Optimizing Brainfuck-to-Python Transpilation: A Semantic vs. Syntactic Analysis Approach

02242 Program Analysis

Jakub Oszczak
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
s233577@student.dtu.dk

Krzysztof Piotr Sawicki
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
s233667@student.dtu.dk

Muse Ali
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
194615@student.dtu.dk

Peter Zajac
DTU Compute
Technical University of Denmark
Kgs. Lyngby, Denmark
s233852@student.dtu.dk

***Abstract*— We propose a project in which we will explore two methods of optimizing brainfuck code. The first method involves detecting optimizable patterns based on the semantics of Brainfuck and second method relies on syntactic analysis. We will evaluate the methods by comparing wall-clock runtime of transpiled brainfuck to python code.**

***Index Terms*—Program analysis, Brianfuck, Transpilers, Semantic analysis, Syntactic analysis**

I. Introduction

As our project we chose to research two ways of optimizing brainfuck code. One approach involves detecting patterns using the semantics of Brainfuck, which requires executing the code and looking for optimizable segments that can not be noticed without execution. The second approach is using syntactic analysis, which entails recognizing optimizable patterns visible in code without executing the code.
Brainfuck is an esoteric programming language that consists of only eight simple commands, while still being fully Turing complete. A Brainfuck program is a sequence of characters (8 characters available) that is basically a sequence of commands, which the code interpreter goes through and executes. Brainfuck programming language consists of a memory of 30,000 byte cells, movable data pointer and two byte streams, one for input and one for output [1]. Available commands (characters) are listed below in Table I.

TABLE I
BRAINFUCK COMMANDS [1]

Character	Meaning
>	Increment the data pointer by one (point to the next cell).
<	Decrement the data pointer by one (point to the previous cell).
+	Increment the byte at the data pointer by one.
-	Decrement the byte at the data pointer by one.
.	Output the byte at the data pointer.
,	Accept one byte of input, storing its value in the byte at the data pointer.
[	If the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.
]	If the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

A. Motivation

As you might already have realised, Brainfuck's minimalistic command set makes the language complex and hard to write by design - even for the most seasoned programmers. Now imagine using this unique, simple - yet Turing complete - programming language to write programs longer than the 15 character example seen in listing 1. Your primary goal would likely just be to get the program to work, but why set the bar to just having code that works? Why not aim to optimize it? Furthermore, why not optimize it using program analysis techniques to help you along finding optimizable pattern in Brainfuck code? That is our underlying motivation for this project where we take on two analysis techniques to determine and compare their efficiency in optimizing Brainfuck programs in terms of wall-clock run-time.

B. Proposal

As previously mentioned, we propose a comparison of semantic and syntactic analysis for optimization of Brainfuck code transpiled to python. This comparison has led us to the research question:

1) Problem Statement: Can detecting patterns using semantics of Brainfuck be more efficient than syntactic analysis in terms of performance measured by wall-clock run-time?

2) List of Contribution: In the project, we aim to touch upon the following:

1) Developing a Brainfuck interpreter in Python
2) Developing a Brainfuck to Python transpiler
3) Optimizing transpiled Brainfuck code using semantic and syntactic analysis approaches
4) Evaluating the effiency of optimization through semantic and syntactic analysis respectively by comparing wall-clock run-time.
5) Discuss alternative approaches to Brainfuck optimization

We will have to build an interpreter and transpiler. The way of building interpreter is to read through the code character by character and with conditional statements execute appropriate commands to move data pointer, change bytes, take input, print output and handle loops. As for the transpiler it is quite similar but we must also generate the python code.
In theoretical part of the paper we will write out the operational semantics equations that describe all the operations that our interpreter is capable of executing.
When it comes to optimization and analysis of the Brainfuck code we will utilize approach that uses intermediary representation of code. That is, we will use our source-to-source compiler (transpiler), which enables us to implement various optimization techniques that would not be accessible while working only on Brainfuck code. Some examples of optimization methods we will use in syntactic analysis include [3]:

- Contraction - condense the same operations into one, eg. series of "+" combined into add(x) where "x" is number of "+".
- Removing snippets of code that cancel out eg. "+ + - - ><"
- Removing empty loops

For semantic analysis we will mainly focus on optimizing specific loops like $[++-]$ which basically zeros out the current memory position.

C. List of Contributions

In the project, we aim to touch upon the following:

1) Developing a Brainfuck optimizer using syntactic analysis techniques. See Section II-B for high level theory and Section III-A for the implementation overview.
2) Developing a Brainfuck optimizer using semantic analysis techniques. See Section II-C for high level theory and Section III-B for the implementation overview.
3) Comparing transpiled Brainfuck to Python code in Section IV-C.
4) Evaluating the efficiency of the comparison of semantic and syntactic analyses by comparing wall-clock run-time. Note Section IV-E.
5) Discuss alternative approaches and techniques to optimize Brainfuck. See Section V

II. High level description

A. Operational semantics

Operational semantics is a formal way of describing the behavior of a programming language in terms of the computation steps taken on its programs. In the case of Brainfuck, we can describe its operational semantics using transition rules for each of its commands. These rules specify how the state of the execution changes with each command. The state can be represented as a tuple containing the program (a sequence of Brainfuck commands), the pointer position, and the tape (an array of memory cells). Let's denote:

- P as the program (a sequence of Brainfuck commands),
- i as the pointer position,
- T as the tape (an array of memory cells),
- T[i] as the value at the current cell,
- P[j] as the j-th command in the program.

Here are the operational semantics for Brainfuck:

a) Increment the Pointer ('>') Command:

$$(P[j] = '>', T, i) \rightarrow (P, T, i + 1)$$

b) Decrement the Pointer ('<') Command:

$$(P[j] = '<', T, i) \rightarrow (P, T, i - 1)$$

c) Increment the value at the Pointer ('+') Command:

$$(P[j] = '+', T, i) \rightarrow (P, T[i] = (T[i] + 1) \mod 256, i)$$

d) Decrement the value at the Pointer ('−') Command:

$$(P[j] = '-', T, i) \rightarrow (P, T[i] = (T[i] - 1) \mod 256, i)$$

e) Output the value at the Pointer ('.') Command:

- This command involves an external effect (output) and does not change the state.

$$(P[j] = '<', T, i) \rightarrow (P, T, i)$$

f) Take single input value (x) and Store it in the Pointer (',') Command:

$$(P[j] = ',', T, i) \rightarrow (P, T[i] = x, i)$$

g) Jump Forward if the value at the Pointer is Zero ('[') Command:

$$(P[j] = '[', T, i, T[i] = 0) \rightarrow (P, T, k)$$

h) Jump Backward to the Matching '[' if the value at the Pointer is Non-Zero (']') Command:

$$(P[j] = ']', T, i, T[i] \neq 0) \rightarrow (P, T, k)$$

These rules define the state changes in the Brainfuck program's execution. They are simplified and assume that the tape is unbounded and that the program does not attempt to move the pointer to a negative position. In practice, implementations may have limitations on tape size and behavior at the boundaries.

B. Syntactic analysis

Syntactic analysis, commonly known as parsing, is a fundamental aspect of compiler design and language processing. It involves analyzing a sequence of symbols in programming languages to understand its grammatical structure according to the rules of a formal grammar. Syntactic analysis is geared toward understanding command order and grouping. The approach to syntactic analysis can vary significantly between programming paradigms. In our case brainfuck is an imperative language so the focus is on the sequence of commands and control structures.

Our methodology involves constructing an **Abstract Syntax Tree (AST)** from Brainfuck code, followed by a series of **optimization steps** that are applied to this AST before **generating the equivalent Python code**.

a) Construction of the Abstract Syntax Tree (AST): The first step in our process is the conversion of Brainfuck code into an AST. This tree structure represents the hierarchical syntactic arrangement of the Brainfuck commands. Each node in the AST corresponds to a distinct command or a control structure in the source code. The hierarchical nature of the AST mirrors the nested structure of Brainfuck loops and sequences of operations, allowing for the easy identification of patterns and sequences for optimization. Since brainfuck is quite simple language with very limited set of operations, using AST is not really necessary, but this is a good practice while doing transpilers. This representation provides a structured and easily navigable format of the code, setting the stage for subsequent optimization processes.
Let's consider following simple brainfuck program.

```
++[+[->+<]]--
```

Listing 1.]Example of simple brainfuck program]

Below is the unoptimized Abstract Syntax Tree of the above code:

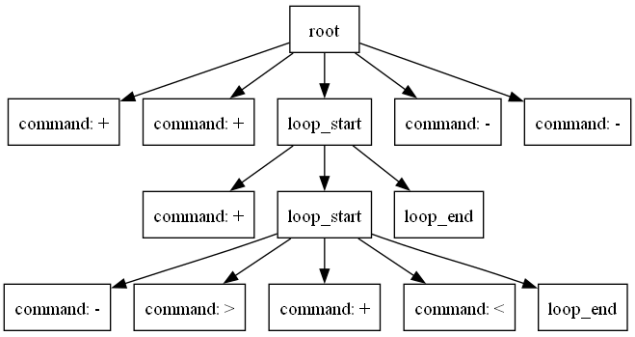

Fig. 1. Unoptimized version of AST for the simple brainfuck program $(++[+[->+<]]--)$.

And here is the AST optimized by compressing "+" and "-" characters, and handling so called "copy loops" $([->+<])$:

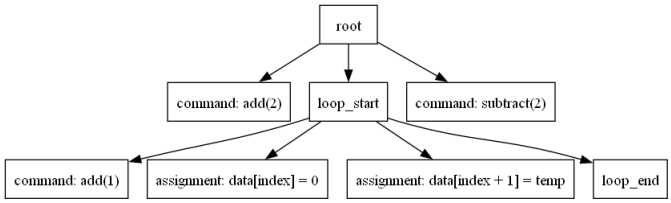

Fig. 2. Optimized version of AST for the simple brainfuck program $(++[+[->+<]]--)$.

b) Implemented optimizations: Second step is to apply optimizations to the Abstract Syntax Tree. Those are optimization that look for patterns in program's syntax. We have applied following optimization techniques:

- optimize_arithmetic Brainfuck programs frequently contain extended runs of the symbols + or -. In a basic translation approach, each of these Brainfuck commands corresponds to an individual line of Python code. As an example, this snippet of Brainfuck code: $++++>++++>----$ could concatenate each appearance of repeating "+" and "-" sign and in the end just be add(3) right(1) add(4) right(1) sub(4). Our program instead of 14 commands, now only has 5. This behavior is very popular in brainfuck programs, sometimes repeating the same character tens of times because sometimes the program needs to go through many characters to find it's ASCII representation.
- optimize_pointer This is the same case as in the previous optimization technique, this time focusing on repeating $<$ or $>$ characters. The amount of optimization that we can achieve by concatenating $<$ and $>$ characters depends on the way the initial program was written.
- optimize_consecutive_loops This optimization method removes all but the first loop in sequence of consecutive loops in brainfuck code. This is effective because once the pointer exits the first loop, its value is zero, causing it to bypass subsequent loops. We didn't apply this technique to our test programs, as professionally written brainfuck code rarely presents this scenario. However, it's a valuable optimization for self written brainfuck programs seeking efficiency.
- optimize_clear_loops In Brainfuck, the common practice to make sure that the current pointers value is zero is to implement a clearing loop. It is a loop $[-]$ that decreases the current cell's value by one, until the cell's value becomes 0. The potential for optimization is as high as the maximum value that the cell can hold, so potentially 255 operations can be concatenated to just one.
- delete_first_loop Every brainfuck program start with a tape of 30000 characters, all set to zero. If the first operation that our program tries to execute is the beginning of the loop, it will simply bypass it. It is a similar case as for the optimize_consecutive_loops, so it will also not be useful in professionally written benchmarks for brainfuck language, but it might be useful for optimizing our own programs.

- remove_redundant_sequences The ',' character in Brainfuck acts as an input for our program. It overrides the value at the pointer's current position with an ASCII value equivalent to the character provided by the user. Since the input in Brainfuck overrides the current cell's value, every '+' and '-' character preceding the input can be deleted, as the cell's value will be overwritten either way. As the previous case, this shouldn't take place in a professional brainfuck program, so there is no use in testing it on our benchmarks.
- copy_loop_optimization A very widely used concept in brainfuck is a copy loop. Let's look at this code sniipet as an example: $[->+<]$. It subtracts one from the current cell, while simultaneously adding one to the next cell. This operation will repeat until the value of the first cell reaches zero. As a result, the next cells value will increase by the same amount, that the first decreased. Since this is a popular method used in brainfuck programs, we can show significan optimization potential on our benchmarks.
- all_optimization This section combines all previously discussed optimization techniques for Brainfuck programs. By integrating them, we aim to achieve the maximum possible efficiency in Brainfuck code execution that syntactic analysis approach allows us to.Combining those optimizations is especially beneficial in complex Brainfuck programs, where they work together to reduce runtime.

c) Generation of Python Code: The final stage of our approach is the generation of Python code from the optimized AST. This step involves traversing the AST and translating each node into its Python equivalent. The optimizations applied in earlier stages significantly reduce the complexity and improve the efficiency of the resultant Python code.

d) Theoretical guarantees: Based on selected testing we can conclude that our program is sound since it's not making any mistakes. Every applied optimization is guaranteed to be applied in a proper fashion, not causing any bugs. However the code is not complete because it is not able to detect every possible syntactic optimization that could have been applied, and some methods of optimization are narrowed down, because it would be very complex to code them in their broadest scope.

C. Semantic Analysis

For the comparison to syntactic analysis, we chose to combine multiple techniques, the most prominent is dynamic analysis, but we also used a symbolic state representation and partial execution for complex cases.

1) Theoretical Background:

a) Dynamic Analysis: Involves examining the characteristics of a program in its active state, as opposed to static analysis, which reviews the program's code to deduce properties applicable to all its runs. In dynamic analysis, properties relevant to certain runs are identified by observing the program in action. While this method can't conclusively verify if a program meets a specific property, it is useful in spotting property breaches and offering insights into how the program behaves when it's running, which made it a sensible choice for our project.

b) Symbolic Execution: Symbolic execution is a method for analyzing sequential programs by running them with symbolic values rather than concrete. This approach enables the exploration of different potential paths and outcomes in a program, based on varying inputs. It is particularly effective for us when dealing with user inputs. [4]

c) Partial Execution: This method consists of analyzing parts of a program's code without executing it. It's more about understanding the code's behavior, structure, or properties through static analysis methods. This can involve examining control flow, data flow, dependencies, or other aspects of the code. In our case, due to inability to explore static analysis techniques, we skip complex states involving symbolic expressions. [4]

2) The Tool Overview: The 'BrainfuckSolver' class integrates concrete and symbolic execution methodologies to analyze and transpile Brainfuck programs into Python.

Key Components and Functionality

- *Tape and Pointer Management*
- *State Snapshot*
- *Loop Management*
- *Optimization and Execution*
- *Python Transpilation*

3) Data Structures and Flow: Mimicking Brainfuck's memory mode, the tool uses a fixed-size list('tape') with size range from 0-255 bits. The array can store both integer (concrete) and symbolic values. A pointer, simulating the BF instruction pointer, navigates the tape, changing the tape based on commands. A list was also used to store the last state to prevent duplicate transpilation. The flow navigation via 'preprocess_loops' is essential for mapping the loops.

4) Theoretical Walk-through: This tool combines actual (concrete) execution, symbolic values and partial execution. When dealing with simple scenarios e.g. basic commands $+, -, <, >$, the code is executed concretely and saved into the tape. When dealing with the input ',' command, a symbolic value is saved into the tape, representing the possible ASCIIref input from the user. The SymbolicValue class utilizes the SymPy library [5] for symbolic representation.

optimization strategies At the core of the optimization we used dynamic analysis, which executes the program character by character, saving the state of the tape until it has to be transpiled, later refered to as the 'forcing condition'. These include:

- **Efficient Value Manipulation:** Instead of incrementing and decrementing once for every '+' and '-' sign, the tool assigns the final value until a forcing condition happens.
- **Pointer Movement:** In Brainfuck, frequent pointer movements can be costlyref, so similarly to value manipulation, all movements are saved in the tape until the forcing condition.
- **Optimized Loops:** When dealing with exit conditions that do not depend on symbolic values, we use the

mapped loops to check if the loop doesn't have a null value on its exit conditions, removing unnecessary loops. Furthermore the loops are dynamically handled, basically them to just setting values reducing the number of operations in the transpiled code.

This forcing condition can happen in a few scenarios. Firstly, when the print character '.' is encountered, this way the values need to be set or input before they can be printed. Secondly, when a loop exit function depends on a symbolic variable. The problem here is that the state after the loop is also unknown, therefore any further code is not optimized rather just transpiled. Lastly, the end of the program, where all optimizations are applied. Overall, a cutoff condition is also set, where we don't consider anything taking longer than 1 hour for the analysis and optimization. Dynamic analysis can be rather costly especially with many nested loops.

Brainfuck to Python Transpilation The tool also includes transpilation to Python which serves two main purposes. Debugging and further optimization of the execution. The optimization are applied during this phase, resulting in a more efficient code.

D. Theoretical Guarantees

Guarantee of Correctness: Our approach makes sure that the optimized code preserves the semantic invariants of the original Brainfuck program. This means that the behavior of the program post-optimization remains functionally equivalent to its behavior pre-optimization. Furthermore, the use of SymbolicValue ensures that symbolic representations accurately reflect the constraints concrete values, this means that most logical paths are considered and maintained throughout the execution.

The code for the semantic analysis optimizations is sound according to the testing done, but is not complete because not handling the symbolic exit conditions, means that completeness was not achieved but perhaps could be with future work.

Efficiency and Performance: Using the grouping of consecutive commands and efficient loop handling our methods guarantee a reduction in the computational complexity, which enhances performance. Additionally, by efficiently managing symbolic and concrete values we improve the memory management which is crucial in Brainfuck code.

III. Implementation

A. Syntactic analysis

1) Code: Our program is designed to transpile Brainfuck code into Python by utilizing an Abstract Syntax Tree (AST) for optimization and translation. The process begins with the initiation of the ASTtranspiler, which is the primary driver of the program. Upon execution, the ASTtranspiler triggers the ASTgenerator, a crucial component that parses the input Brainfuck code and constructs an AST representation of the program. The Node class, the building block of the AST, is structured as follows:

```python
class Node:
    def __init__(self, kind, value=None):
        self.kind = kind
        self.value = value
        self.children = []

    def add_child(self, child):
        self.children.append(child)
```

Listing 2. Node class definition

Each `Node` object has a `kind` attribute indicating the type of syntactic element it represents (e.g., command, loop start, loop end), an optional `value` that stores the specific Brainfuck command, and a `children` list that holds any nested syntactic elements, maintaining the tree structure.

Following the AST generation, the program enters a decision point where it determines whether to apply certain optimizations. If optimizations are selected, the program processes the AST to optimize it, enhancing efficiency and execution speed of the resulting Python code. These optimizations can include consolidating consecutive operations, eliminating dead code, and transforming loops into more efficient constructs.

Once optimized, the AST is transpiled into Python code. This transpilation step involves traversing the AST and converting each node into the equivalent Python instruction, taking into account any optimizations previously applied. The final output is a Python code file that mirrors the functionality of the original Brainfuck code but with the benefits of the optimizations applied during the transpilation process.

The process described above can be visualised by following program flow chart:

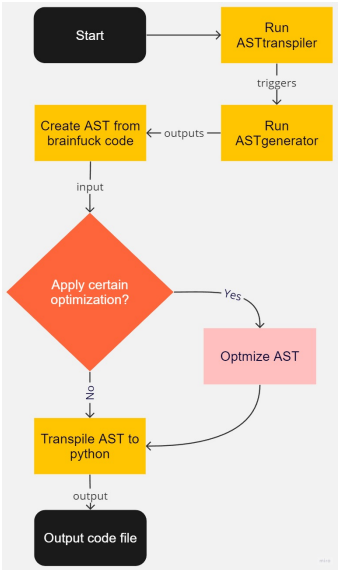

Fig. 3. Program flow chart

B. Semantic Analysis

1) High Level Overview: : This tool is designed to perform dynamic analysis, execute the code concretely, until it hits an input which gets saved as a symbolic value using the

'SymbolicValue' class (see Listing source code). Whenever a symbolic value appears on the place of a loop exit condition, optimizations are applied and the rest of the code, including the loop itself is transpiled without further optimizing it as mentioned in the previous section.

2) Usage: The user starts by instantiating a new *BrainfuckSymbolicSolver* instance, when a tape representing the Brainfuck memory is created as an array of zeroes. After that the member method *optimize and convert to python'* is called, where the user needs to specify the Brainfuck source code file destination and has the option to specify the output Python file name, or it defaults to *test.py*. After that the analysis is performed in the following manner. The *optimize* method is called, where the Brainfuck code is iterated over character (command) by character. With the four basic commands +, -, $<$ and $>$, the code is executed and integers are saved to the tape. Whenever an input command is encountered (,), a symbolic value is instantiated to the tape. The four basic commands are performed normally on the symbolic values. When a print command is encountered (.) only the printed value gets transpiled, and the changed cell is also saved to the *last state* variable, to prevent duplicate transpilation. If, however, a symbolic value is encounter in a loop exit condition, i.e. at the start or end of the loop, Found optimizations are applied and the rest of the code is transpiled to python without further optimizations. Finally, after the code ends, all cells that were not printed are transpiled. The flow is visualized on Figure 4. After that the executable optimized code is generated as a python file with the correct indentation.

3) Performance Remarks: : Due to the nature of loop optimization in this tool, programs with many nested loops take a long time to be analyzed, however as will be seen in the results the simple commands after optimization could offset this. However, the cutoff for the tool was set for 1 hour.

The source code can be found in GitHub repository: https://github.com/matrxi999/Program_analysis_brainfuck.

IV. Evaluation

In this section, we will focus on the evaluation of our research question: *Can semantic analysis of Brainfuck be more efficient than detecting patterns using syntactic analysis in terms of performance measured by wall-clock runtime?*
The chosen benchmarks specified in Subsection IV-A were taken from the internet, which are the closest we could get to industry use-cases for brainfuck. These range from 'Hello World' to Sierpinski's triangle. We will try to answer the following sub questions stemming from our research question:

- **How fast is syntactically optimized brainfuck code?** Here we will take measurements of the syntactically optimized code focusing on wall-clock runtime. The set of brainfuck programs will range from simple to more complex, ignoring inputs.
- **How fast is semantically optimized brainfuck code?** We repeat the process to gather data for the semantically optimized code.

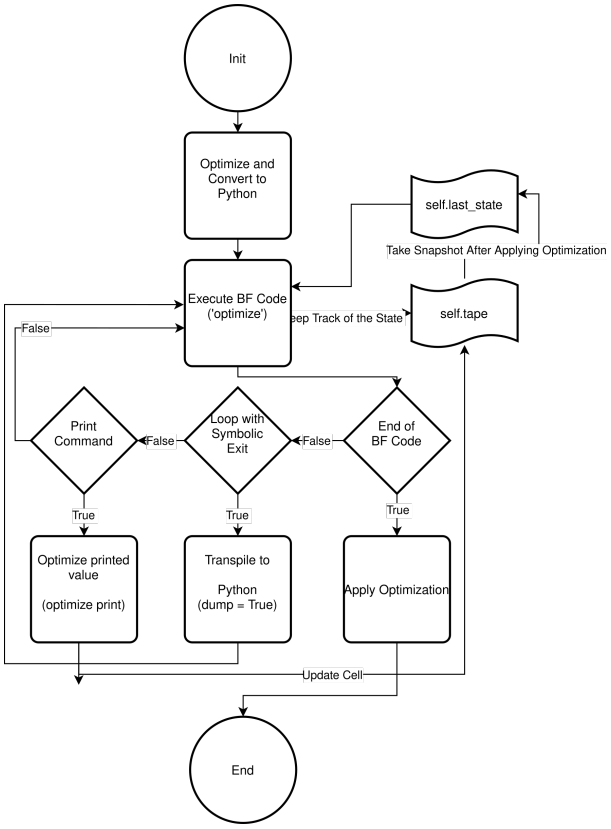

Fig. 4. Flow Chart - Semantic Analysis Tool

After answering these sub question, we can try to find a good enough answer by using statistical analysis on the gathered data seen in Section IV-C, using one-tailed t-test, to the hypothesis expressed in our research question.

A. Benchmark Suite

To effectively evaluate the performance of Brainfuck code that has been transpiled into Python, it is essential that we select Brainfuck programs that are sufficiently complex and different from each other to benefit from our syntactic and semantic optimization techniques. So, if we were use Brainfuck program that are too simplistic or too similar, they would be unsuitable to serve as valid test cases.
Therefore, for this evaluation we will focus on five carefully chosen Brainfuck programs that vary in complexity. These examples will serve as test cases for evaluating the impact of our different optimization techniques on the performance of the transpiled code. The examples chosen are namely,

- triangle.b (a short program for printing sierpinski triangle out of stars)
- mandlebrot.b (a mandlebrot set visualization using ASCII)
- yapi.b (pi digits calculator - we used first 30 digits of pi)
- helloword.b (a simple program that outputs "Hello, World!")
- squares.b (Program that outputs square numbers from 0 to 10000)

As you see the nature of the programs apart from helloworld.b are mathematical computations and visualizations of varying complexity. If we dive deeper into the characteristics of the chosen test files as shown in table II, it is evident the these program vary in complexity in terms of number of characters, loops and nested loops as well as their maximum nesting level for loops.

triangle.b has a moderate complexity mainly due to its nested loops. Mandelbrot.b is significantly more complex and also the most complex program in our analysis. Mandelbrot.b's complexity stems the overall length of the program, 11518 characters, and its high count of loops, 686 - out of these 677 are nested loops! Yet, mandelbrot.b is overcome by yapi.b, when looking at max nesting level, as it as a maximum nesting level of 11, while having 54 nested loops out of 60 loops. As in any language "Hello, World!" is the go-to starting point and in our test cases helloworld.b serves as an alternative to the other test cases focusing on mathematical computations and visualizations. With it having only 164 characters and 5 loops, with no nested loops, it is the simplest test case we selected. Our last test case is squares.b which has a moderate complexity that stems form nested loops.

Name	Character count	Loop count	Nested loop count	Max nesting level
triangle.b	1456	17	14	4
mandlebrot.b	11518	686	677	9
yapi.b	652	60	54	11
helloword.b	164	5	0	1
squares.b	203	20	17	5

TABLE II
CHARACTERISTICS OF TEST FILES

B. Setup & Methodology

Now that we have motivated our choice of test cases, let's move on to the setup of the evaluation. Our hardware and software setup are as follows:

- **Processor:** 11th Gen Intel Core i3-1115G4
- **RAM:** 8 GB LPDDR4x
- **Operating system:** Windows 11
- **Python Version:** 3.11

When it comes to the testing methodology, we use a script for automating the testing process. It is centered around a function called "test_brainfuck_files" which takes the path to the folder containing the transpiled Brainfuck test files and the number of times to run each file and returns a dictionary with the average execution time for each file. This function is called three times on the test files transpiled using syntactic optimization, semantic optimization and a non-optimized transpilation used as a baseline. To actually run the transpiled test files and evaluate the average execution time, we make use of another script with the methods "run_program_multiple_times" which is responsible for actually executing a given program the specified amount of times and timing the executing. To facilitate the execution and timing, we make use of Pythons built-in 'subprocess' and 'time' modules.

C. Analysis of results

Using the described hardware, software and methodology, we transpiled the test files using syntactic optimization, semantic optimization and a non-optimized transpilation and executed them all 10 times. The data from is seen in table III and plotted in figure 5.

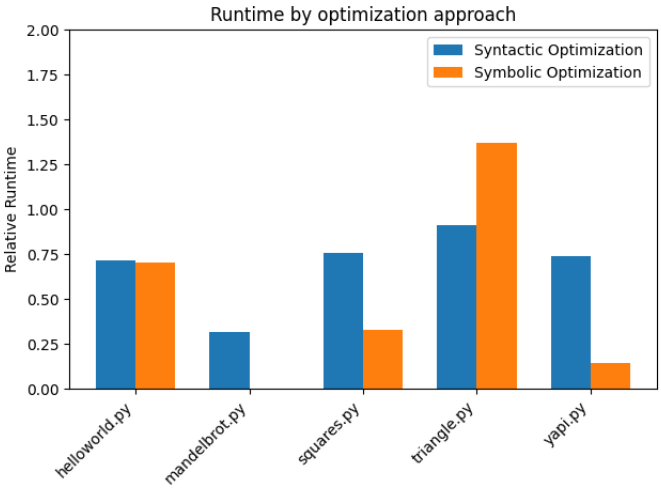

Fig. 5. Execution time of test cases by optimization approach (relative to non-optimized)

Name	Non-optimized (ms)	Syntactic (ms)	Symbolic (ms)
triangle	64.38	58.78	88.25
mandelbrot	606051	193731	187.64
yapi	336.95	248.72	49.52
helloword	67.86	48.55	47.66
squares	184.83	140.41	60.75

TABLE III
EXECUTION TIMES FOR TEST CASES FOR EACH APPROACH IN MILLISECONDS

If we start examining results on for each test case, starting from the simplest, namely helloworld, we see that both approaches achieve similar results - they are around 25% faster than the non-optimized version. When we examine the transpiled files of helloworld.b using semantic and syntactic optimizations, one would think that the semantic would be faster (which it is slighty) as it does not contain any loops and has fewer instructions and assignments of values as to the syntactic version with has 5 loops and its twice as long, however given how simple the helloworld.b program is, the optimization benefits for it is to a larger extend constrained by overheads (i.e. from I/O operations) compared to more complex programs.

For our most complex program, mandelbrot, we find a larger optimization gain - for both approaches, however the clear 'winner' here is the semantic optimization approach as it is 3229 times faster than the non-optimized version. This is due to the semantic approach essentially eliminating all loops and reducing the transpiled version of mandelbrot to assignments and print statements, where the syntactic and non-optimized version still handles the loops of mandelbrot (which had 686 loops with 677 of them being nested), yet the syntactic optimization approach still results in it being 313% faster than the non-optimized version.

Unlike with the semantic optimization for mandelbrot may tell us, it is not always resulting in performance increase when we eliminate the loops, as we see in the triangle test case, where - surprisingly - the semantic is even 37% slower than the non-optimized version. First, this is explained by that the semantic approach on the triangle example results in a lengthy code (3530 lines) of assignments and print statements that is to be executed while the syntactic and non-optimized approaches manage it in 123 and 225 lines respectively. Nonetheless, it is not only the length of the code which explains why the semantic approach for the triangle test case is relatively slower, but also the fact that the while-loops in the non-optimized and syntactic approaches are executed fast (as the exit condition is reach faster), resulting in a faster termination. The syntactic optimisation approach results in a performance gain of 9.5% compared to the non-optimized version.

In the square test case which is similar in complexity to the triangle test case - as both have a closely similar number of loops, nested loop and nesting depth, but with the major difference being less print statements in the square test case. One would think that the semantic approach would perform badly again here, but given the fewer print statement, the semantic approach results in a smaller code file and this time the syntactic approach spending more time in loops before terminating. So for the square case, the semantic approach is 3 times faster than the non-optimized version, while the syntactic is 132% faster than the non-optimized version.

This leads us to our very last test case - namely yapi. The syntactic approach here is 35% faster than the non-optimized version, while the semantic version is 680% faster.

D. Syntactic optimization results

In this section, we focused on evaluating the performance impact of various syntactic optimization techniques. To get reliable measurements of performance, we compared the wall-clock runtime of each program, both in its original (unoptimized) form and after applying the respective optimizations. This evaluation was conducted by averaging the runtime over 10 separate runs for each combination of program and optimization technique. The results are presented in terms of a ratio of optimized vs unoptimized runtime. By analyzing this ratio, we can effectively quantify the efficiency gains offered by each optimization technique, from which we can draw conclusions about use cases of syntactic optimization. All of the techniques compared on grahps are described in detail in subsection **??**

Here is the graph showing relative wall-clock runtime of optimized brainfuck code compared to unoptimized one as ratio. The reason that $sierpinski.b$ and $hellom.b$ benefit from this optimization the most might be because majority of the brainfuck code consists of "+" and "-".

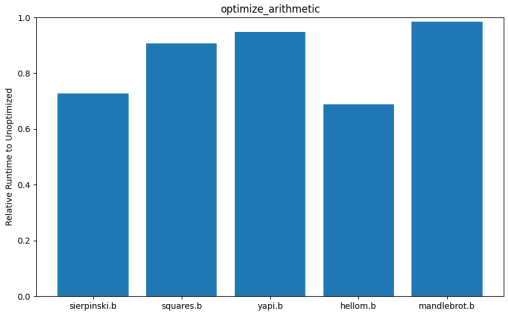

Fig. 6. Chart for relative wall-clock runtime of optimized programs using optimize_arithmetic

This graph shows optimization for optimize_pointer. It shows the best results for $mandlebrot.b$, mainly because of the amount of nested pointer changes relative to the amount of values being outputed.

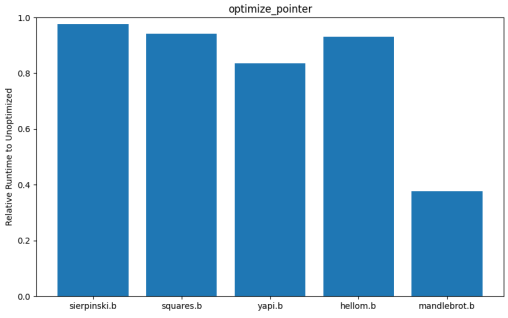

Fig. 7. Chart for relative wall-clock runtime of optimized programs using optimize_pointer

For optimize_clear_loops technique, the best results are achieved on the $sierpinski.b$ program. The reason behind this might be that it uses them two times, both of which are in deeply nested loops.

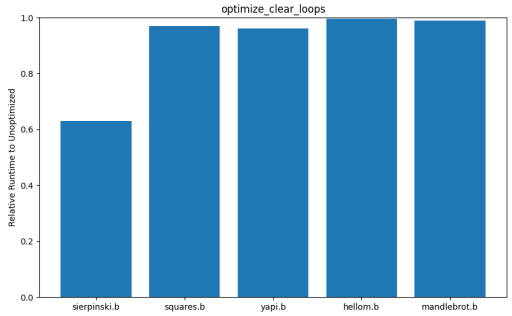

Fig. 8. Chart for relative wall-clock runtime of optimized programs using optimize_clear_loops

copy_loop_optimization was not utilized at all in any of the programs other than $sierpinski.b$. If is used just one time in $sierpinski.b$ and doesn't provide much improvement.

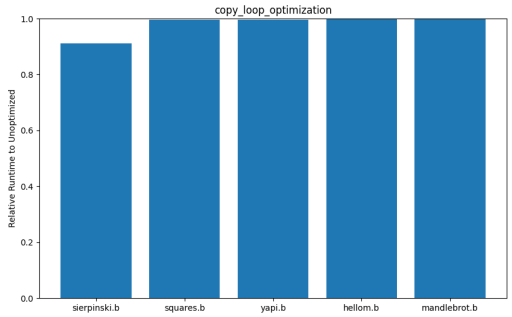

Fig. 9. Chart for relative wall-clock runtime of optimized programs using copy_loop_optimization

This is the chart with all of the optimizations combined. It shows the best results for $sierpinski.b$ and $mandlebrot.b$, which was not surprising looking at separate optimizations techniques. Those two programs implement many typical brainfuck schematics, so it benefited greatly from the optimization.

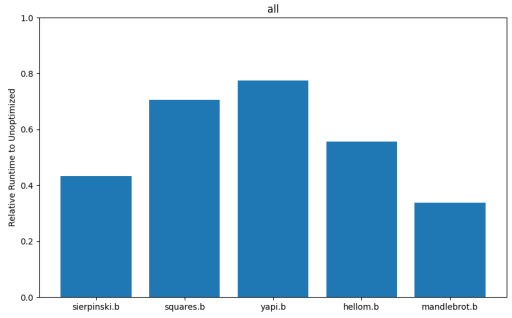

Fig. 10. Chart for relative wall-clock runtime of optimized programs using all optimizations

E. Statistical evaluation

Now that we have seen and explained to results, let us test our hypothesis using student's t-test. Our null-hypothesis is as follows:

H_0: There is no significant difference in the execution time between the symbolic and syntactic approaches.

Along with that we form our alternative hypothesis as follows:

H_1: The symbolic approach has better execution times than the symbolic approach.

As we are testing for the direction of difference, we will conduct a one-tailed t-test and as we are working with execution times we do the one-tailed t-test on the log-transformed data, which results in:

$$p_{value} = 0.141 \text{ and } t_{stat} = -1.152$$

As the p_{value} is grater than the significance level of 0.05, we can not reject the null hypothesis and hence we cannot conclude that the symbolic approach faster than the syntactic approach. The negative t_{stat} value indicate that the symbolic approach then to have lower execution time compared to the syntactic approach, but this is not statistically significant which can be concluded from the p_{value}.

F. Threats to validity

There are of number of factors that threats the validity of our study. We will list them below and discuss each of these threats:

- **Overhead of subprocesses**: In our execution of the transpiled brainfuck programs, we use Pythons subprocess module. Before we start a subprocess to run a program, we start the time and it it after the subprocess is finished. This introduces some overhead, which is not related to the execution of the transpiled brainfuck programs [7]. However, we do not consider this a huge threat for the comparisons we make as all our transpiled Brainfuck program run to the same method.
- **Accuracy of timer**: For timing, we use Pythons time module and so its accuracy as crucial for our study and the data we eventually get from running our tests. Therefore, the time module is another factor that may act as a threat validity. The difference in accuracy of the timer of use is said to be less than 20 ms [6], but given that most of our test cases run a few hundred milliseconds or even less, it may have a larger effect on the accuracy of our data than we initially imagined.
- **Variation in execution times due to system resources**: Due to the inherit nature of how computers work, the same program executed twice will have different execution times. This is due to the fluctuating availability of system resources - including CPU, memory access, I/O devices. We have tried to mitigate by running each

test 10 times and then take the average execution times from those 10 runs. Here, an even bigger number of repetition would have provided even more reliable data not influenced by the variations in the availability of system resources.
- **Limited number of test cases**: We have done our evaluation based on 5 brainfuck program and while these have been selected to vary in complexity, having more programs may have painted a different picture of how the two optimization approaches perform in relation to each other. So, having done the evaluation on more test cases would have allowed for more data and hence also a broader and more detailed evaluation.
- **Outlier on t-test**: The execution time for Mandelbrot in the syntactic approach is significantly higher than the other files and this may affect the results of our statistical evaluation by inflating the variance and skewing the results. Again, more test cases and/or more balanced data could help mitigating this risk.

V. Discussion

In this section we will discuss our two approaches as well as other approaches that could have been suitable for the same task - many optimizing brainfuck programs.

A. Syntactic optimization approach

The main advantage with the syntactic approach is that it is fast when it comes to optimizing and transpiling. This speed is mostly due to the AST that allows for a simple tree structure for identifying patterns and then applying optimizations. One the other hand, it ends up being relatively slow when running the transpiled version of large brainfuck programs as we saw with mandelbrot.

The syntactic approach is limited in doing optimizations which require understand of the codes semantic. This is for example is optimizing loops and evaluating their effect on variables, which is something better suiting by a semantic approach.

All in all, the syntactic analysis approach using AST is a fast way to optimize and transpile, yet it does not achieve the deepest level of optimization as it does not focus on the meaning of the code but the syntactic structure.

B. Semantic optimization approach

Moving on to the semantic approach, its main advantage - which we briefly mentioned above - as its ability to understand the effect of each operation. This allows it to have a deeper understanding of the program and its states and we use that understanding to identify and compress assignments and loops. This leads to simpler programs that only consist on assigning variables to values and printing them. We only introduce loops when they are symbolic loop (which iterate over a input variable that we cannot evaluate).

The biggest disadvantage with the semantic approach is that it is time-consuming and generally more computationally intensive than the syntactic approach. For example it took well over an hour to analyze, optimize and transpile mandelbrot using the semantic approach! That makes it unsuitable for scenarios where speed is an important factor - especially for large and complex programs like mandelbrot.

In conclusion, the semantic approach offer a deep understand allowing in depth optimizations based on the underlying behavior of the programs it is to analyze and optimize. This results in it outputting generally well optimized programs but which comes with a penalty both in form of execution time of the analysis and optimization but also computational resources.

C. Other approaches

In terms of other approaches, for the syntactic analysis we could have implemented the same functionality for optimization without the AST, because of brainfucks minimalistic nature. So, if you do not want to display the wonders of AST, then you can omit it and still have the same functionality instead using many if statements to match the patterns, that would be a simpler solution and faster than parsing and traversing an AST. You could also have used regex expressions to match patterns instead of AST, which you have been simpler and faster as well, yet it would be missing the hierarchical structural approach that AST gives us, especially for nested patterns, thus limiting the usability of regex expression. So, AST are essentially a better option to regex expression as it gives us a structure that we can understand and analyze syntactically easier and thus makes it easier to use for transpilation and deeper analysis i.e. for nested patterns.

Another idea could be to merge the two approaches, so first do syntactic analysis, then do semantic analysis, as this could reduce the time-constraints of semantic analysis and you leverage the advantages of the two approaches.

References

[1] Ball, Katie: ”BrainFuck Programming Tutorial”, https://gist.github.com/roachhd/dce54bec8ba55fb17d3a (last accessed 10/23).

[2] Esolangs: ”Brainfuck”, https://esolangs.org/wiki/Brainfuck (last accessed 10/23).

[3] Linander, Mats: ”brainfuck optimization strategies”, https://calmerthanyouare.org/2015/01/07/optimizing-brainfuck.html (last accessed 10/23).

[4] Schemmel, Daniel et al.:”Symbolic Partial-Order Execution for Testing Multi-Threaded Programs”, https://dl.acm.org/doi/abs/10.1145/2345156.2254118 (last accessed [11/23]).

[5] Meurer, Aaron et. al.: ”SymPy”, https://www.sympy.org/en/index.html, (last accessed [11/23]).

[6] Dharmkar, Rajendra:”How to measure time with high-precision in Python?”, https://www.tutorialspoint.com/How-to-measure-time-with-high-precision-in-Python (last accessed [11/23]).

[7] javatpoint:”Python Subprocess Run”, https://www.javatpoint.com/python-subprocess-run (last accessed [11/23]).
[8] Ball, Thomas:”The Concept of Dynamic Analysis”, https://dl.acm.org/doi/pdf/10.1145/318774.318944 (last accessed [11/23]).