Operator Language: Design Goals and Intent

David Friant

May 29th, 2026

Abstract

One of the minor inconveniences of C and C++ is the small selection of operators allowed for use within the language. This limits the ability of programmers to express certain ideas, especially those involving formal logic and set theory, in clear and compact ways. This article lays out the beginnings of a project to create a new programming language, Operator, in the C-Family that addresses this and several other minor issues which could be resolved with the aid of full Unicode support and a specialized compiler.

Key Words: Programming Language, Operator, Lexical Analysis, Unicode Support

Introduction

Creating a programming language is a daunting project, but one worth undertaking for the sake of learning, truly, how programming languages function. This work is intended not as a serious attempt to make a language which could dethrone the likes of C++ or Python, but as a personal challenge and learning experience. The design goals and the motivation for them will be discussed in a following section, but first it should be established just what *precisely* a programming language is.

What is a Programming Language?

At their most fundamental level, computers[A] are machines which sequentially[B] read instructions encoded in machine code[21] and perform specific actions based on each instruction. The set of possible instructions that can be read by any given computer is defined by its instruction set architecture[C] (ISA)[17].

Programming languages are, in short, contrived formal languages[16] which are compiled[13] to generate the instructions (and associated data) specific to a given ISA. That is to say that a programming language provides a convenient (or not[D]) way to specify instructions for a computer in a manner which both abstracts programs from specific ISAs[E] and makes the process of writing correct[F] programs faster and easier.

The complexity of programming languages can vary widely. Assembly language[8], for example, strongly corresponds to the ISA it is intended to be compiled (assembled) into. This provides a nearly one-to-one matching between the code that is written and the machine code that is the end result. Python[3], on the other hand is highly-abstracted from the machine code, indeed it runs as an interpreted[18] language in many circumstances. Something like C++[11] is somewhere in the middle where low-level commands (often those inherited from C[12]) are very close to the metal[G] while the complexity of the language allows for high levels of abstraction if necessary or desirable.

Design Goals

The main impetus for the development of the Operator language was the small selection of operators[23] allowed within the C and C++ languages. While sensical from a historical perspective, as only a small num-

[A] Computers based upon the Von Neumann architecture[30], that is.

[B] Barring control flow commands.[15]

[C] Modern examples: x86-64, ARM, and RISC-V.

[D] Brainfuck[10] is an infamous example of an inconvenient esoteric language

[E] Well, sort of. This assumes that a compiler from the chosen language to the target ISA exists, and, often times, low-level optimizations require targeting specific ISA instructions in non-abstractable ways.

[F] The definition of 'correctness' is difficult to define for all but the most trivial programs. This will be discussed further in a later article.

[G] "Close to the metal" is a phrase used in computer science to roughly describe writing code close to what the ISA defines, i.e. with few abstractions.

ber of operators were defined in ASCII[7] at the time of C's creation, modern Unicode[29] supports a large number of operators[4, 5]. Thus it is sensible to allow more operators to be used when writing code. In theory, this could allow for more compact, readable code, albeit at the cost of a somewhat lengthier writing process as modern keyboards are typically not set up to easily allow the use of even a small fraction of the possible Unicode characters. This should be considered a reasonable trade-off as code is (hopefully) run many more times than it is read and read many more times than it is written. Further, there is potentially some ambiguity introduced when using overloaded operators which may have different semantic meanings when different parameters are given, e.g. addition and string concatenation.

In more concrete terms, the Operator language is, at its base level, intended to be a language largely in C-family[20] with a greatly enhanced selection of operators to use when writing programs. Other, more aspirational goals are included in the following non-exhaustive list:

- Greatly expand the allowed operators when writing programs.
- Allow for multiline structures such as matrices[22] to be written naturally.
- Allow set builder notation[27], or some close approximation of it, to be used when natural to do so.
- Allow the definition of operators beyond simple unary and binary infix ones such as specialized bracket operators and perhaps even overline, underline, strikethrough, etc. operators.
- Utilize Unicode characters to allow for the abbreviation of certain common identifiers such as **int32_t**, **double**, and **bool**.
- Provide first-class support for rational numbers[25], complex numbers[14], and quaternions[24].
- Allow the definition of limits on function parameters such that they can be verified as unable to cause problems at compile and/or runtime.
- Implement a more modern module/namespace system than the preprocessor *#include* directives of C and C++.

The next section provides a few examples of what the programming language should look and feel like. The syntax highlighting is provided by a prototype lexer[19] written using the Pygments[2] Python library. As it is only a prototype, readers should be prepared to forgive some inaccurate highlighting and inconsistencies.

Examples

Listing 1: A 'Hello, World!' type program. Note: the monospace nature of typical listings is very likely broken here. This is due to the poor font support for many of the glyphs that are to be used in this project. A complementary project to construct a monospace or duospace font to provide consistent rendering will be introduced at some point.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is a basic 'Hello, World!' type program to provide a gentle introduction
3  ※ to the syntax.
4  ※※
5
6  ※ Imports for CLI IO and ASCII text strings
7  import std.io.print;
8  import std.string.ascii;
9
10 ※ The main method of the program.
11 ℤ32 main(ℤ32 argc, 𝔹8** argv){
12     𝔸 myString ≔ "Hello, World!";
13     print(myString);
14     return 0;
15 }
```

Listing 1 provides the traditional "Hello, World!" program that is written when learning any new programming language. Here, one should first note the use of the reference mark[26] "※" as a character for delimiting both line (※) and block (※※...※※) comments. The **import** commands roughly correspond to how the module system is imagined to be implemented. The blackboard bold[9] characters , , and are Unicode aliases for the (quasi)-fundamental types ASCII-encoded text, bits, and signed integers, respectively. A full listing of fundamental and first-class types and aliases will be included in a later article.

Overall, the syntax should be deeply familiar to anyone who has written C or C++ before, and that is by design. The goal is mostly to enrich the possibilities for writing compact, readable code while sticking to known conventions where they're useful. Though, there may yet be changes. The syntax for pointers, for example, is not something that is entirely set in stone as of yet do to C-style dereferencing potentially conflicting with the definition of a unary prefix * operator. That is something that will be determined in the future, though the @ symbol may be a contender.

Listing 2: A small program demonstrating some boolean logic.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is used to display the use of operators for boolean logic.
3  ※※
4
5  ※ Imports for CLI IO
6  import std.io.print;
7
8  ※ The main method of the program.
9  ℤ32 main(){
10     ※ 𝔹1 and bool are aliases for bits1, i.e. a bitfield of length 1 (it will
11     ※ still almost certainly be packed into a byte sized memory location.)
12     ※ true,  1, and ⊤ (U+22A4) are all equivalent, bits1 = 1
13     ※ false, 0, and ⊥ (U+22A5) are all equivalent, bits1 = 0
14     𝔹1 a ≔     1;
15     𝔹1 b ≔  true;
16     𝔹1 c ≔     ⊤;
17     𝔹1 d ≔     0;
18     𝔹1 e ≔ false;
19     𝔹1 f ≔     ⊥;
20
21     ※ In logic, AND (∧) typically has a higher precedence than OR (∨)
22     if(a ∧ d ∨ ¬b ∧ e){
23         print("This should not print!");
24     }
25     elif(c ⊻ f ∧ (d ⊼ e)){ ※ Demonstrate use of XOR (⊻) and NAND (⊼) operators
26         print("This should print!");
27     }
28     else{
29         print("This should not print!");
30     }
31
32     return 0;
33 }
```

Listing 2 provides a simple demonstration of the utility of having an expanded set of available operators. Starting from line 22, the symbols for logical NOT (¬), AND (∧), and OR (∨) are used in place of the C-style !, &&, and ||, respectively. This isn't a large gain on its own, but line 25 shows the use of the XOR ⊻ and NAND ⊼ operators, greatly simplifying the statement compared to if it was expanded into only ¬, ∧, and ∨ expressions. Further, because the boolean type is considered a bitfield (of length one) first and foremost, these same operators can be used on longer bitfields to perform bit manipulation.

The boolean values themselves are defined in a variety of equivalent ways, including the use of the up- and down-tack characters ⊤ and ⊥, not to be confused with the Latin capital T and perpendicular symbol ⊥ (U+27C2). While it may be equivalent to writing a 1 or 0, using the specified symbols allows the code author's intent to be clearly inferred from the choice of symbol: this is a boolean value. Further, while a programmer could accidentally assign a value of 1 to an integer instead of a boolean, a compiler could/would catch an attempt to assign an ⊤. This provides an opportunity for best-practices to double as a way to catch potential bugs.

Listing 3: A small program demonstrating some complex number computation and the definition of a new operator.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is used to display the use of complex numbers and the definition of a
3  ※ new operator.
4  ※※
5
6  ※ Imports for CLI IO and complex numbers
7  import std.io.print;
8  import std.math.complex;
9
10 ※※
11 ※ This function defines a method for checking if two 32-bit floating point
12 ※ number are approximately equal to each other. Note use of absolute value
13 ※ bracket notation.
14 ※※
15 𝔹1 approx(ℝ32 a, ℝ32 b){
16     return |a - b| < 0.001;
17 }
18
19 ※※
20 ※ Define the ≈ operator for the approx function.
21 ※ Note the order of definition: symbol, function, precedence, associativity.
22 ※ (Precedence value currently completely arbitrary.)
23 ※※
24 binary_infix_operator(≈, approx, 250, no_assoc);
25
26 ※ The main method of the program.
27 ℤ32 main(){
28     ℂ32 a ≔ 1.0 + 0.5i; ※ Define ℂ numbers in a natural way
29     ℂ32 b ≔ -0.5i;
30     ℂ32 c ≔ 0.0;
31
32     b ≔ b*;     ※ Complex conjugation operator
33     c ≔ a · b; ※ Use of dot multiplication notation
34
35     if(Re(c) ≈ -0.25 ∧ Im(c) ≈ 0.5){
36         print("This should print!"); ※ Assumed native system encoding
37     }
38     else{
39         print("This should not print!");
40     }
41     return 0;
42 }
```

As the expanded array of potential operators is the main idea of this language, it only makes sense to demonstrate how to assign a function to one. This is shown in Listing 3 where the character ≈ is defined to be a binary infix operator assigned to a function which checks if two 32 bit floating point numbers are approximately equal to each other. This makes the actual check on line 35 quite easy to read and write. The alternative would be to have the function call spelled out explicitly or, worse, simply write the expression there twice. Either option would result in longer, less clear code.

Listing 3 also demonstrates the native support for complex numbers that will be built into the language. As can be seen, there is good native support for normal complex number operations such as complex conjugation and isolating just the real and imaginary components. This support will extend to quaternions as well, in time.

Listing 4: A small program demonstrating some matrix and vector computation. This introduces the notation for templates with the true angle brackets (not greater-than, less-than signs). Note: The misalignment caused by poor monospace font support for the characters used is too bad to use the and characters for vectors and matrices, respectively.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is used to display some vector and matrix computations.
3  ※※
4
5  ※ Imports for CLI IO and matrices
6  import std.io.print;
7  import std.math.constants;
8  import std.math.matrix;
9  import std.math.vector;
10 import std.math.trigonometry;
11
12 ※ The main method of the program.
13 ℤ32 main(){
14     ℝ32 θ ≔ π / 4.0;
15     ※ Create a rotation matrix about the y axis
16                                 ⎡ cos(θ), 0, sin(θ)⎤
17     matrix⟨3, 3, float32⟩ A ≔ ⎢      0, 1,      0⎥
18                                 ⎣-sin(θ), 0, cos(θ)⎦;
19
20     ※ Create a vector to rotate
21                                ⎡ 1.0⎤
22     vector⟨3, float32⟩ b ≔ ⎢-1.0⎥
23                                ⎣ 0.0⎦;
24
25     vector⟨3, float32⟩ c ≔ A·b; ※ Matrix/Vector multiplication from right to left
26     ℝ32 d ≔ [1.0, 0.0, 0.0] · c;
27
28     if(|d - ⌷2.0 / 2.0| < 0.001 ∧ ‖c‖ ≥ 1.0){
29         print("This should print!");
30     }
31
32     ※ Print the ceiling and floor of d. Note that these are different characters
33     ※ than those used to delimit vectors and matrices
34     print(⌈d⌉);
35     print(⌊d⌋);
36
37     return 0;
38 }
```

Listing 4 provides an example of vector and matrix declarations and computations. The goal here is largely to simplify the syntax of a declaring matrices compared to what might be expected in C with many nested curly brackets. It is still somewhat undecided on whether to allow only square matrix brackets, round brackets, or both. Further, there is a small bit of ambiguity surrounding the declaration of a row vector, as seen on line 26. This will be something that should be handled with care when writing a formal specification for the language.

This example also showcases the use of templates for the first time. The mathematical angle brackets ⟨ ⟩ are used for this in place of the greater- and less-than signs <> typically used in C++. This choice was made largely to avoid ambiguity, though the angle brackets do have some other uses. As such, this was mostly a matter of moving the ambiguity problem from highly-used symbols <> to less commonly used ones.

Listing 5: A small program demonstrating some set-builder notation and some templated container classes.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is used to display some basic set-builder notation.
3  ※※
4
5  ※ Imports for CLI IO and template arrays
6  import std.io.print;
7  import std.io.array;
8
9  ※ The main method of the program.
10 ℤ32 main(){
11     ※ Create static and dynamic arrays
12     Static_Array⟨10, ℕ8⟩ arr ≔ {0, 8, 3, 7, 5, 9, 0, 2, 5, 1};
13     Dynamic_Array⟨ℕ8⟩ vec;
14
15     ※ Put all values of arr that are not divisible by 3 into vec
16     for(x ∈ arr | x % 3 ≠ 0){
17         vec.append(x); ※ Append operator?
18     }
19
20     ※ Print if vec is not empty and there exists an element of vec that is equal
21     ※ to 3.
22     if(vec ≠ ∅ ∧ ∃ x ∈ vec | x = 3){
23         print("This should not print!");
24     }
25
26     print(vec); ※ Assumes some sort of to_string method exists for Dynamic_Array
27
28     return 0;
29 }
```

Listing 5 again showcases the use of templates a bit. However, the main point of interest here should be the use of set-builder notation in the for loop and if statement. This notation provides a convenient and compact way to iterate through some and/or all elements of a container. This is most useful in separating the code for selectively choosing elements from the code which operates on the elements chosen. Further, the use of the existential quantifier ∃ and universal quantifier ∀ can allow for compact checks on the characteristics of elements of containers. This is clearly demonstrated on line 22 where the existential quantifier acts on all elements of the dynamic array to check if there are any equal to three.

Listing 6: A small program demonstrating some theoretical ways of function parameter validation.

```
0  ※※
1  ※ This is a an example prototype program for the Operator programming language.
2  ※ It is used to display compile and/or runtime validation of function
3  ※ parameters.
4  ※※
5
6  ※ Imports for CLI IO
7  import std.io.print;
8  import std.string.ascii;
9
10 ※ Define a division function which prevents denominators of 0
11 ℤ8 protected_div(ℤ8 l, ℤ8 r | r ≠ 0){
12     return l / r;
13 }
14
```

```
※ Define protected_div operator, precedence still arbitrary
binary_infix_operator(÷, protected_div, 250, left_assoc);

※ The main method of the program.
ℤ32 main(ℤ32 argc, 𝔹8** argv){
    ℤ8 a ≔ stoi(𝔸(argv[1])); ※ Convert bytes to ASCII; interpret as integer
    ℤ8 b ≔ stoi(𝔸(argv[2]));
    ℤ8 c;

    ※ Assuming compile-time checking; must validate b ≠ 0 before division, else
    ※ throw compiler error. Possibly very expensive at compile time
    if(b ≠ 0){
        c ≔ a ÷ b;
    }

    ※ Assume run-time checking; function throws error if r ≠ 0 returns false
    try{
        c ≔ a ÷ b;
    }
    catch(Exception e){
        print(e); ※ 'Function parameter assertion false' or something like that.
    }

    ※ Need to decide on paradigm, or perhaps both with different syntax?
    ※ Perhaps validate at compile-time if possible, do run-time checks
    ※ otherwise?

    return 0;
}
```

For the final prototype example, Listing 6 demonstrates two possible ways that function parameter validation might work. Here, the parameters of functions may defined with a ”such that” delimiter borrowed from the set-builder notation. This validation is theorized to work in one of a few ways:

1. At compile-time, validate that the parameter must have an acceptable value. This could be ensured by either validating that the parameter has an acceptable before calling the function, or by tracing the parameter back to its declaration and finding that its value never becomes unacceptable at any time before the function call. This could potentially be very expensive at compile time, perhaps even impossible in some circumstances.

2. At run-time, the validation is performed after the function call but before entering the body of the function. If the value is found to be unacceptable, an exception should be thrown and handled (or not). This has the potential to be slow at run-time. What's more, this does open the opportunity for the validation procedure to have side-effects (intentional or otherwise) which may not be entirely clear to someone reading the code.

3. Some combination of 1 and 2 is possible as well. In this case, one could imagine that compile-time checks are performed as much as possible and run-time validation is added where compile-time checks are impossible or too lengthy.

As a somewhat deeper problem, there is the issue of type signature[28]. Functions are given type signatures when they are compiled. This is how overloaded functions are discriminated against each other. It is very likely difficult or impossible to include such validation information into a type signature (if it should even be desirable), thus the functions will almost certainly compile down to standard functions for the sake of library creation. In a way, this is convenient as it means that any library compiled for Operator should be compatible with another language if the function signatures are created in, for example, a C header file.

Next Steps

As all of the examples given here are simply prototypes to check the look and feel of an imagined language, the next step is to begin making the language real. This has somewhat been started already by building the prototype lexer with Pygments. However, a Pygments lexer is unlikely to be as useful in the long term as a dedicated one written in C or C++. As well, some language features may need careful consideration; the row-vector definition problem described before is a good example of such a problem. Thus, the next steps in this journey will be two-fold. First, a more formal

(though not overly lengthy) specification for just the most fundamental aspects of the language should be written with an eye towards future expansion. Only then should a dedicated lexer be created.

After a specification is written and a lexer created, it will be time to create a parser for converting programs written in the language to an abstract syntax tree (AST)[6]. While it is tempting to directly convert the AST directly into x86-64 or some other machine code, a more viable approach is to convert (transpile) the AST into LLVM Intermediate Representation (IR)[1]. The IR can then be used with the LLVM compiler to produce executable code. However, that is a long way off from where this project stands today.

As a final note, there will be a complementary project started to create a font (probably duospace) which supports many, if not all, of the characters that are useful in a clean manner. As can be seen from the code listings here, the current fonts used do not provide adequate support for characters such as 𝕄 and even ※. This will have to be a long term project that slowly accumulates improvements and updates as time progresses. The simple scale of attempting to cover even a fraction of the characters defined in Unicode is daunting, to say the least.

References

[1] Llvm intermediate representation. https://llvm.org/docs/LangRef.html. Accessed: 2026-05-29.

[2] Pygments. https://pygments.org/. Accessed: 2026-05-27.

[3] Python programming language. https://www.python.org/about/. Accessed: 2026-05-27.

[4] Unicode chart: Mathematical operators (2200-22ff). https://www.unicode.org/charts/PDF/U2200.pdf. Accessed: 2026-05-27.

[5] Unicode chart: Supplemental mathematical operators (2a00-2aff). https://www.unicode.org/charts/PDF/U2A00.pdf. Accessed: 2026-05-27.

[6] Wikipedia: Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree. Accessed: 2026-05-29.

[7] Wikipedia: Ascii. https://en.wikipedia.org/wiki/ASCII. Accessed: 2026-05-27.

[8] Wikipedia: Assembly language. https://en.wikipedia.org/wiki/Assembly_language. Accessed: 2026-05-27.

[9] Wikipedia: Blackboard bold. https://en.wikipedia.org/wiki/Blackboard_bold. Accessed: 2026-05-28.

[10] Wikipedia: Brainfuck. https://en.wikipedia.org/wiki/Brainfuck. Accessed: 2026-05-27.

[11] Wikipedia: C++ programming language. https://en.wikipedia.org/wiki/C%2B%2B. Accessed: 2026-05-27.

[12] Wikipedia: C programming language. https://en.wikipedia.org/wiki/C_(programming_language). Accessed: 2026-05-27.

[13] Wikipedia: Compiler. https://en.wikipedia.org/wiki/Compiler. Accessed: 2026-05-27.

[14] Wikipedia: Complex numbers. https://en.wikipedia.org/wiki/Complex_number. Accessed: 2026-05-27.

[15] Wikipedia: Control flow. https://en.wikipedia.org/wiki/Control_flow. Accessed: 2026-05-27.

[16] Wikipedia: Formal language. https://en.wikipedia.org/wiki/Formal_language#Programming_languages. Accessed: 2026-05-26.

[17] Wikipedia: Instruction set architecture. https://en.wikipedia.org/wiki/Instruction_set_architecture. Accessed: 2026-05-27.

[18] Wikipedia: Interpreter. https://en.wikipedia.org/wiki/Interpreter_(computing). Accessed: 2026-05-27.

[19] Wikipedia: Lexical analysis. https://en.wikipedia.org/wiki/Lexical_analysis. Accessed: 2026-05-27.

[20] Wikipedia: List of c-family programming languages. https://en.wikipedia.org/wiki/List_of_C-family_programming_languages. Accessed: 2026-05-27.

[21] Wikipedia: Machine code. https://en.wikipedia.org/wiki/Machine_code. Accessed: 2026-05-27.

[22] Wikipedia: Matrix (mathematics). https://en.wikipedia.org/wiki/Matrix_(mathematics). Accessed: 2026-05-27.

[23] Wikipedia: Operator. https://en.wikipedia.org/wiki/Operator_(computer_programming). Accessed: 2026-05-27.

[24] Wikipedia: Quaternions. https://en.wikipedia.org/wiki/Quaternion. Accessed: 2026-05-27.

[25] Wikipedia: Rational numbers. https://en.wikipedia.org/wiki/Rational_number. Accessed: 2026-05-27.

[26] Wikipedia: Reference mark. https://en.wikipedia.org/wiki/Reference_mark. Accessed: 2026-05-28.

[27] Wikipedia: Set-builder notation. https://en.wikipedia.org/wiki/Set-builder_notation. Accessed: 2026-05-27.

[28] Wikipedia: Type signature. https://en.wikipedia.org/wiki/Type_signature. Accessed: 2026-05-29.

[29] Wikipedia: Unicode. https://en.wikipedia.org/wiki/Unicode. Accessed: 2026-05-27.

[30] Wikipedia: Von neumann architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture. Accessed: 2026-05-27.