1. **“Growing a Language” by Guy L.Steele Jr.**

In this talk, Guy L. Steele talks about the approach the programming language designers should take in designing their language. The whole talk can be divided into parts.

In the first part, the speaker discusses the correlation between the size and popularity of programming by exemplifying languages that were used that time. According to him, the big language is not popular among users simply because it is too big to learn (for example, PL/I). But that does not mean that small language can succeed simply because of its size. Mr.Steele points out that the programming language will be in use for a long time if it is a small language at first, but is designed to grow in the near future. This leads us to the second part of the talk: language growth.

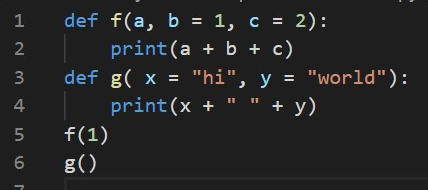
In the second part, the speaker discusses the kinds of growth in a language and who and how should contribute to this growth. The author claims that it is the users who should help grow the language by adding functions they really need. A successful example of such growth is LISP, a language that had grown fast due to the experimentations of users. One of the most notable features of LISP that is still in use is garbage collection. As it was emphasized in the first part, the language should be able to grow with its user. Therefore, a main goal in designing a language should be to plan for growth. The growth can be characterized by the change in vocabulary or the interpretation rules and according to Mr. Steele, there are two ways to grow the language. The first way is to have a person/group-in-charge to take in, test, and judge. The second one is making source code available to public. Both approaches are not ideal, but their combination – putting the source code out there and having someone to recognize good work and implement it – is faster and better than all other ways. This approach does not alienate users and makes it possible for them to work on the issues that matter to them. Following this logic, Mr.Steele suggests that instead of designing a thing, but the design of languages should focus on designing a pattern, a pattern that enables users to grow a language. As author notes, if generic types and operator overload are added to language, the users can grow it in many different ways. But all the user contributions do not have to be added to the language, since diversity of user needs may make the language too complicated. Instead, the desired feature can be coded by a few and put up as libraries for public use; there is no need in making it a part of the base language. This concludes the main ideas of this talk.

For my opinion, I totally agree with Mr.Steele’s claims. In real-world languages like English, etc. constantly evolve via the people who speak it and this evolvement ensures their existence. Analogously, programming languages should grow via their users and adjust to their needs and the only way to do it is to make language growth more natural for every user. And this idea is still popular today, considering how Java is among the most widely programming languages even more than two decades after this talk. I believe that this idea will continue to be important in language design as more industries utilize programming.

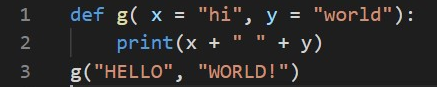
1. **Growing a language**
   1. **Choosing a Feature**

* **Feature:** Functions with default arguments
* **Programming language:** Python
* **Base language:** MRFVAE
* **Description:**

This feature allows users to create function with default parameters and call them with less to no arguments.

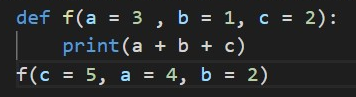


When the function is called, the default parameters can be changed to the values specified by user:



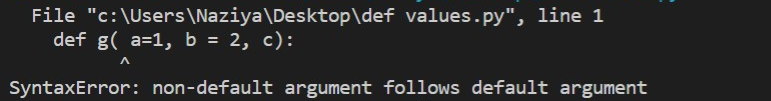
Here the default values of x and y get overwritten when the function is called, so the output would be “HELLO WORLD!” instead of “hi world”.

Additionally, in the function call site the arguments can be written in an unordered fashion if the parameters to which they are assigned are indicated. For example:

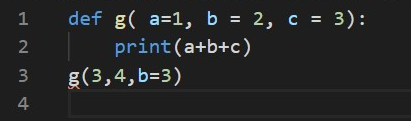


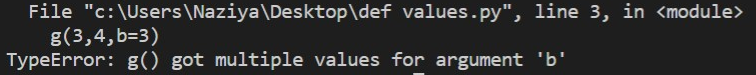
However, there are some limitations:

In case when the function have both default and non-default parameters, during the function declaration non-default ones should be written before the default ones. The violation of this rule gives an error:



Additionally, referring to the same parameter is also unacceptable:





* + 1. **Links to materials**
* <https://www.geeksforgeeks.org/default-arguments-in-python/>
  + 1. **Justification of benefits**

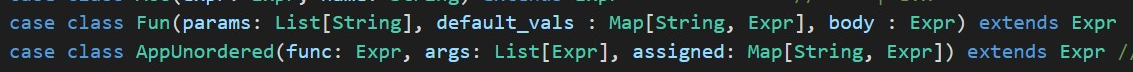
In the language MRFVAE one can create and call functions with multiple arguments. However, MRFVAE’s usage may have limitations for programs of large sizes with repeated functions that have a lot of parameters, whose values may change for specific cases. When you call a function in MRFVAE, the number of arguments and parameters should be the same, if you call a function with less arguments the program gives you an error. Inability to call the function with less arguments than parameters is the first limitation of MRFVAE.

Secondly, one has to know the order of parameters in the declared function to be able to call it. In the cases when there are a lot of arguments it may be problematic for programmer. Having to follow the declared parameter order in function call is the second limitation of MRFVAE.

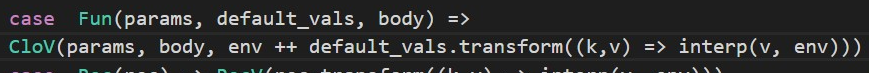
In addition to that, for cases where some function arguments have the same value most of the time, but not always, repeatedly writing the value during every function call may be tedious for programmer, but making the value fixed inside the function body may cause problems when the value needs to be changed (you will have to define a new function).

Considering the limitations and possible problems they create for programmers, functions with default parameters offer some flexibility in function declaration, call and reduce mental load on the programmer since during the function call arguments are only given to the parameters that need to be varied. Furthermore, Python’s function call with unordered arguments makes programmers’ routine easier since keeping just a few parameter names in mind is easier that the order of several parameters and names together.

Let’s look at how the feature overcomes 2 limitations of MRFVAE. To overcome the first difficulty, we allow the assignment of expressions to parameters in function declaration. So the “Fun()” gets an additional parameter default\_vals, which is Map[String, Expr]:



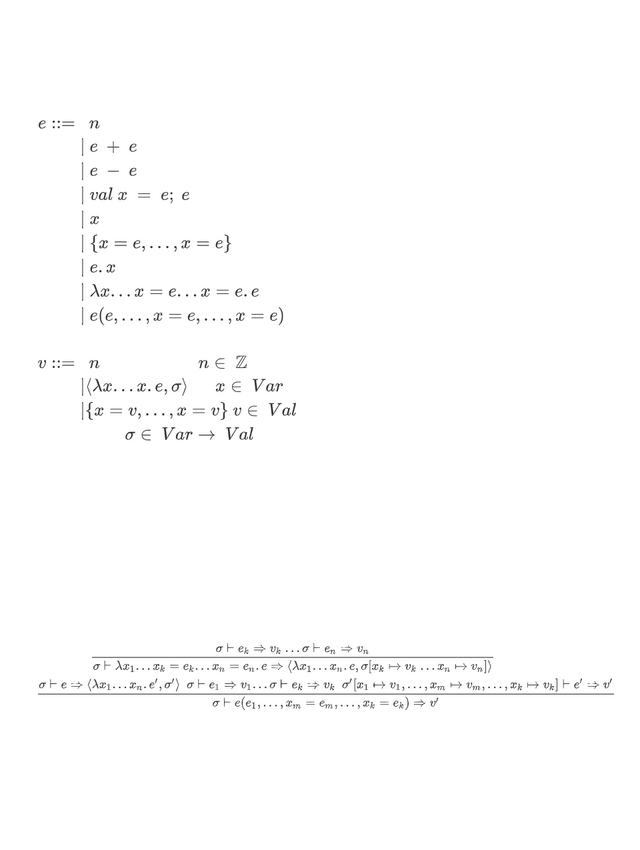
The assigned expressions get interpreted and added to the function environment, so during the function call this values are available to use in body interpretation.



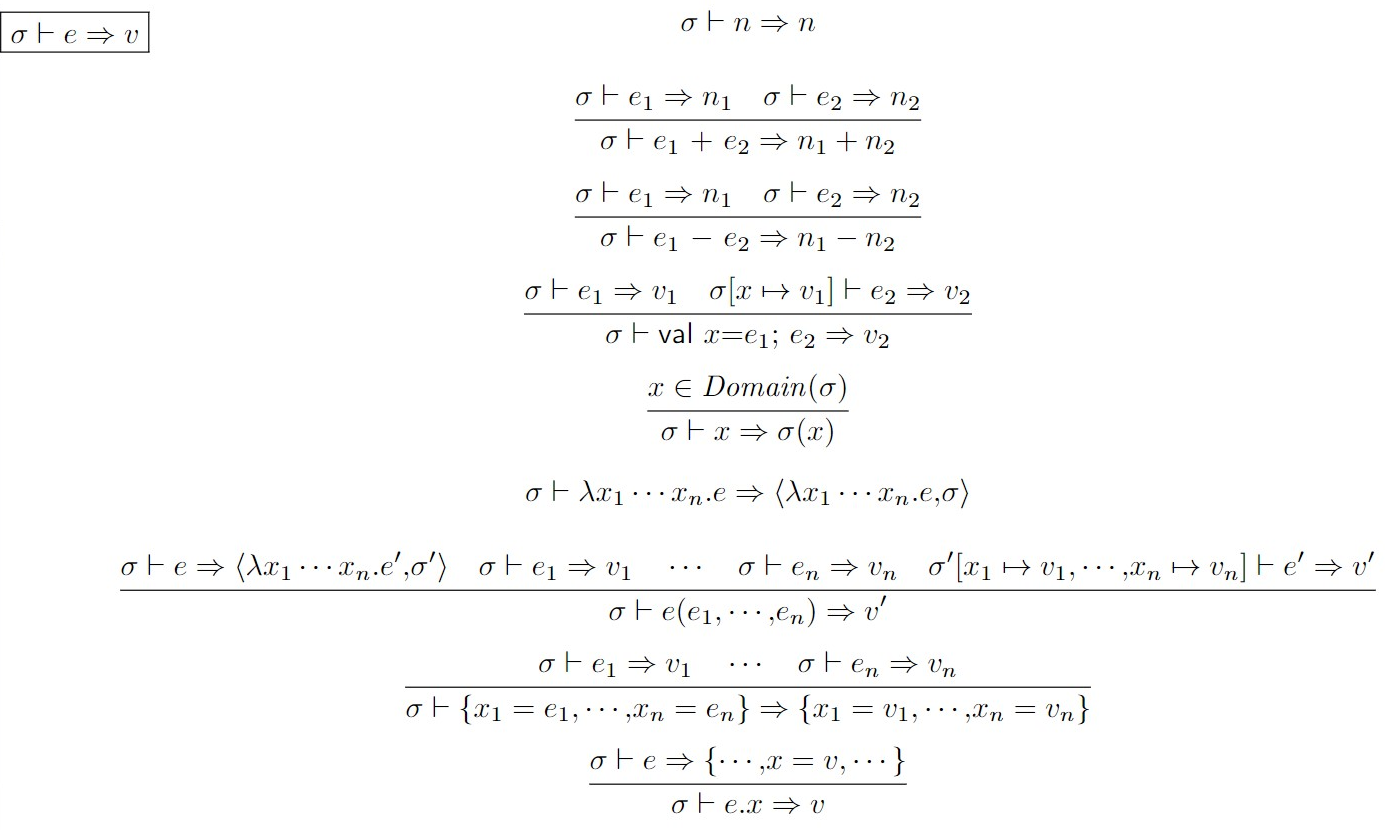
Using this, expressions { val h = { (x, y, z, w) => (x + w + z + y) }; h(1, 4, 5, 6) } and { val h = { (x=1, y=4, z=5, w=6) => (x + w + z + y) }; h() } yield the same result.

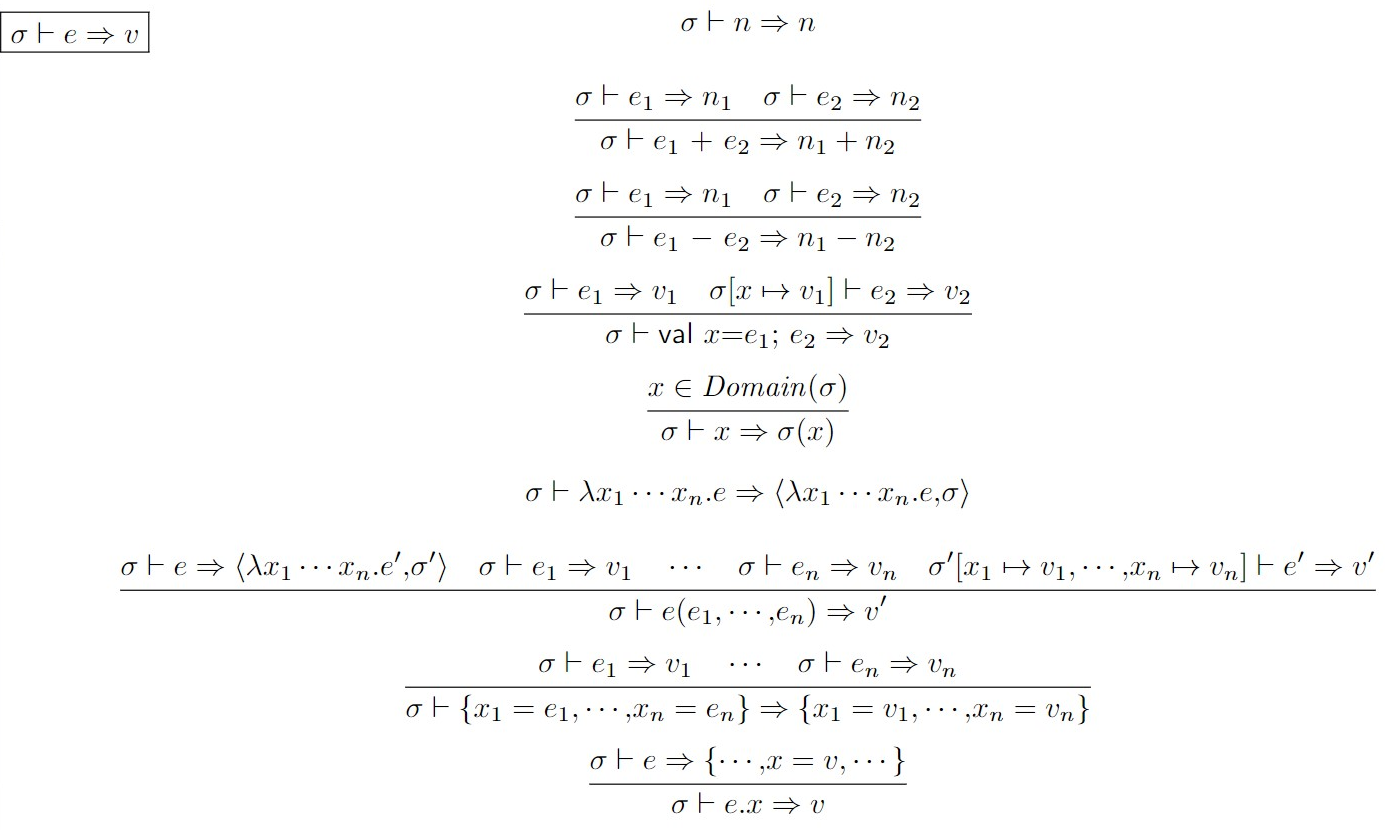
To overcome the second difficulty I allowed the function parameters to be assigned to expressions during the function call. During the interpretation the assigned expressions get interpreted and mapped to the respective parameters. Previous expressions can be written as{ val h = { (x, y, z, w) => (x + w + z + y) }; h(y = 4, x = 1, w = 6, z = 5) } and yield the same result with this extension (you can see these expressions in my implementation test cases).

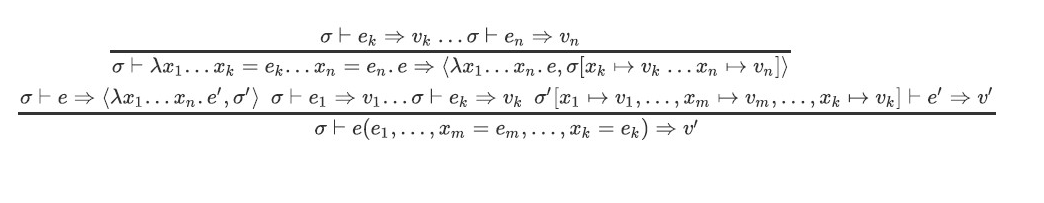
* 1. **Abstract Syntax**

****

* 1. **Operational Semantics**

****

****

****