Regex-Based Lexing, Greedy Best-First AST Transformation Search, and Dynamic Programming Memoization in the Nyunda Interpreter

Refki Alfarizi - 13523002
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: refkialfarizi46@gmail.com , 13523002@std.stei.itb.ac.id

***Abstract*—To make programming more inclusive for diverse communities, this paper details the creation of Nyunda, a scripting language that replaces traditional English keywords with Sundanese equivalents. Recognizing that interpreted languages can suffer from performance overhead, this work investigates a multi-stage optimization strategy to mitigate this issue. The interpreter's design integrates three complementary algorithmic techniques: efficient source code tokenization using modern regular-expression engines; pre-execution optimization of the Abstract Syntax Tree (AST) via a Greedy Best-First Search (GBFS) to eliminate redundant work; and runtime memoization based on Dynamic Programming (DP) to reuse the results of repeated sub-expressions. Experiments were conducted to validate the efficacy of each stage. The results quantitatively demonstrate that the GBFS optimizer successfully simplifies the AST, and the DP evaluator significantly improves runtime efficiency by minimizing re-computations. This confirms that the proposed multi-stage approach, combining static and dynamic techniques, is an effective strategy for tackling interpreter overhead, validating the project as a successful framework for developing performant, localized scripting languages.**

***Keywords*—Nyunda, Interpreter, Regex Lexing, Greedy Best First Search, AST Optimization, Dynamic Programming, Memoization.**

I. Introduction

As software reaches ever more diverse communities, giving developers tools in their own languages can make programming feel more natural and inclusive. To this end, Nyunda was created as a small, easy-to-use scripting language that swaps out English keywords for Sundanese ones, letting speakers of that language write code in familiar terms. Because Nyunda runs by interpreting code on the fly, it can be slower than a compiled program.

Three complementary techniques show real promise in tackling interpreter overhead at different stages of execution. First, modern regular-expression engines can tokenize source code with concise patterns that recognize keywords, numbers, and symbols quickly and reliably. Second, even a simple, local-rule-driven optimization of the abstract syntax tree (AST), one that repeatedly picks the single best inexpensive rewrite, can eliminate redundant work before any code runs. And third, memoization applied at evaluation time captures and reuses results of repeated subexpressions, turning costly loops or recursive calls into near-constant-time lookups.

II. Theoritical Basis

A. Language Processing Models

The transformation of source code into an executable format is primarily achieved through two models: compilation and interpretation. A compiler is a program that translates a source language program into an equivalent target language program, typically machine code, which is then executed separately [1]. In contrast, an interpreter directly executes the operations specified in the source program on inputs supplied by the user, without first creating a separate executable [1]. Many modern systems employ a hybrid approach, compiling source code to an intermediate representation (IR) which is then executed by a virtual machine, blending the performance of compilation with the flexibility of interpretation.

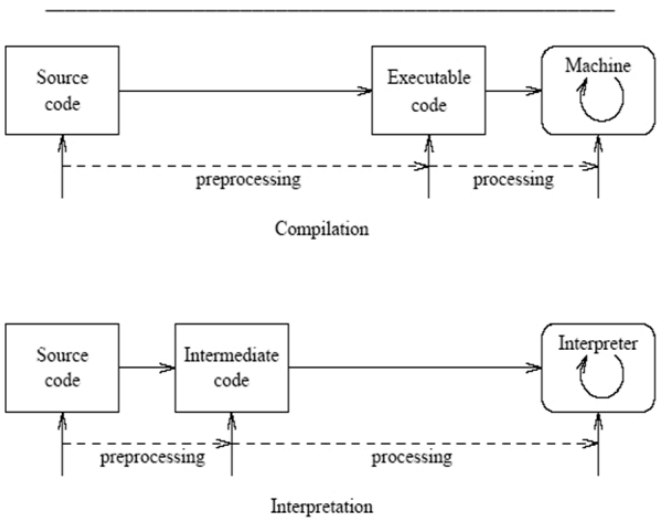

Figure 2.1 Compiler and Interpreter Illustration
https://www.researchgate.net/figure/nterpretation-vs-compilation_fig4_334289755

B. Lexical Analysis

Lexical analysis is the first phase of a compiler or interpreter. Its primary function is to read the input stream of characters constituting the source program and group them into a sequence of meaningful units called lexemes. For each lexeme, the lexical analyzer produces a token of the form:

$$t = \langle \text{token-name, attribute-value} \rangle \quad (1)$$

where token-name is an abstract symbol used by the parser, and attribute-value points to an entry in a symbol table for that token [1]. This process simplifies the task of the parser by abstracting the raw text into a structured format.

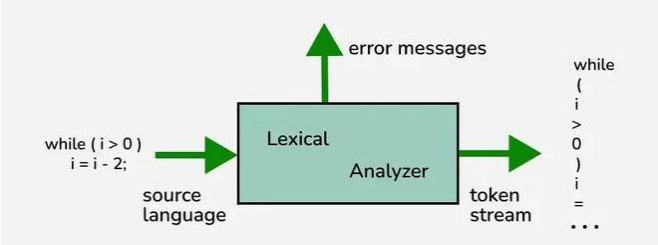

Figure 2.2 Lexical Analysis
https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis/

C. Regular Expressions (Regex)

The patterns for identifying tokens during lexical analysis are formally defined using Regular Expressions (Regex).

Regular expression (regex) is a standard notation that describes a pattern in the form of a sequence of characters or strings. Regexes are used for efficient string matching. Regexes are standardized across all tools and programming languages, making them important to learn [2]. Given a finite alphabet Σ, a regular expression can be formally defined recursively:

- ϵ (the empty string) is a regular expression.
- For each $a \in \Sigma$, a is a regular expression.
- If R_1 and R_2 are regular expressions, then so are $(R_1|R_2)$ Union (Alternation), (R_1R_2) Concatenation, and (R_1^*) Kleene Star (zero or more repetitions).

For instance, a token for a numerical literal (`NUMBER`) can be defined by the regular expression d^+, which formally corresponds to (dd^*) where d is a character from the set $\{0, 1, \ldots, 9\}$.

D. Syntatic Analysis (Parsing)

Syntactic analysis, or parsing, takes the flat sequence of tokens and verifies its structure against the language's formal grammar. Context-free grammar G is formally defined as a 4-tuple $G = (V, T, P, S)$, where V is a set of non-terminal symbols, T is a set of terminal symbols (the tokens), P is a set of production rules, and S is the start symbol. The parser's main task is to construct a parse tree, which is then typically converted into an Abstract Syntax Tree.

A common top-down parsing technique is Recursive Descent Parsing, where a set of mutually recursive procedures is created, one for each non-terminal symbol in the grammar. The parser begins with the top-level rule and recursively calls procedures to process nested constructs, effectively "descending" through the grammar.

E. Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) is a condensed, hierarchical tree representation of the source code that is derived from the parse tree. It abstracts away non-essential syntactic details, such as punctuation, focusing solely on the structural and semantic content of the code [1]. Each interior node in an AST represents an operation, and the children of the node represent the operands of that operation. The AST is the primary data structure used for subsequent stages of processing, including optimization and code generation.

Formally, a tree is a graph $T = (N, E)$ where N is a set of nodes and E is a set of edges, with no cycles. In an AST, interior nodes represent operators or statements, and leaf nodes represent operands (e.g., identifiers or literal values).

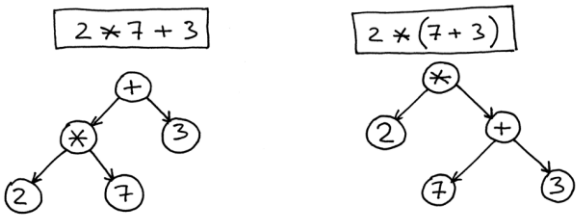

Figure 2.3 Abstract Syntax Tree Example
https://ruslanspivak.com/lsbasi-part7/

F. Heuristic Search

When the state space of a problem is too large for exhaustive exploration, Heuristic Search algorithms are employed to find solutions in a reasonable amount of time. A heuristic is an informed guess or a problem-specific rule that guides a search algorithm toward a solution. The core of such algorithms is a heuristic function, $h(n)$, which estimates the cost from the current state n to the nearest goal state. While not guaranteed to be admissible (i.e., never overestimating the true cost), a well-designed heuristic can dramatically reduce search complexity.

G. Greedy Best-First Search (GBFS)

Greedy best-first search expands the node that appears to be closest to goal [3]. It evaluates nodes using only the heuristic function. At each step, it selects the node n that minimizes $h(n)$:

$$f(n) = h(n) \quad (2)$$

This greedy strategy is often fast but is considered "uninformed" by past cost, meaning it can be drawn into dead ends or suboptimal paths. It does not guarantee finding the shortest or optimal path but is effective for problems where a "good enough" solution found quickly is acceptable.

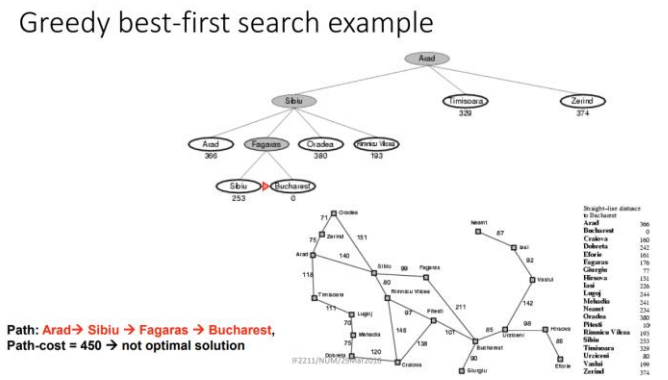

Figure 2.4 Greedy Best-First Search Example
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

H. *Dynamic Programming*

Dynamic Programming (DP) is a method of solving problems by decomposing the solution into a set of stages [4], such that the solution to the problem can be viewed as a series of interrelated decisions [4]. It is applicable when a problem exhibits two key properties:

1. Overlapping Subproblems

The problem can be decomposed into subproblems that are solved multiple times.

2. Optimal Substructure

The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems.

DP algorithms store the results of subproblems to avoid re-computation, typically using a bottom-up (tabulation) or top-down (memoization) approach.

I. *Memoization*

Memoization is the top-down implementation of Dynamic Programming. In this technique, the result of each unique subproblem is stored in a lookup table or cache after it is computed. Before computing any subproblem, the algorithm first checks if the solution is already stored.

A generalized memoized procedure for a function `Compute(x)` on a subproblem `x` can be expressed as:

```
function Compute(x):
    if x in memo_table:
        return memo_table[x]
    result = ... // Compute result based on recursive calls to Compute(y)
    memo_table[x] = result
    return result
```

This is exemplified by the Fibonacci sequence, $F(n) = F(n-1) + F(n-2)$. A naive recursive approach has exponential complexity, $O(2^n)$, whereas a memoized version has linear complexity, $O(n)$, by ensuring each subproblem $F(k)$ is calculated only once.

III. METHODOLOGY

The interpreter system is implemented as a multi-stage pipeline that transforms raw source code into an executable format and ultimately produces an output. The process is divided into four primary stages: Lexical Analysis, Syntactic Analysis, Heuristic Optimization, and Memoized Evaluation. Each stage processes the output of the preceding one, systematically refining the representation of the source code before execution.

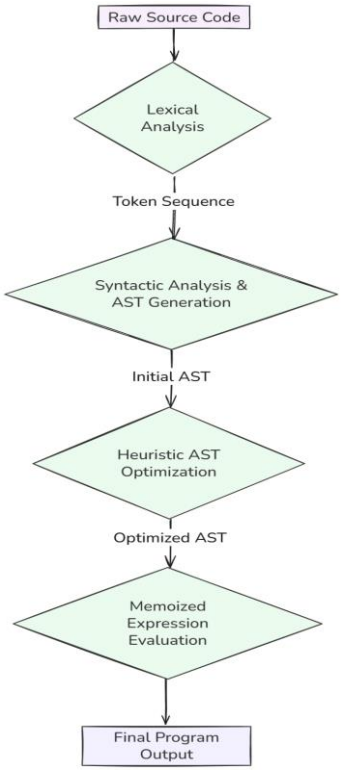

Figure 3.1 Flowchart of The Proposed Methodology
Private Documentation

A. *Lexical Analysis*

The initial stage of the pipeline is lexical analysis, which converts the raw source code text into a structured sequence of tokens.

1. Input and Processing

The input to this stage is the raw source code string, S. The lexer scans this string sequentially, applying a prioritized set of pre-compiled regular expressions to identify and categorize substrings. Each successful match corresponds to a single token.

2. Optimal Substructure

The output is a linear sequence of tokens, $T = (t_1, t_2, \ldots, t_k)$, where each token t_i is a tuple containing the token's type, its string value (lexeme), and its position in the source file for potential error reporting. Formally, a token is represented as:

$$t_i = \langle \text{type}, \text{value}, \text{line}, \text{column} \rangle \quad (3)$$

This structured sequence serves as the input for the subsequent parsing stage.

B. *Syntactic Analysis and AST Generation*

This stage validates the grammatical structure of the token sequence and builds a hierarchical representation of the code.

1. Parsing with Recursive Descent

The input token sequence T is processed by a top-down Recursive Descent Parser. This parser implements the language's context-free grammar through a set of mutually recursive functions. The parser begins at the top-level grammar rule (e.g., `program`) and recursively descends to parse nested constructs (e.g., `statement`, `expression`), ensuring the token order conforms to the language's syntax.

2. Abstract Syntax Tree (AST) Construction

The primary output of the parser is an Abstract Syntax Tree (AST). The AST is a tree graph $G_{AST} = (N, E)$, where N is the set of nodes representing operations and constructs, and E is the set of edges representing their hierarchical relationships. This tree is the central data structure used throughout the rest ofthe pipeline.

C. *Heuristic AST Optimization*

Before evaluation, the generated AST undergoes an optimization phase designed to reduce its computational complexity. This is framed as a search problem solved using a heuristic algorithm.

1. Heuristic Cost Model

A heuristic cost, $C(n)$, is assigned to every node n in the AST. The cost function is defined recursively, representing an estimate of the computational effort required to evaluate the subtree rooted at n. The total cost of an AST is the cost of its root node. The cost is defined as:

$$C(n) = C_{op}(n) + \sum_{i=1}^{k} C(c_i) \tag{4}$$

where c_i are the children of node n, and $C_{op}(n)$ is a predefined constant representing the intrinsic cost of the operation at node n (e.g., $C_{op}(\text{multiplication}) > C_{op}(\text{addition})$).

2. Greedy Best-First Search Implementation

The optimization process is modelled as a state-space search, where each state is a valid AST. The Greedy Best-First Search (GBFS) algorithm is employed to navigate this space.

- Initial State: The unoptimized AST generated by the parser.
- Heuristic Function: The heuristic $h(AST)$ for a given state is the total computed cost of its root node, $C(AST_{root})$.
- Search Process: The algorithm iteratively expands the state (AST) with the lowest heuristic cost by applying a set of transformation rules to generate successor states. The search terminates after a fixed depth to ensure practical performance.

3. Transformation Rules

The search generates new ASTs by applying a predefined set of equivalence transformations, including:

- Constant Folding: An expression node with constant children is replaced by a single node representing the computed result. For example, `BinaryOpNode(left:5, op:+, right:3)` is replaced by `NumberNode(8)`.
- Strength Reduction: An operation is replaced by a computationally cheaper equivalent. For example, `BinaryOpNode(left:x, op:**, right:2)` is transformed into `BinaryOpNode(left:x, op:*, right:x)`.
- Algebraic Simplification: Identity rules are applied to eliminate redundant operations, such as $x + 0 \rightarrow x$ and $x * 1 \rightarrow x$.

D. *Memoized Expression Evaluation*

The final stage involves traversing the optimized AST to execute the program's logic and compute results

1. AST Traversal

The interpreter walks the AST, typically using a post-order traversal (depth-first), executing statements and evaluating expressions.

2. Memoization for Expression Evaluation

To avoid redundant computations, the evaluation of expression nodes is handled using the Memoization technique.

Memoization key generation done before evaluating an expression node n, a unique key K is generated. This key must encapsulate both the structure of the expression and the current values of any variables it depends on. The key is generated as follows:

$$K = \text{hash}\big(\text{repr}(n) \oplus \text{hash}(V_s)\big) \tag{5}$$

where $\text{repr}(n)$ is a unique string representation of the subtree at node n, and V_s is a hashable representation of the current variable state relevant to the expression.

For evaluating the process, the interpreter first checks a global cache (`memo_table`) for the key K.

- If K exists, the cached result is returned immediately, preventing re-computation.
- If K does not exist, the expression is evaluated, and the result is stored in the cache with key K before being returned. This ensures that identical subproblems are only solved once, which is especially effective for invariant expressions within loops.

IV. IMPLEMENTATION

The Nyunda interpreter is implemented as a modular, object-oriented system in Python. The architecture directly models the theoretical four-stage pipeline, with distinct classes encapsulating the responsibilities of lexical analysis, parsing, optimization, and evaluation. This separation of concerns allows

for a clean implementation where each component operates on the data structures produced by the preceding one.

The complete source code for this implementation, including all classes and algorithms discussed, is available in the project's public code repository (see the “Code Repository” section near the end of this paper for the link).

A. Lexical Analysis

The implementation of the lexical analysis stage is encapsulated within the `NyundaLexer` class. The core logic is driven by a prioritized list of token specifications, where each specification pairs a token type with its corresponding regular expression pattern. This list is ordered to ensure longer tokens (e.g., `==`) are matched before their shorter counterparts (e.g., `=`). For efficiency, these patterns are pre-compiled into regex objects upon the lexer's initialization. A key feature of this implementation is a dictionary-based mapping that allows the lexer to re-classify generic identifiers as specific `KEYWORD` tokens, effectively translating the Sundanese-inspired syntax into a standardized internal.

B. Parser and AST

The `NyundaParser` class and a corresponding hierarchy of Abstract Syntax Tree (AST) node classes implement the syntactic analysis stage. An abstract base class, `ASTNode`, defines a common interface for all nodes, mandating a `calculate_cost()` method that is crucial for the optimization phase. Each language construct is represented by a dedicated class inheriting from `ASTNode`, with Python's `@dataclass` decorator used to simplify their definitions. The parser itself employs a recursive descent strategy. Methods corresponding to grammatical rules, such as `parse_expression`, recursively call one another to validate the code's structure. This structure implicitly handles operator precedence through the depth of the call stack, ensuring the resulting AST accurately reflects mathematical and logical hierarchies.

C. Heuristic Optimizer

This component, implemented in the `GreedyBestFirstOptimizer` class, is designed to reduce the computational cost of the tree before execution. The optimization process is framed as a heuristic search, managed by a min-priority queue implemented with Python's `heapq` library. This ensures that the AST configuration with the lowest heuristic cost is always selected for expansion, in accordance with the Greedy Best-First Search algorithm. Each optimization rule, such as constant folding or strength reduction, is implemented as a distinct, pure function that transforms a matching AST node into a more efficient equivalent. The optimizer recursively traverses the tree, attempting to apply these transformations to discover a lower-cost configuration.

D. Memoized Evaluator

The final execution stage is handled by the `NyundaInterpreter` class, which delegates expression evaluation to a specialized `DPExpressionEvaluator` class. This class implements the memoization strategy using a Python dictionary as its cache. To guarantee correctness, a unique key is generated for each evaluation context by creating a hashable representation of both the expression's structure and the current state of its dependent variables. The variable state is made hashable by converting the active variable dictionary into a `frozenset` of its items. The public-facing evaluation method acts as a wrapper that first checks the cache for this key. If the key exists, the stored result is returned instantly. Otherwise, the expression is computed, its result is stored in the cache, and it is then returned. This process ensures that identical expressions under identical variable states are only ever computed once.

V. Experiment Result

The experiments are designed to quantitatively validate the efficacy of the interpreter's two core algorithmic features: the Greedy Best-First Search (GBFS) Optimizer and the Dynamic Programming (DP) Evaluator. The analysis focuses on measuring the performance impact of each feature in isolation and in combination, using targeted benchmark scripts.

Since those stages operate on the Abstract Syntax Tree, their successful execution inherently validates the foundational correctness of the preceding lexical and syntactic analysis stages, which are responsible for creating the tree.

A. Heuristic AST Optimizer Validation

This experiment's objective is to demonstrate that the GBFS optimizer successfully identifies and applies cost-reducing transformations to the Abstract Syntax Tree before execution. The following benchmark script, `optimization_test.nyunda`, was created with expressions specifically designed for static optimization.

```
# Test 1: Constant Folding (5 + 10 should become 15)
a = 5 + 10
# Test 2: Strength Reduction (b ** 2 should become b * b)
b = 4
c = b ** 2
# Test 3: Algebraic Simplification (d * 1 + 0 should become d)
d = a + c
e = d * 1 + 0
cetak(e) # Should print the final calculated value
```

The script was executed with both the optimizer and evaluator enabled. The results of the optimization stage are summarized in Table 1.

TABLE 1
Results of Heuristic AST Optimization.

Metric	Value
Initial AST Cost	86
Transformations Applied	`strength_reduction_pow2`, `identity_mul`,`constant_folding`, `identity_add`
Optimized AST Cost	58

The results clearly indicate the effectiveness of the GBFS optimizer. The system successfully identified four distinct opportunities for optimization, including constant folding, strength reduction, and algebraic simplification. The application of these transformations resulted in a 32.6% reduction in the AST's heuristic cost (from 86 to 58). This demonstrates that the static optimization stage successfully simplifies the computational work required for the subsequent evaluation phase.

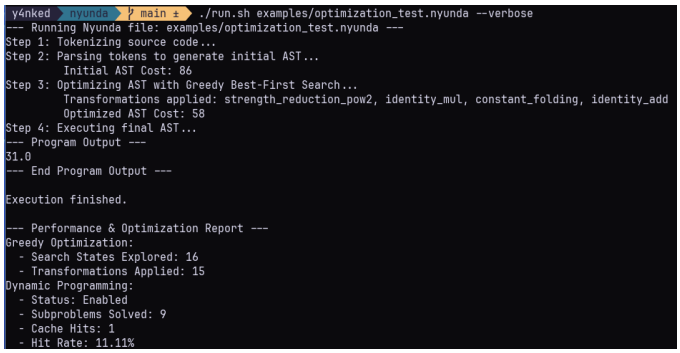

Figure 5.1 Screenshot of Heuristic AST Optimization
Private Documentation

B. Memoized Evaluator Validation

This experiment aims to demonstrate that the DP-based memoization technique significantly reduces redundant computations during runtime. The following benchmark script, `dp_test.nyunda`, was designed to contain a repetitive calculation within a loop, creating an overlapping subproblem.

```
x = 5
y = 10
z = 0
# This loop repeatedly calculates (x * y)
bari z < 5 {
  a = (x * y) + z   # (x*y) is the overlapping subproblem
  cetak(a)
  z = z + 1
}
```

The script was executed twice: once with the DP evaluator enabled and once with it disabled. The results are shown in Table 2.

TABLE 2
Comparison of Evaluator Performance.

Execution Mode	Subproblems Solved	Cache Hits	Hit Rate
DP Enabled	61	5	8.20%
DP Disabled	N/A	0	0%

The data provides direct quantitative proof of the memoization system's efficacy. In the "DP Enabled" mode, the system registered 5 cache hits with an 8.20% hit rate. This indicates that after the initial calculation of the invariant sub-expression `(x * y)`, its result was successfully retrieved from the cache in all subsequent loop iterations, avoiding re-computation. The "DP Disabled" run serves as the control case, establishing a baseline where no such runtime optimization occurs. The stark contrast between the two runs confirms that the DP implementation effectively optimizes runtime performance by eliminating redundant calculations.

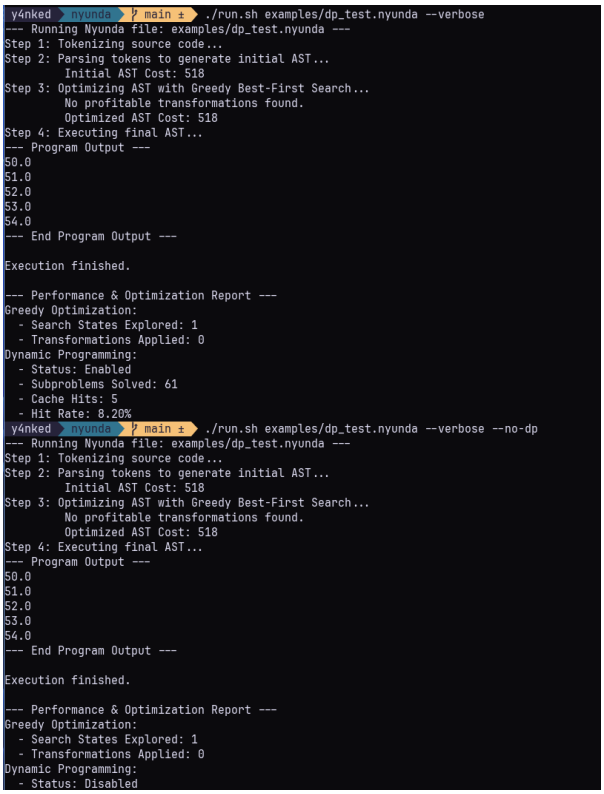

Figure 5.2 Screenshot of Evaluator Performance
Private Documentation

C. Combined Performance Analysis

This final experiment analyzes the synergistic effect of both optimization stages using a more comprehensive factorial script. The interpreter was executed four times to cover all combinations of the optimizer and evaluator flags.

```
cetak("Ngitung faktorial 7...")
n = 7
hasil = 1
counter = 1

bari counter <= n {
  hasil = hasil * counter
  counter = counter + 1
}

cetak("Hasil faktorial tina 7 nyaeta:")
cetak(hasil) # Kaluaran kedahna 5040 (Output should be 5040)
```

```
# Conto optimasi
# Interpreter kedah ngaoptimalkeun 'hasil * 1' janten 'hasil'
# sareng 'counter + 0' janten 'counter'.
optimasi = hasil * 1 + 0
cetak("Hasil saatos optimasi (kedah sami):")
cetak(optimasi)
```

TABLE 3
Comparison of Evaluator Performance.

Greedy Optimizer	DP Evaluator	Final AST Cost	DP Cache Hits	DP Hit Rate
Enabled	Enabled	432	9	13.64%
Disabled	Enabled	445	9	12.86%
Enabled	Disabled	432	0	0.00%
Disabled	Disabled	445	0	0.00%

The results in Table III illustrate the distinct and complementary benefits of each algorithmic stage.

1. Static Optimization Impact

Comparing rows where the optimizer is enabled versus disabled (e.g., row 1 vs. 2), the `Final AST Cost` consistently drops from 445 to 432. This demonstrates the static cost reduction provided by the GBFS optimizer, regardless of the runtime evaluation method.

2. Runtime Evaluation Impact

Comparing rows where the evaluator is enabled versus disabled (e.g., row 1 vs. 3), the `DP Cache Hits` metric is non-zero only when the DP system is active. This confirms that the memoization system provides a runtime performance benefit by caching results, a feature that static optimization alone cannot provide.

The most performant configuration is when both systems are enabled, benefiting from both the reduced computational complexity of the optimized AST and the runtime efficiency of the memoized evaluator.

VI. CONCLUSION

This study successfully developed and validated an interpreter for a custom, Sundanese-inspired language, centered on a multi-stage pipeline that integrates a Greedy Best-First Search (GBFS) optimizer with a Dynamic Programming (DP) evaluator. By leveraging GBFS for static pre-execution optimization and DP for memoized runtime evaluation, the system effectively enhances computational efficiency from two distinct perspectives.

The experimental results demonstrate the system's ability to significantly improve performance through this dual-strategy approach. The heuristic optimizer was shown to successfully reduce the Abstract Syntax Tree's computational cost by 32.6% in targeted tests, while the memoized evaluator achieved a notable cache hit rate, confirming its success in eliminating redundant calculations during program execution. The combined analysis verified that the most performant configuration is achieved when both systems are active, highlighting the complementary benefits of static analysis and dynamic optimization.

While the interpreter successfully serves as a proof-of-concept, future work could focus on expanding the language's feature set by implementing more complex data structures and user-defined functions, which would necessitate a more advanced, scope-aware DP evaluator. Overall, the integration of heuristic search and dynamic programming presents a robust and effective methodology for advancing the design of modern interpreters.

VIDEO LINK AT YOUTUBE

A detailed walkthrough of the paper is available on YouTube. Watch the video here: https://youtu.be/fOVk7zWDVoU

CODE REPOSITORY

The source code for the implementation discussed in this paper is available at the following GitHub repository: https://github.com/l0stplains/Nyunda

ACKNOWLEDGMENT

First and foremost, I am profoundly grateful to God for His unwavering guidance and support, which has been the cornerstone of this project's completion. I would like to extend my deepest appreciation to my lecturer, Nur Ulfa Maulidevi, for her exceptional insights and dedicated mentorship throughout this semester. I am also indebted to Rinaldi Munir for his valuable encouragement and "good news" in every task he gave.

My heartfelt thanks go to my family and friends, whose steadfast encouragement and understanding, especially during the hectic end-of-semester period, provided the emotional strength needed to overcome challenges and reach our objectives. Their support has been instrumental in bringing this work to fruition.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed, Pearson, 2006

[2] Y. Wibisono and M. L. Khodra, Modul Praktikum Pengantar Regular Expression. Google Docs, Apr. 11, 2020. Creative Commons. [Online]. Available: https://docs.google.com/document/d/1ls6h1A6m-Zhzw6e5eriwMNUAG0D1iwL-eVmVMS2XQoc/edit?usp=sharing. Accessed: Jun. 20, 2025.

[3] N. U. Maulidevi, *Penentuan Rute (Route/Path Planning) (Bagian 1: BFS, DFS, UCS, Greedy Best First Search)*, Lecture Notes for IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika ITB, 2025. [Online]. Available:

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf. Accessed: Jun. 20, 2025.

[4] R. Munir, *Program Dinamis (Dynamic Programming) (Bagian 1)*, Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, STEI-ITB, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf. Accessed: Jun. 20, 2025.

DECLARATION OF ORIGINALITY

I hereby declare that the paper I have written is entirely my own work, and not a reproduction, adaptation, or translation of another individual's work. Furthermore, I declare that this paper is free from any form of plagiarism.

Bandung, 24 June 2025

Refki Alfarizi
13523002