Arrcy: A Productive, Functional, and Performant Array Processing Language

Rudy Soliz

October 16, 2024

Abstract

Arrcy is a functional array processing language designed with simplicity, performance, and developer productivity in mind. This paper presents the design and implementation of Arrcy, which offers a succinct syntax for functional operations, such as mapping, filtering, and reducing arrays. With a focus on branchless and procedural processing, Arrcy enables straightforward optimization and ease of use. An example implementation, written in C++, demonstrates that Arrcy can achieve performance comparable to low-level alternatives. Benchmarks comparing Arrcy to C++, Python, JavaScript, and C# reveal that Arrcy delivers similar performance to C++ while maintaining a simpler syntax. The language's simplicity and performance potential make it a compelling tool for array-heavy computational tasks.

1 Introduction

New programming languages, especially those made by newcomers to language design, are important for innovation in software development and computational theory. Arrcy, is an insightful exploration into what a new functional array processing language could look like in an age where speed of execution and speed to develop is of paramount importance. By providing a succinct and expressive syntax for functional operations, Arrcy simplifies complex computations while maintaining clarity. Its procedural, branchless design enables straightforward optimizations, making it adaptable for future advancements. The example implementation of Arrcy has been written in C++ matches the performance of low-level alternatives.

2 Design

Arrcy is a functional, multi-dimensional array processing language with a focus on user productivity. Arrcy's design is inspired by other traditional languages such as APL [Ive62], and adopts certain array functions inspired from languages such as Haskell [HHJW07] or JavaScript (cira ES5) [ECM09]. Arrcy is a compiled language, aiming to achieve C/C++ performance on simple array programs.

Arrcy provides in-built operators for array processing, the ability to compute multiple outputs and print to the console. However, it does not allow user-defined functions and currently, Arrcy only allows statically-sized arrays containing values of type doubles.

Syntax: Arrcy's syntax is designed to be written quickly and familiar to programmers of C-like languages. A program consists of a semicolon-separated list of statements. A statement can be a declaration, assignment, a for-each loop, or a built-in function call (`print`). A declaration or assignment looks the same: an identifier followed by an equal sign, then an expression. Expressions can be arrays, literals, binary operations (+, -, /, *), or identifiers.

```
number = 5;
array = [1, 2, 3, 4, 5];
nested_array = [[1, 2], [3, 4]];
```

The types in Arrcy are inferred and static, which means you don't need to specify the type, however, the language will enforce that these types do not change during the pre-processing phase.

Functional Operators: In general, all functions in Arrcy will follow a similar structure. You first encapsulate your array inside of parenthesis, then you will place the symbol that denotes the desired function, then you will name your two variables, and finally, it will be closed off with a set of curly braces that contains your statement or expression.

To start, let's begin with the most simple `forEach`. This is for iterating with side effects. While there is no mechanism in the language to prevent side effects in the other functions, this is specifically designed for this use case and is the only statement function.

```
(array)*item, idx {
    // statements
    print(item);
};
```

In the above example, the `forEach` loop is declared for the array and then each item in the array is printed to the console.

The `map` function iterates through elements of an array and executes one expression on each element. This then creates a new list with the results of the expressions.

The `filter` function iterates through elements of an array and executes an expression on each element. If the result of the expression is 0, then the element is removed from the array and the new array is returned.

The `reduce` function iterates through elements of an array and is given an accumulator. The final value of the accumulator is returned as the result. The result, at this time, can only be a number.

Standard Functions: These functions utilize specific compiler features, and may not be the same across all implementations. While defining functions is not a feature of Arrcy, the specific compiler implementation may provide external functions that may be called at runtime. The syntax for calling the aforementioned functions

3 Code Examples

4 Implementation

There is a nearly fully implemented compiler for Arrcy, with the host language being C++. There are 4 major components to the compiler, consisting of a Lexer, Parser, Type Checker, and Code Generator. This is an intentionally low optimization implementation due to time constraints. It also is not fully featured to the design, and where it falls short will be mentioned specifically.

Lexer: A lexer is a pattern-matching machine, that determines which character sequences make valid words in Arrcy. The lexer is written in flex [Pax07]. The lexer identifies a limited number of tokens, these being: numbers, identifiers, operations, brackets, and parentheses. The syntax is similar to a Regular Expression (RegEx) [Fri06]. After the pattern matching, curly braces will contain the C++ code to be executed when a given pattern has been matched. The full string will be placed in the `yytext` variable. Groups are defined in square brackets and ascii characters may be placed inside of these square brackets. Placing a dash in between these characters will match any characters from the start of the ASCII sequence until the end. Then adding a plus sign will match the previous group at least one time. A full example will be the numerical parsing for Arrcy, which is as follows:

```
[0-9]+ {
    yylval.float_val = atof(yytext);
    return NUMBER;
}
```

This pattern will parse an integer, and return a `NUMBER` token. All tokens may optionally specify an associated type, which will be present on the `yyval` variable, this we call `float_value` which is set to be equivalent to the input value. To achieve this, it will first locate a character sequence that has a character between '0' (48) and '9' (57), which includes all numbers.

Parser: The parser's job is quite similar to the Lexer's. Where the lexer would find words, the parser's job is to determine which groups valid tokens make valid sentences in Arrcy to then be assembled into a tree, which links them together. The parser used was bison [DS21], an open source implementation of yacc [Joh75]. These groups of tokens are called nodes, which themselves represent logical concepts within the language. Any nodes are derived from either StatementNode or ExpressionNode, which will determine how references can be stored to any single AST Node. Statements may stand on their own, meanwhile expressions must be contained by a statement. Nodes are then oriented into a tree, with a starting root node with a list of statements. These statements may contain things such as assignments, functions, or calls. The outermost statement node is a CodeBlockNode, which is a statement that contains several other statements inside of it. Code blocks are executed sequentially, from top to bottom. Statements can be assignments, reassignments, external function calls, or a forEach functional operation.

The Type Checker: The type checker is a comparatively smaller step than the parser or lexer. Its job is to walk the AST, and get an idea of how the program is structured, and determine the meaning throughout the statements of the program. It builds a symbol tree, and checks for any undefined variables being used, as well as ensures that arrays of different sizes are assigned to eachother. It maintains a strict typing system.

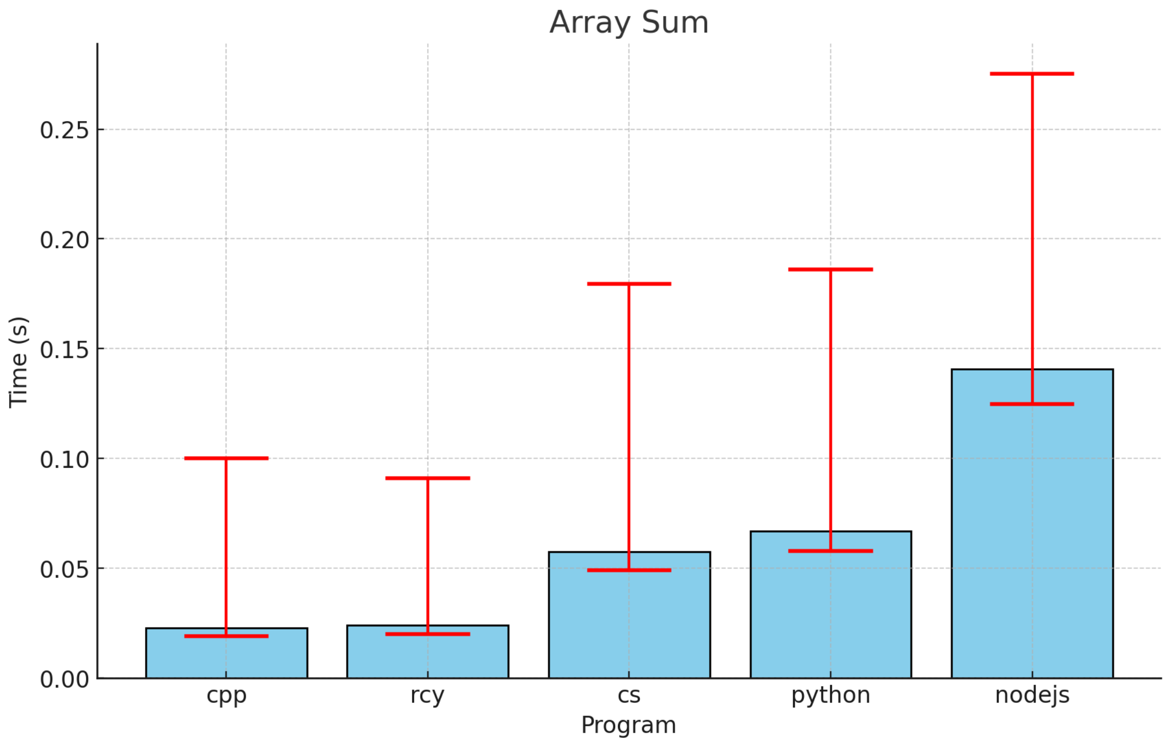
Code Generation: The Code Generator is created in an abstract approach, allowing for transpilation to any language. For this paper, I have chosen to transpile to C++ for the benefits of speed and ease of compilation to machine code. The code generator will first check all assignments, and assign them to their initial values at the beginning of the program. Then it walks statement by statement and generates C++ code to be compiled. This is a naive approach, as minimal steps have been taken to optimize the code. This is a topic to be explored further, as this language develops. After the C++ code is generated, it can be compiled by any modern C++ compiler. For my implementation, I used the GNU C++ compiler (G++) [Fou24a]. The code generator does not currently support reduction, or filter actions.

5 Evaluation

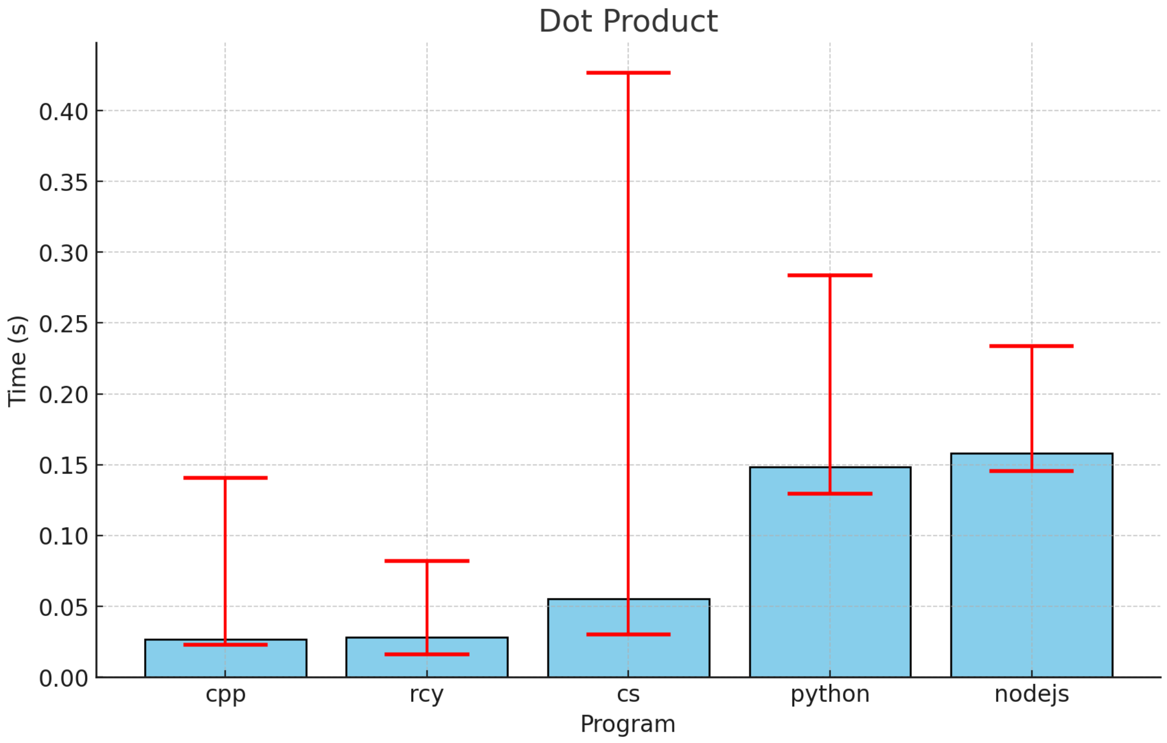
Benchmarks are an important part of comparing languages, as its one of the few purely quantitative ways of measuring qualities of a language. All languages will attempt to run a very similar program that produces the same result. The programs will each be written by me, Rudy Soliz, in the most straightforward and analogous way that can be written for each language. This being said, not all implementations will be perfectly optimal, and future benchmarking may need to be done, however it is sufficient for the purposes of this paper. The benchmarking was done using a python script which starts a timer, runs an instance of the application, and then stops the timer after the application has closed. It will do this 3,000 times for each language benchmarked. The languages that Arrcy will be benchmarked against are the following: python [Fou24c], javascript [Fou24b], C# [Mic24], and C++ [Fou24a]. The expectation, before seeing any of the data is that C++ will have a similar or equal performance to Arrcy. Python [Fou24c] and Javascript are both interpreted, and will more than likely take longer. C# [Mic24] is JIT compiled, so it should place 3rd on nearly all metrics. The code for each of these is here.

Clarifications: The Python interpreter of choice is CPython [Fou24c]. The javascript runtime of choice is NodeJS [Fou24b]. Arcy and C++ will both be compiled using G++, with O2 optimizations enabled.
The computer that will perform the benchmarks is a Framework 13 running Windows 10, with 16GB of RAM and an AMD Ryzen 5 7640U.

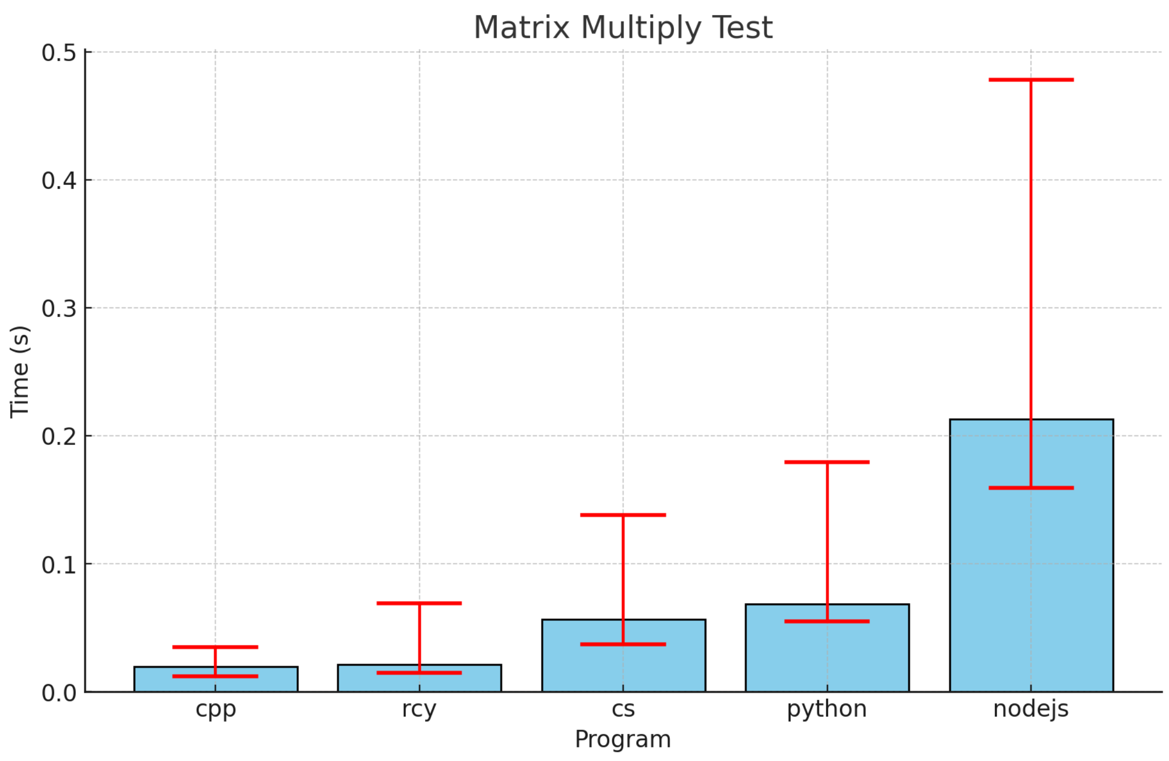
The Data:



This first test was by far the most simple. All this code does, is create an array from 1-100 and adds up all of the values. Then it prints the final value. One thing of note here is that javascript, python and even C# (but to a lesser degree) will always have a large amount of overhead that could never be present in a truly compiled language. So these will always be higher values. Arrcy's variance is comparible to c++ and has almost the same average time, which will show to be a pattern.



This test involves calculating the dot product of two 100k arrays. then it prints the result. Here, Arrcy and c++ have only 0.001 difference on average.



The matrix multiplication test was on a 10x10 matrix of increasing values. Very similar results to previous graphs.

6 Lessons Learned

As the author of Arrcy, I had not previously utilized a parser generator, this project was a valuable learning experience for me. Moving forward, I will be using parser generators similar to YACC [Joh75]. Manual parsing is much more time consuming and tends to yield results either equivalent or worse to results provided by a parser generator. Arrcy represents the most complete language I have developed, and I am much more knowledgeable in language architecture. An inheritance-based approach to nodes has proven to be effective. In contrast, I found the visitor pattern to be particularly challenging in parsing state, and often led to unintended side effects and may cumbersome code structures.

As the project came into its later states, it came across many issues. There became a heavy reliance on error flags, which eventually became very problematic. It may result in continued execution even after a failure state. Additionally, the code includes an excessive number of monolithic functions, ultimately resulting in high coupling with all members being public.

While this language achieves satisfactory results given its current state and the fact it was developed in just a few months. This language would benefit from optimizations such as SIMD [Cor19] and multi-threaded processing [Wil19] or even a move to the GPU like Open Computing Language (opencl) [AM11]. One aspect i I am pleased with, and wouldn't be changed is the language's design; the syntax is efficient and generally quicker to write compared to the benchmark languages I used in the evaluation section. Overall, this was a wonderful learning experience, and in the future, the implementation of my programming languages will be better organized and be even more efficient.

References

[AM11] Timothy G. Mattson James Fung Dan Ginsburg Aaftab Munshi, Benedict Gaster. *OpenCL Programming Guide*. Addison-Wesley, 2011.

[Cor19] Intel Corporation. Simd programming techniques for c++. 2019.

[DS21] Charles Donnelly and Richard Stallman. *Bison: The GNU Parser Generator*, 2021.

[ECM09] ECMA International. ECMAScript Language Specification, 5th Edition. https://www.ecma-international.org/ecma-262/5.1/, 2009. Accessed: 2024-10-15.

[Fou24a] Free Software Foundation. *GCC: The GNU Compiler Collection*, 2024.

[Fou24b] OpenJS Foundation. *Node.js Documentation*, 2024.

[Fou24c] Python Software Foundation. *CPython: The Python Interpreter*, 2024.

[Fri06] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly Media, 3rd edition, 2006.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. ACM, 2007.

[Ive62] Kenneth E. Iverson. A programming language. *10.1145/1460833.1460872*, page 345–351, 1962.

[Joh75] Stephen C. Johnson. *YACC: Yet Another Compiler-Compiler*. Bell Laboratories, 1975.

[Mic24] Microsoft. *.NET Framework Documentation*, 2024.

[Pax07] Vern Paxson. *Flex: The Fast Lexical Analyzer*, 2007.

[Wil19] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 2019.