Application of String Matching and Regular Expression in Lexical Analysis for Programming Languages

Henry Filberto Shenelo - 13523108
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: henryfilbertoshenelo@gmail.com , 13523108@std.stei.itb.ac.id

***Abstract*— Lexical analysis plays a critical role in the compilation process by transforming raw source code into structured tokens. However, with the increasing complexity of programming languages and growing codebases, traditional regex-based lexers face challenges in performance and scalability. This study explores the application of string-matching algorithms—Knuth-Morris-Pratt (KMP), Boyer-Moore, and Aho-Corasick—as alternatives or supplements to regular expressions for keyword detection in lexical analysis. By implementing and benchmarking multiple lexer variants, including a regex-only baseline and hybrid manual lexers, the paper evaluates their performance across various input sizes and edge cases, including illegal characters. Results show that Aho-Corasick offers the best overall performance, combining speed with robust error handling. These findings highlight the potential of integrating string-matching techniques into modern lexer implementations for more efficient and resilient compiler front-ends.**

***Keywords*— Lexical analysis, string matching, Aho-Corasick, compiler, regular expressions.**

I. Introduction

Lexical analysis is a fundamental phase in the compilation process, transforming source code into a sequence of tokens that serve as the basis for syntax parsing and semantic analysis. A lexer, or lexical analyzer, must efficiently identify language constructs such as keywords, identifiers, literals, and symbols while handling increasing code complexity and volume. Traditionally, lexers rely heavily on regular expressions due to their expressive power and compatibility with finite automata, which enable efficient linear-time scanning of input.

However, regex-based tokenization can become inefficient when distinguishing between keywords and identifiers, especially in languages with large or evolving keyword sets. To address this, string matching algorithms such as Knuth-Morris-Pratt (KMP), Boyer-Moore, and Aho-Corasick offer optimized methods for pattern recognition, particularly for fixed patterns like keywords. These algorithms can reduce redundant comparisons and improve keyword matching performance.

This paper explores a hybrid approach to lexical analysis that combines regular expressions for general token types with string-matching algorithms for fast keyword recognition. Implementations of manual lexers using KMP, Boyer-Moore, and Aho-Corasick are compared against a baseline regex-only lexer and a lexer built using the ply.lex library. Performance benchmarks and error-handling tests are conducted to evaluate the speed, accuracy, and resilience of each method across multiple input sizes and scenarios.

The remainder of this paper is organized as follows: Section II introduces the theoretical foundations behind regular expressions, finite automata, and classic string-matching algorithms including KMP, Boyer-Moore, and Aho-Corasick. Section III describes the implementation of multiple lexer variants using regex and manual string matching. Section IV presents experimental results, including performance benchmarks and tokenization outputs. Finally, Section V concludes with key takeaways and suggestions for future improvements in lexical analysis techniques.

II. Theoretical Basis

A. Pattern Matching

Pattern matching is the process of finding the occurrence of a specific sequence of characters, known as a pattern, within a larger body of text. Formally, given:

T: a text string of length n characters, and

P: a pattern string of length m characters (m << n)

Maintaining the Integrity of the Specifications

the goal is to identify the position(s) in T where P appears as a contiguous substring. Efficient algorithms like Knuth-Morris-Pratt (KMP) and Aho-Corasick are often employed to perform pattern matching in optimal or near-optimal time.

B. Knuth-Morris Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is an efficient pattern matching algorithm that searches for a pattern string P of length m within a text string T of length n from left to right. When a mismatch occurs at position j in P, the algorithm uses a border function (failure function) to determine the longest prefix of $P[0..j-1]$ that is also a suffix of $P[0..j-1]$. The KMP algorithm consists of two main steps:

1. Preprocessing: Compute the border function for the pattern.
2. Searching: Use the border function to perform pattern matching throughout the text (T).

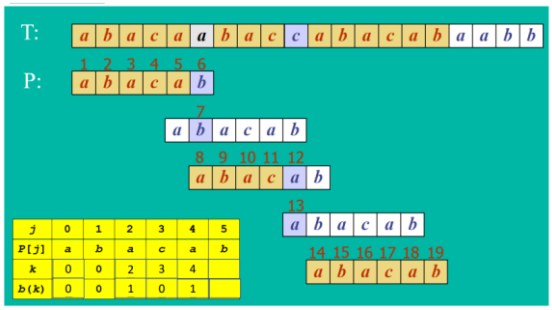

Figure 1. Pattern matching with KMP algorithm
Source: [2]

C. *Aho-Corasick Algorithm*

The Aho-Corasick algorithm begins by building a trie from all the patterns, where each node represents a character. Suffix links are then added to efficiently handle mismatches by pointing to the longest valid suffix that is also a prefix of another pattern. Output links are used to detect overlapping pattern matches. During text processing, the automaton moves forward on matching characters or follows suffix links on mismatches. When reaching an accepting node, a match is recorded, and output links are checked for additional matches at the same position.

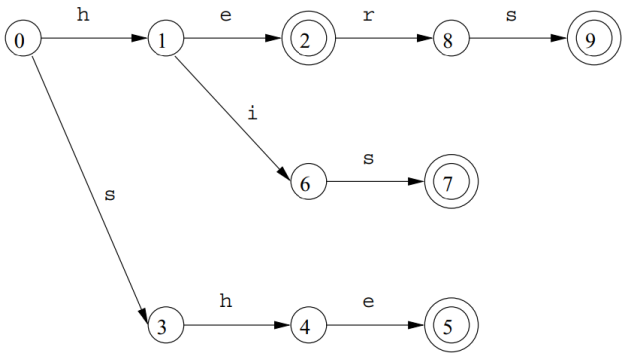

Figure 2. Example trie of Aho-Corasick
Source: [1]

D. *Boyer-Moore Algorithm*

The Boyer-Moore algorithm is a fast pattern-matching algorithm that scans the pattern from right to left using the looking-glass technique and applies character jump heuristics to skip ahead in the text during mismatches. It precomputes a last occurrence table, which maps each character in the alphabet to its last position in the pattern. When a mismatch occurs, the algorithm uses this table to decide how far the pattern can safely be shifted to the right, potentially skipping multiple comparisons. There are 3 cases:

1. If the mismatched character exists in the pattern, it aligns with its last occurrence (Case 1).
2. If not or if it cannot be aligned, the pattern shifts by one (Case 2)
3. Else: aligns the start of the pattern with the next character in the text (Case 3).

This reduces unnecessary comparisons, making Boyer-Moore efficient, especially for large alphabets and long texts.

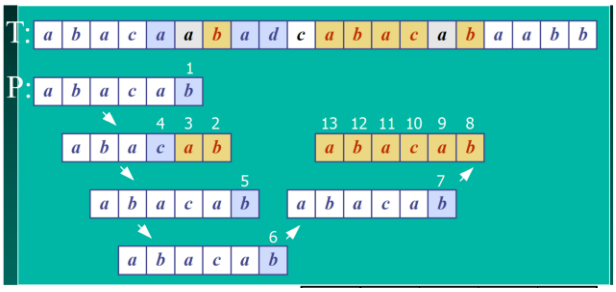

Figure 3. Pattern matching with BM algorithm
Source: [2]

E. *Regular Expression*

A regular expression (regex) is a formal notation used to describe patterns within strings. It originates from formal language theory and automata, particularly regular languages. A regex defines a set of rules that can be used to identify or match specific sequences of characters, such as words, digits, or symbols, making it useful for validating formats or extracting information from text.

F. *Lexical Analyzer*

Lexical analyzer is the first phase of a compiler whose primary function is to read the source program's input characters, group them into lexemes, and convert them into a sequence of tokens. These tokens are then passed to the parser for syntax analysis. During this process, the lexical analyzer may also interact with the symbol table, especially when identifying identifiers that need to be recorded or looked up.

In its operation, the lexical analyzer typically removes unnecessary elements such as whitespace and comments to simplify the input stream for subsequent phases. Lexical analysis involves three key concepts:

1. A token is a pair ⟨token name, optional attribute⟩, where the name (e.g., IF, ID, NUMBER) is used by the parser, and the attribute holds extra information
2. A pattern defines the structure a lexeme must follow to be recognized as a token. For fixed tokens like keywords, the pattern is the exact character sequence. For more general tokens like identifiers or numbers, the pattern is typically expressed using regular expressions.
3. A lexeme is the actual sequence of characters in the source program that matches a given pattern. The lexical analyzer identifies this sequence and classifies it as an instance of the corresponding token.

Table 1. Tokens, patterns, and lexemes

Token	Description	Lexemes
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< , >, <=, >=, !=, ==	<=, !=
id	letter followed by letters or digits	pi, score
number	numeric constant	0, 2.7, 3.14
literal	anything surrounded by “	“error”

G. Finite Automata

Finite automata convert regular expressions into working lexical analyzers. Finite automata come in two types:

1. Nondeterministic Finite Automata (NFA): Allow multiple transitions for the same input symbol and can include ε-transitions (transitions without consuming input).
2. Deterministic Finite Automata (DFA): For each state and input symbol, there is exactly one defined transition, and ε-transitions are disallowed.\

Although nondeterministic finite automata (NFAs) are easier to construct from regular expressions, lexical analyzers typically simulate deterministic finite automata (DFAs) due to their predictable control flow. Every NFA can be converted into a DFA through subset construction, ensuring both recognize the same regular language.

In practice, these DFA-based transition diagrams are implemented using control structures like switch-case statements or transition tables. This approach allows lexical analyzers to efficiently match lexemes against regular expression patterns. Reserved words and identifiers, for instance, can be differentiated either through tailored DFA structures or symbol table lookups after matching a general identifier pattern.

III. IMPLEMENTATION

Based on the theoretical basis, we implement a simple lexical analyzer (lexer) to tokenize source code into keywords, identifiers, literals, and symbols. We explore the use of regex and compare multiple approaches to tokenize, including the use of built-in lex library, Knuth-Morris-Pratt (KMP), Boyer-Moore, and Aho-Corasick automata. The following codes are written in Python inspired by the project from https://github.com/airportyh/smallang and modified to handle a broader set of language features and token types.

First, we define the keywords/reserved words that are going to be used in our custom programming language.

```
keywords = {
    'if', 'else', 'while', 'for', 'in', 'do', 'match', 'switch', 'case',
}
```

Then, we define the tokens used, which serve as the building blocks for syntax analysis by representing distinct elements like keywords, operators, and literals.

```
tokens = (
    'WS', 'COMMENT', 'NUMBER', 'STRING',
    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE',
    'LBRACKET', 'RBRACKET',
    'IDENT', 'FATARROW', 'ASSIGN', 'NL',
    'KEYWORD', 'COMMA', 'COLON',
    'DOT',
)
```

Each of the tokens then can be represented in the form of regular expressions

```
t_LPAREN    = r'\('
t_RPAREN    = r'\)'
t_LBRACE    = r'\{'
t_RBRACE    = r'\}'
t_LBRACKET  = r'\['
t_RBRACKET  = r'\]'
t_FATARROW  = r'=>'
t_ASSIGN    = r'='
t_COMMA     = r','
t_COLON     = r':'
t_DOT       = r'\.'
```

Furthermore, we add other rules using regular expressions for the lexical analyzer. Some of the regexes implemented are:

```
def t_WS(t): //ignore whitespace
    r'[ \t\r]+'
    pass

def t_COMMENT(t): //ignore comment
    r'\/\/[^\n\r]*'
    pass

def t_MLCOMMENT(t): //ignore multi-comment
    r'/\*(.|\n|\r)*?\*/'
    t.lexer.lineno += t.value.count('\n')
    pass

def t_FLOAT(t): //convert to float
    r'[0-9]+\.[0-9]*([eE][+-]?[0-9]+)?'
    t.value = float(t.value)
    return t

def t_NUMBER(t): //convert to int
    r'0|[1-9][0-9]*'
    t.value = int(t.value)
    return t
```

```
def t_STRING(t): //convert string
    r'"(?:\\[ntr"\\]|[^\\\n\r"])*"'
    t.value = bytes(t.value[1:-1], "utf-
8").decode("unicode_escape")
    return t

def t_IDENT(t): //identifiers
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    if t.value in keywords:
        t.type = 'KEYWORD'
    return t

def t_NL(t): //new line
    r'\r?\n+'
    t.lexer.lineno += len(t.value)
    pass

def t_error(t): //illegal characters
    print(f'Illegal character {t.value[0]!r} at line
{t.lexer.lineno}')
    t.lexer.skip(1)
```

The next step is to implement KMP for the pattern matching.

```
def kmp_table(pattern: str): //border function
    m = len(pattern)
    fail = [0] * m
    j = 0
    for i in range(1, m):
        while j > 0 and pattern[j] != pattern[i]:
            j = fail[j - 1]
        if pattern[j] == pattern[i]:
            j += 1
            fail[i] = j
    return fail
```

```
def kmp_match(text: str, pattern: str, table):
    if len(text) != len(pattern):
        return False
    if table is None:
        table = kmp_table(pattern)
    i = j = 0
    n = len(text)
    m = len(pattern)
    while i < n:
        if text[i] == pattern[j]:
            i += 1; j += 1
            if j == m:          # found pattern
                return True     # exact length check
        else:
            if j == 0:
                i += 1
            else:
                j = table[j - 1]
    return False
```

We then also implement Boyer-Moore for comparison of different pattern matching methods.

```
def bm_last_table(pattern: str): //last occurrence table
    last = [-1] * 128
    for i, ch in enumerate(pattern):
        last[ord(ch) & 0x7F] = i
    return last
```

```
def bm_match(text: str, pattern: str, last=None) -> bool:

    if len(text) != len(pattern):
        return False
    if last is None:
        last = bm_last_table(pattern)

    n = len(text)
    m = len(pattern)
    i = m - 1        // index in text
    j = m - 1        // index in pattern

    while i < n:
        if text[i] == pattern[j]:
            if j == 0:
                return True         // full match
            i -= 1
            j -= 1                  // looking-glass
        else:
            lo = last[ord(text[i]) & 0x7F]
// last occurrence of mismatched char
            i += m - min(j, 1 + lo)    // character-jump
            j = m - 1
    return False
```

```
import ahocorasick
A = ahocorasick.Automaton()
for kw in keyword_list:
    A.add_word(kw, kw)
A.make_automaton()
```

We can finally implement manual tokenization with three different approaches and regular expressions.

```
def tokenize(code: str) -> List[Token]:
    tokens: List[Token] = []

    # Position bookkeeping
    line = 1
    last_nl_idx = -1   # index of last '\n' before current
token

    pos = 0
    n = len(code)
```

```
while pos < n:
        m = _master_pat.match(code, pos)
        if not m:  # no match => illegal character
            snippet = code[pos:pos+20].splitlines()[0]
            raise SyntaxError(f"Illegal character
{code[pos]!r} "
                              f"at line {line} col {pos -
last_nl_idx}\n→ {snippet}")
        kind = m.lastgroup
        value = m.group(kind)

        newlines = value.count("\n")
        if newlines:
            line += newlines
            last_nl_idx = m.end() - 1 - value[::-
1].find("\n")  # idx of final \n in this match

        # Skip whitespace/comments
        if kind in _skip_types:
            pos = m.end()
            continue

        # Calculate 1-based column
        col = m.start() - last_nl_idx

        # Value conversions
        if kind == "STRING":
            value = bytes(value[1:-1], "utf-
8").decode("unicode_escape")
        elif kind == "FLOAT":
            value = float(value)
        elif kind == "HEX":
            value = int(value, 16)
        elif kind == "BIN":
            value = int(value, 2)
        elif kind == "NUMBER":
            value = int(value)

        # Keyword check
        if kind == "IDENT" and is_keyword(value):
            kind = "KEYWORD"

        tokens.append(Token(kind, value, line, col))
        pos = m.end()

    return tokens
```

IV. RESULTS

The results of the implemented lexer with different test cases are shown here.

First, we test with simple code written in our custom programming language. Here is the sample of the file written in that language (short.aban).

```
// ---------- Helper Functions ----------
print("Hello, world")
print("3 + 5 =", add(3, 5))
```

The results of the lexical analysis using regex and ply.lex are shown on Table 2

Table 2. Tokenization results of given sample code

Token Type	Token Value	Line	Position
KEYWORD	print	2	42
LPAREN	(	2	47
STRING	Hello, world	2	48
RPAREN	)	2	62
KEYWORD	print	3	64
LPAREN	(	3	69
STRING	3 + 5	3	70
COMMA	,	3	79
IDENT	add	3	81
LPAREN	(	3	84
NUMBER	3	3	85
COMMA	,	3	86
NUMBER	5	3	88
RPAREN	)	3	89
RPAREN	)	3	90

The same lines of code are also tested using manually built lexer with the help of string matching. The results are shown below.

Table 3. Manual tokenization results of given sample code

Token Type	Token Value	Line	Column
KEYWORD	print	2	1
LPAREN	(	2	6
STRING	Hello, world	2	7
RPAREN	)	2	21
KEYWORD	print	3	1
LPAREN	(	3	6
STRING	3 + 5 =	3	7
COMMA	,	3	16
IDENT	add	3	18
LPAREN	(	3	21
NUMBER	3	3	22
COMMA	,	3	23
NUMBER	5	3	25
RPAREN	)	3	26
RPAREN	)	3	27

We also conduct benchmark tests to compare the time taken to tokenize different files with different lengths using every method discussed.

1. Short code (8 words)

Table 4. Time comparison to process short code

No	Without string matching	KMP	Boyer-Moore	Aho-Corasick
1	0.00306	0.00048	0.000484	0.00033
2	0.001149	0.000516	0.0004785	0.0003158
3	0.001887	0.000759	0.0006542	0.000439
4	0.002208	0.001046	0.000735	0.000406
5	0.0010989	0.0004916	0.0004564	0.0003149
Avg	**0.00188**	**0.000658**	**0.000562**	**0.000361**

2. Medium code (93 words)

Table 5. Time comparison to process medium code

No	Without string matching	KMP	Boyer-Moore	Aho-Corasick
1	0.007013	0.014898	0.00786	0.00449
2	0.005632	0.0100315	0.0081276	0.004505
3	0.005958	0.013833	0.010863	0.0056857
4	0.0103468	0.009138	0.0099435	0.0059944
5	0.00568	0.0104329	0.0163726	0.0058612
Avg	**0.006926**	**0.01167**	**0.01063**	**0.005307**

3. Long code (471 words)

Table 6. Time comparison to process long code

No	Without string matching	KMP	Boyer-Moore	Aho-Corasick
1	0.024489	0.0533	0.048673	0.0273208
2	0.027188	0.0519959	0.0551206	0.027423
3	0.023817	0.05903	0.055695	0.0307356
4	0.023297	0.0556105	0.054404	0.0232055
5	0.0309209	0.0526324	0.04748789	0.0284981
Avg	**0.0259424**	**0.0545138**	**0.052276**	**0.0274366**

We also test tokenization of a file with illegal characters and compare the time taken by each method.

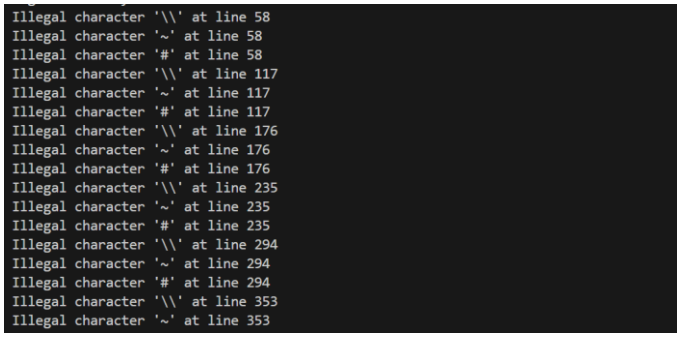

Figure 4. Example result to detect illegal character in tokenization

Table 7. Time comparison to process file with illegal characters

No	Without string matching	KMP	Boyer-Moore	Aho-Corasick
1	0.003469	0.0068031	0.006926	0.003649
2	0.0044389	0.005397	0.0054302	0.0028848
3	0.0034599	0.0074711	0.0052853	0.0026011
4	0.004053	0.0055969	0.0055421	0.0030429
5	0.006241	0.005499	0.0063687	0.0041024
Avg	**0.004332**	**0.006153**	**0.005910**	**0.003256**

The results demonstrate slight performance advantages when integrating string matching into a lexer, particularly using the Aho-Corasick algorithm. As shown in the benchmark tables, Aho-Corasick consistently outperforms the other approaches across short and medium file sizes. However, there is no noticeable difference between Aho-Corasick performance for long file code and "without string matching" lexer.

Overall, Aho-Corasick's ability to match multiple patterns simultaneously in linear time leads to speed-ups, compared to the regex-only baseline. Meanwhile, the Boyer-Moore and KMP implementations, though theoretically efficient, suffer from overhead due to repeated pattern comparisons for each keyword, making them slower than even the baseline in larger inputs. The regex-only approach, while being competitive, lacks keyword awareness, and also perform worse for short codes.

The manual lexers showed good performance in handling illegal characters. Among them, the Aho-Corasick lexer not only maintained the fastest processing times but also detected illegal characters efficiently, making it highly suitable for continuous analysis in tools like editors or compilers. KMP and Boyer-Moore also performed well in this aspect, offering solid error handling and competitive speed.

V. CONCLUSION

The integration of regular expressions with string-matching algorithms such as Knuth-Morris-Pratt, Boyer-Moore, and Aho-Corasick enhances the efficiency of lexical analyzers. Regular expressions provide a robust foundation for pattern recognition, while string-matching algorithms optimize keyword detection by reducing redundant comparisons. Among these, Aho-Corasick stands out for its ability to match multiple patterns simultaneously with minimal overhead, making it particularly well-suited for large-scale or real-time lexical analysis. The experimental results confirm that combining these techniques yields faster tokenization, faster error handling, and improved scalability, demonstrating their practical value in building modern, high-performance compiler front-ends.

VI. APPENDIX

Source code used in Section III Implementation can be viewed here: https://github.com/henry204xx/abanlang

VIDEO LINK AT YOUTUBE

https://youtu.be/DrofxSiNxgE

ACKNOWLEDGMENT

The Author would like to express gratitude to Mr. Rinaldi Munir, the lecturer for Algorithm Strategy at Bandung Institute of Technology, for his guidance and comprehensive teaching of the subject, which provided the foundation for understanding the concepts applied in this paper.

The Author also acknowledges the various sources of information that greatly contributed to the completion of this paper. These include academic journals, articles, online resources, and publicly available code repositories, all of which offered invaluable insights and practical tools necessary for the development of this work.

REFERENCES

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

[2] R. Munir, "Pencocokan string," *Kuliah STMIK*, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf

[3] R. Munir, "String matching dengan regex," *Kuliah STMIK*, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf

[4] airportyh, "Smallang: A small programming language demo," GitHub Repository. [Online]. Available: https://github.com/airportyh/smallang

[5] B. Sukumaran, "The essence of Aho-Corasick algorithm," *Medium*, 2020. [Online]. Available: https://medium.com/@balajisukumaran96/the-essence-of-aho-corasick-algorithm-4056dd742842

[6] V. Paxson et al., "Flex: The fast lexical analyzer," *Flex Manual*, [Online]. Available: https://westes.github.io/flex/manual/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

Henry Filberto Shenelo 13523108