PROGRAMMING LANGUAGES: DIFFERENCES IN APPROACH TO ABSTRACTION

Constantin SPRINCEAN

Department of Informatics and Systems Engineering, IA-234, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

Corresponding author: Mihail-Veleşcu Lilia, lilia.mihail@ia.utm.md

Tutor/coordinator: **Lilia MIHAIL-VELEȘCU**, university assistant,
Department of Foreign Languages, TUM

Abstract. *Programming languages can be thought of as abstractions from the assembly, necessary for optimizing human interaction with the computer across various domains. Depending on the intended function of a programming language, different approaches to some aspects of this abstraction are employed. For example, languages created for statistical analysis often incorporate a wide range of built-in primitive operators applicable to arrays, whereas general-purpose languages usually have few primitive operators that can only be used on single values. Furthermore, even languages within the same domain may implement different approaches to abstraction. Consider that some relevant general-purpose languages implement an imperative paradigm, while others – a declarative one. To overview these differences more closely, a little comparative study of three general-purpose languages C, Lisp and Haskell and one specialized language J will be done; more specifically, an algorithm for adding together positive integers of an array will be implemented in these four languages while observing their syntactic and semantic specifics.*

Keywords: *imperative programming, functional programming, metaprogramming, array programming, Lisp*

Introduction

We can think of a programming language as of an abstraction built on top of the assembly language (which, in its turn, is an abstraction built on top of the machine code). It is simply a combination of symbols written in a specific notation that can be transformed to the assembly code by the compiler or the interpreter. But what would that notation be? How do we formalize algorithms that we can express in our natural language in terms of a notation that could be successfully translated to an assembly code?

There are various approaches to these questions, and the massive variety of programming languages show this. This article will focus on comparing these approaches by observing how a single solution to the same problem may be written in different languages, more specifically – C, Lisp, Haskell and J, languages of various paradigms with unique syntactic and semantic properties.

Formulating the sample problem and its solution

We will assume that there is an array of n integers named “A” and the goal is to compute the sum of its positive elements. The algorithm for doing so is rather simple and the pseudocode for it is presented on Fig. 1.

```
A(n) of Integers
sum = 0
for each element of A
    if (element is positive)
        add element to sum
```

Figure 1. Implementation in pseudocode

Even though the algorithm is very simple, it requires cycling through an array, choosing elements in based on a condition, and adding them together. There may be vast differences in how these aspects are implemented in various languages. For demonstrating the algorithm in actual languages, it will be assumed that the array A is defined as {4, -3, -9, 5, 4}.

Implementing the process in C

Firstly, we will observe how this process is formulated in C. C is, as we know, an imperative language with few primitive operators [1]. It is also said to be a language of a "structured" paradigm. Fig. 2 shows us the code implementing the algorithm in C.

```c
int sum_of_positive(int A[], int n) {
    int sum = 0;
    for (int i = 0; i<n; i++)
        if (A[i] > 0)
            sum += A[i];

    return sum;
}

int main() {
    int A[5] = {4, -3, -9, 5, -4};
    int sum = sum_of_positive(A, 5);
}
```

Figure 2. Implementation in C

As a result of this program, the sum of positive elements of the array will be contained in the variable "sum".

Comparing this to pseudocode, it is of notice that, firstly, the form is almost identical to the pseudocode formulation; secondly, it is clearly visible what happens on each step; finally, there is mutability – that is "sum" is called a variable. These are the characteristics of paradigms laying behind C: imperative and structured programming. Imperative programming is characterized by mutability and clear descriptions of algorithms step-by-step [1], and structured – is by usage of sequencing, selection and iteration [2].

This way of notation may seem as most natural, but later it will be shown that this is not the only relevant approach to programming.

Implementing the process in Lisp

Lisp is one of the earliest high-level programming languages [3]. It is a multi-paradigmal language, often specifically being called as a language of functional and meta paradigms [4]. It has many dialects, of which Scheme is used here. Consider the Lisp implementation on Fig. 3.

```lisp
(define A '(4 -3 -9 5 -4))

(define (calc-sum-of-positive A)
  (cond ((null? A) 0)
        ((positive? (car A))
         (+ (calc-sum-of-positive (cdr A)) (car A)))
        (else
         (+ (calc-sum-of-positive (cdr A)) 0))))

(calc-sum-of-positive A)
```

Figure 3. Implementation in Lisp

Lisp has an unconventional syntax. It consists of the so-called s-expressions, each of which can be evaluated. An s-expression can either consist of a single element, called an atom, or of an expression in "(operator operands)" format. Notice how the operator is always in the beginning of the expression, even when writing simple mathematical expressions, like addition in this example. This notation of mathematical expressions is called the Polish notation, or prefix notation. Using it in Lisp helps retain uniformity.

Consider the sample implementation. Here, "define" is an operator of defining a macro (the calc-sum-of-positive macro may also be called a function in the given context, as it behaves exactly like it). On the evaluating stage, the Lisp interpreter substitutes the defined symbolic sequence with its definition (if it exists). The definition of *calc-sum-of-positive* consists of a conditional operator, which generally works the same as in C. Each block after a predicate recursively calls the same function. Recursion is used very often in Lisp, as there is no mutability as in imperative languages; instead, there are s-expressions that can be called anytime and anywhere that will be evaluated somehow.

Notice the "car" and "cdr" operators. Lisp uses linked lists for practically all data storage, which means '(4 -3 -9 5 -4) isn't actually an array – it is a linked list. A linked list in Lisp consists of two elements; "car" is a selector of the first element, and "cdr" is a selector of the second element. The first element of '(4 -3 -9 5 -4) is 4, and the second is the rest of the list: '(-3 -9 5 -4).

As such, Lisp has a pretty unique approach to many aspects of programming. It is a minimalistic language with simple primitives, but built so that these primitives can be combined effectively – thanks to uniformity. This makes it very flexible and capable of easily creating new layers of abstraction [4]. The ability to extensively use macros makes it a suitable language for metaprogramming – a technique in which a program may transform itself or the other programs while executing, and the fact that functions are treated as generic s-expressions, making them capable of being used as function arguments or as returnable objects makes it a functional language too.

Implementing the process in Haskell

Haskell is a pure functional language. The difference between pure and impure functional programming is, per Sabry, that impure languages may allow for mutability in incidental cases [5]. See Fig. 4 for the implementation of algorithm in Haskell.

```
module Main(main) where

sumOfPositive :: [Int] -> Int
sumOfPositive xs = sum [x | x <- xs, x > 0]

main :: IO()
main = do
    let a = [4,-3,-9,5,-4]
    print (sumOfPositive a)
```

Figure 4. Implementation in Haskell

Haskell is a statically typed language with type inference. That is why there was a need to specify the type of the function *sumOfPositive*, but not the type of the array when declaring it under *main*.

"sumOfPositive xs" indicates that the argument of the function will be called "xs". Now, after the equal sign, a built-in *sum* function is called on the list of all positive elements *x* of *xs*. Consider the notation in square brackets after *sum*. It resembles the set-builder notation from mathematics and reads as follows: "the set of *x,* such that each *x* is an element of *xs* and each *x* is larger than 0". This syntactic construct is called the list comprehension [6] and it is also used in other programming languages, like Erlang, Python, Julia and others. It returns a list of positive elements from *xs*, which is then passed to *sum*, producing the needed result.

Implementing the process in J

J is based on APL. It is an array programming language narrowly-focused on applying various operations on large chunks of data.

As J is specialized on working with arrays, it is safe to assume that it would be a good choice for resolving the problem of filtering the contents of an array and finding sum of some of its elements. Fig. 5 contains the program in J for doing so.

```
A =: 4 _3 _9 5 _4
A_mask =: A >0
+/ A_mask # A
```

Figure 5. Implementation in J

Negative numbers in J are marked with "_" instead of "-".

In the first line, the array is assigned to the variable A. Next, "A >0" is assigned to the variable A_mask; but what does "A >0" mean? It returns a binary array of the same size, replacing integers that do not match with ">0" predicate with 0, and those that do match – with 1. It will be used as a mask to select those positive elements in the third line: the "#" operator returns an array where each element A_i of the second operand appears A_mask_i times, meaning elements from A corresponding to 0 in A_mask do not appear at all, and those corresponding to 1 appear once. Finally, this returned array is summed up by the "+/" operator that performs addition of all elements in the given array, producing the anticipated answer.

Notice how there was no need for declaring and describing the ">0", "+/" and "#" operations, though their function is not really generic and they are used in a narrow (compared to very generic and necessary operators in C or Lisp) range of situations. This large number of primitives and built-in operators is what separates J from other programming languages.

Conclusions

We have observed how various programming paradigms approach the same computational process. Each code written here produces the same result, the difference being in the code itself – in other words, how differently the same idea is formalized. These differences mainly impact the process of coding: it is without a doubt easier to work with statistical data in languages with high level of abstraction such as J, without having to worry about, for example, managing memory. This leads to differences in popularity of programming languages across different domains. Besides, although imperative programming languages are dominating the software development industry, there is no clear consensus on whether they are the most effective choice. Some people argue that functional languages are better suited for developing complex software, and that the current state of the industry is due to random factors [7]. This article has showed how differently a simple algorithm may be implemented in imperative and functional languages, and a more in-depth study is needed to fully address this debate.

References

[1] R. W. Sebesta, *Concepts of Programming Languages (10th Edition)*, Pearson, 2012, ISBN 978-0-13-139531-2.

[2] "What is Structured Programming?" [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/structured-programming-modular-programming . Retrieved on 10.04.2024.

[3] "Conclusions" [Online]. Available: https://web.archive.org/web/20140403021353/http://www-formal.stanford.edu/jmc/history/lisp/node6.html . Retrieved on 10.04.2024.

[4] G. L. Steele, R. P. Gabriel, "The evolution of Lisp", *History of programming languages---II*, pp. 233–330, 1996, doi: 10.1145/234286.1057818.

[5] A. Sabry, "What is a purely functional language?", *Journal of Functional Programming*, vol. 8, issue 1, pp. 1–22, Jan. 1998, doi: 10.1017/S0956796897002943.
[6] "list comprehension from FOLDOC" [Online]. Available: https://web.archive.org/web/20050125080818/http://ftp.sunet.se/foldoc/foldoc.cgi?list+comprehension . Retrieved on 10.04.2024.
[7] Y. Yang, "An Overview of Practical Impacts of Functional Programming", 2*4th Asia-Pacific Software Engineering Conference Workshops*, 2017, doi: 10.1109/APSECW.2017.27.