a more spacious tree-like statement (as a single statement). But it must be convenient, of course. Obviously, this can be realized in various ways. Hereinafter this item can be considered as a light attempt on this subject, which the author would like to propose.

Let us imagine, for a start, that we want to implement an algorithm corresponding to a parse tree (or another algorithm corresponding to a tree's structure). We can implement this, for example, with a procedure-statement $Tree\_Parse(P_1,...,P_k)$ and its subprocedures, which implement this tree's structure (this procedure can have a special mark in the declaration, for example,

`tree procedure` $Tree\_Parse\ (\ P_1,...,P_k\ )\ ;$

or in a like manner). The main point, we would like to concentrate on, is a termination of the computations (of this procedure-statement $Tree\_Parse$).

Let us suppose, further, that its sub-procedure F has fulfilled required operations. Logically and for the sake of program's clarity, it is desirable that the procedure $Tree\_Parse$ (and its subroutines) could complete its work as soon as possible (before a programmer would begin making new bugs). In other words, it is desirable (in the body of subroutine $Tree\_Parse's\ F$) a statement like the following:

$Tree\_Parse$**`.return`** $(\ t_1,...,t_k\ );$

It can be noted that this statement (as any ones that complete a computing branch) makes a graph of a subprogram nearer to a tree. And, in this sense, the graph becomes clearer. Other decisions can be not so convenient here (multiple checks and returns, a call of an exception handler, a using of a **`goto`**-statement, and etc.).

There are some objections here. Let us consider the following situation. Suppose, for example, that certain F_j (a *Tree_Parse's* sub-procedure) makes some unsafe actions (opens files, allocates memory, and etc). There is a practice (or, if you wish, a programming style) that consists in the following. Logically cohesive unsafe actions are tried to group together (in the same subroutine). Otherwise, if they are separated, a programmer can forget about remaining actions (to close files, to free memory and etc) and they will not be written and executed. An occurrence of this kind can quite happen here. There can be a few decisions here. One of the language stock decisions is a restraint of the class of these subroutines (where this **return** can be used). Or, for example, **return** could occur in those places where unsafe actions had not begun yet.

Another question is "What to do if we have a recursive function?". The purpose of this design is a simplification of a program structure. For the recursive functions, probably, similar simplifications will not add clarity. Therefore a suitable solution is, likely, to ban this statement for the recursive functions.

7. On complementary statements. This tiny suggestion, which is a light draft, has concern with statements that are viewed of a programmer as complementary (for example, an opening and a closing of a file, an allocation and a deallocation of memory, and etc). For example, a programmer can make a decision that if he have allocated memory *M*, he must (later but in foreseeable future) free this memory *M* without fail.

It would be wonderfully if these intentions could be implemented with a syntax. Syntax itself can be very various here. We could link two statements in a complementary pair, for example, either in time of the first statement's calling (thereby "pointing out" another (complementary) statement) or in a declaration area (if they are in different units of a compilation or, in other words, if it is necessary to extend their visibility scope). For example, it can be something like the follow:

```
complementary.Open( F,... );
......
complementary.Close( F );
```

This construction seems to be very alluring to the author, though, on the other hand, there are some difficulties here. For example, there are similar constructions almost in all languages, these are left and right brackets. But, excepting simple cases, they are used rather different. Thus, for example, a statement "open file" can be placed in **if**-statements and a statement "close file" outside. A program can begin branching, so several statements "close file" can be necessary for one statement "open file". A syntax must very well be worked out here.

The author would like to draw attention that he considers this rather as a (maybe modest) tool against, for example, memory leak (which is the serious problem in programing) then as a convenient counter for a programmer. In addition, it is the good style to express programmer's intention clearly; this decreases a quantity of comments and does a program more reliable and clearer.

Using constructors and destructors in the object-oriented programming (OOP) we could attempt to create a similar construction but OOP is rather cumbersome (it can be rather tiresome to create an object for every tick). Besides, not all languages support OOP.

8. On declarations of indexes' subtypes and variables. This tiny item related to Ada solely. The author pays attention to this topic since programmers work with arrays often and often, and this work requires some attention. As it is known, language Ada allows an implicit declarator of a counter in a **for**-statement:

```
for   I   in   Array_A'Range   loop
      Array_A[I]  :=   .....
```

Let us suppose that a programmer wants to process array's components and its indexes. It is quite reasonable if the programmer would like to create indexes' variables of the correspond subtype (fit for array *A*):

```
subtype    Subtype_Array_A_Index   is   Type_Indexes
                  range  Array_A'First .. Array_A'Last ;
I, J, K, L   :   Subtype_Array_A_Index ; --  variables' declaration
```

id est in order that values of the variables *I, J, K, L* do not go out of the indexes' limits (of array `Array_A`). But this construction is some cumbersome, especially if there is a lot of variables. Perhaps, a construction

```
I, J, K, L :  Array_A'Range ;  --  variables' declaration
```

is more convenient as in case of a **for**-statement. As far as the author knows, (at least) some Ada's compilers do not permit this construction.

It should be required, however, (for the safe programming) that the scope of these variables must not include statements that change the index's range of array `Array_A` (otherwise their types can be unmatched). If an array is static, this always takes place. If an array is dynamic, this may not be but (rather often) the variables' visibility can easily be limited (localized) by the instrumentality of blocks, procedures, functions, and etc. On the other hand, if a mismatch has happened for all this then it means that a programmer had made a mistake somewhere and a call of an exception handler ought to help a programmer to find one.

Another question is “What to do if the indexes' boundaries (in a case if array `Array_A` is dynamic) has not been defined (by mistake)?”. In this case, probably, an exception handler must be called.

This construction has the following advantages:

1 . it is shorter;

2 . a programmer does not have a need to read and to think of constants in this expression;

3 . a successful compilation of this expression would show a programmer that the utilization of these variables is safe enough.

9. A few words on graphical tools and on some other points. If we turn to the sources, it can be noted the following. Although graphical representations and a graphical processing seemed to somebody no strictly something, it should be recognized that they had great expressive power. Thus, many programmers supposed that widespread commercial success of some well known software products and software companies had been achieved with using of quick and easy graphicals' interfaces (as one of the main factors). This does not absolutely disparage from adepts of a command line, which is a powerful tool of a programmer; it only means that a graphic shape of certain information can be more convenient then a text. Though it could be noted also that a good visualization and visibility changed situation not only in programming. It would be wonderful if compiler developers began more actively to develop tools in this field.

They could improve visualization of calculations, debuging, a graph's presentation of programs, memory work, fast manipulations of modes, data, etc. Maybe, tools for graph's processing would be very useful. Maybe, a compiler's insert of (special) comments or values of variables (fully or partially) into the text of a program (together or instead of variables, of course optional) could be a useful tool of program's debugging. Particular interest is a visualization of parallel calculations and debugging. The author considers that these questions are sufficiently important in hands-on programming.

Tools of this kind are particularly appropriate since verifications of programs are often not carried out, but are carried out some testing and debuging instead of them. However, these tools could certainly be useful for well-developed programs, too, since a program is a set of instructions that must be executed literally and any inaccuracies or typos are not allowed.

Conclusion. An implementation of the most part of these proposals does not have any appreciable difficulties and big expenditures. They entail an additional language's complication. But this complication is quite natural and this increase is more “additive” than “multiplicative”. Thus, for example, a use of a value **null** and its specification can hardly cause any difficulties; a tree-like structure control of a parse procedure-statement does not offer any difficulty; attribution of constants (during an compile-time elaboration) could be made, probably, by a good trained student, and etc. The author supposes that these humble proposals can be useful and improve quality of programmer's work.

References

1. Heinlein C. Null values in programming languages. *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, Las Vegas, Nevada, USA, 27–30 June, 2005. Available at: https://www.researchgate.net/publication/221271111_Null_Values_in_Programming_Languages (accessed 13.04.2019).

2. Klimov A., Adamovich A. Can parallel programs be deterministic by construction? Problem statement. *National Supercomputing Forum (NSCF 2017)*, Pereslavl-Zalessky, Russia, November-December 2017. Available at: http://2018.nscf.ru/nscf-eng/ (accessed 13.04.2019).

3. Freeman S. *Partial Revelation and Mjdula-3*. Available at: https://www.cs.tut.fi/lintula/manual/modula3/modula-3/html/partial-rev/index.html (accessed 13.04.2019).

4. *Modula-3*. Available at: https://en.wikipedia.org/wiki/Modula-3 (accessed 13.04.2019).

5. Wyant G. Introducing modula-3. *Linux Journal*, 1 December, 1994. Available at: https://www.linuxjournal.com/article/9 (accessed 13.04.2019).

Information about the author

Alexander V. Leont'ev, Cand. Sci. (Phys.-Math.), Senior Researcher, Program Systems Institute of Russian Academy of Sciences, Pereslavl-Zalessky, Yaroslavl Region, Russia.
E-mail: alex@leont.botik.ru; psi@botik.ru

Информация об авторе

Леонтьев Александр Владимирович, кандидат физико-математических наук, старший научный сотрудник, Институт программных систем Российской академии наук, Переславль-Залесский, Ярославская область, Россия.
E-mail: alex@leont.botik.ru; psi@botik.ru