# Leveraging Karnaugh Map Logic for Streamlining Code Design

Nadhif Radityo Nugroho (13523045)[1,2]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523045@mahasiswa.itb.ac.id, [2]nadhifradityo@gmail.com

***Abstract*—Code development has been a struggle for most people. Constructing a robust and complex algorithms always involve convoluted logical expressions. Developers should not worry about making complex logic, as they have more important works to do: implementing algorithms and business requirements. This paper explains how Karnaugh map logic simplification paired with language-tailored abstract syntax tree, can be used in a code development process as a user suggestion. Allowing more readable code and streamlined code design.**

***Index Terms*—Karnaugh Map, Abstract Syntax Tree, Code Development**

## I. Introduction

Code development has been evolving since computers exist. Well known and intelligent people have invented code development practices and standards, allowing projects to be more collaborative. Said practices and standards usually differ between projects, revolving around a domain the projects are working on. Naming convention, branching rules, and ownership management are few examples the practices are about. But they all have one characteristic: readable simple logical expressions are always preferred than redundant complex logical expressions. Poorly written logics can lead to higher error rates, increased debugging time, and difficulty in future updates and scalability. Simplifying logical expressions are not only for maintainability and readability, but it also about efficiency. Compiler can produce more performant executable with the help of simple logics. All in all the requirement for readable simple logical expressions are in need in modern software engineering practices.

## II. Theoretical Foundations

### A. Karnaugh Map

Karnaugh Map (K-Map) is a method to optimize logical expressions. Maurice Karnaugh invented this technique in 1953, with idea that similar neighbouring values can be expressed in a simplified manner with blocks recognition. A rule must be satisfied first, which is neighbouring cells must have a single difference in its bit value. Cells also must contain only one literal—a final boolean result the logical expressions returned.

### B. Abstract Syntax Tree

Codes, as a way for humans to tell computers to do things, are often optimized to bridge human concepts to machine instructions. These bridges are usually designed to be human-like languages, but still on a threshold to match the language usage and their principles. Codes that act as bridges are often problematic to compile on their own. The use of Abstract Syntax Tree (AST) helps convert the language set of rules to a tree-like syntax that can be parsed and understood quickly by a compiler.

### C. Code Development

Standards and practices on code development have evolved over time. Paradigms have been invented to satisfy the requirements of modern-world businesses. Object-oriented design and agile methodologies have shown their ability to provide the ease of coding. But object-oriented design, agile methodologies, and any other paradigms have things in common: the importance of readability and maintainability. Clear and logical code structures often lead to lower error rates and easier debugging. Not to mention in a system that life could be at stake, the importance of clear controlled code flow is rather mandatory.

## III. Methodology

This paper leverages practical methods as its methodology. This approach was chosen because it fit the need to prove the implementation of the Karnaugh map in a code design. With that said, here are the main outlines of this paper:

- Abstract Syntax Tree Parsing: Parses the language abstract syntax tree to help identify logical conditions.
- Logical Conditions Extraction: Extracts all possible logical conditions from the previous step.
- Karnaugh Table Construction: Creates the equivalent truth table based on the logical conditiions.
- K-Map Pattern Recognition: Simplifies the Karnaugh map table by using well-known Karnaugh techniques.
- Code Reassemble: Generates the code for simplified logical conditions.

Although this method can be generalized, this paper will use Javascript Langugae as a proof medium. A general approach can still be achieved when conversion to generic AST and generic code assembler are available.

## IV. Implementation

Implementation of this paper basically consists of 5 steps. Starting with parsing the code to AST form, continued with extraction of logical conditions, Karnaugh table construction, pattern recognition, and finally reassemble the optimized code. Each step has their own unique problem that fits with a specific domain.

### *A. Abstract Syntax Tree Parsing*

This step basically converts the input code to a syntax that a machine can process easily. This conversion includes recognizing token and structuring it into a tree. With the help of Backus-Naur form, rules with context-free grammars can be defined.

Fig. 1: Backus-Naur Form of Simple Arithmetic Language

```
NUMBER     ::= [0-9]+
IDENTIFIER ::= [a-zA-Z_][a-zA-Z0-9_]*
OPERATOR   ::= "+" | "-" | "*" | "/"

<program> ::= <expression>
<expression> ::= <term> (("+" | "-") <term>)*
<term> ::= <factor> (("*" | "/") <factor>)*
<factor> ::= <number> | <identifier> |
    "(" <expression> ")"
<number> ::= NUMBER
<identifier> ::= IDENTIFIER
```

### *B. Logical Conditions Extraction*

Logical conditions can be easily extracted within keywords that expect logical values, such as `if` statements, `while` statements, and `for` statements. But a general approach can be achieved in some languages, since they general expression evaluation. The Algorithm 1 explains how this process works.

**Algorithm 1** Extract Logical Conditions

```
Require: Abstract Syntax Tree (AST) ast
Ensure: List of logical conditions logicalExpressions
  logicalExpressions ← empty list
  Walk over all nodes in ast:
     if node.type does not end with "Expression" then
        skip node
     else
        Add node to logicalExpressions
     end if
  filteredExpressions ← empty list
  for each node in logicalExpressions do
     parentNode ← parent of node in ast
     if parentNode.type does not end with "Expression" then
        Add node to filteredExpressions
     end if
  end for
  logicalExpressions ← filteredExpressions
```

Although this representation is enough to move forward, it would be wise to normalize these conditions to identifiable expressions. This approach is taken to simplify the construction of Karnaugh table as it involves evaluating distinct expressions.

**Algorithm 2** Normalize Logical Conditions

```
Require: List of logical conditions logicalConditions
Ensure: List of normalized logical conditions normalizedLogicalConditions
  normalizedLogicalConditionsMap ← empty map
  for each node in logicalConditions do
     if normalizedLogicalConditionsMap contains node then
        skip node
     else
        normalizedLogicalCondition ← copy of node
        Assign unique identifier to normalizedLogicalCondition.id
        Add node and normalizedLogicalCondition to normalizedLogicalConditionsMap
     end if
  end for
  normalizedLogicalConditions ← all values in normalizedLogicalConditionsMap
```

### *C. Karnaugh Table Construction*

Constructing Karnaugh table works by evaluating the logical expression for each permutation of available variables. The permutated bits depend on the row and column that they occupy. The table rows and columns are also defined by Gray Code sequences generated by Algorithm 3.

**Algorithm 3** Generate Gray Code Sequences

```
  function GenerateGrayCodeSequences(n)
     if n = 0 then
        return "0"
     end if
     if n = 1 then
        return "0, 1"
     end if
     previous ← GenerateGrayCodeSequences(n - 1)
     result ← empty list
     for each string s in previous do
        Append "0" + s to result
     end for
     for each string s in reversed(previous) do
        Append "1" + s to result
     end for
     return result
  end function
```

If the number of variables is odd, the Algorithm 4 will put one more variable in its row. In this way, the total number of variables remains the same. The resulting value is a matrix with size rows by columns.

**Algorithm 4** Construct Karnaugh Table

**Require:** Logical conditions `logicalExpressions`
**Ensure:** Generated Karnaugh Table

```
N ← size of unique elements in logicalExpressions
colsVar ← ⌊N/2⌋
rowsVar ← ⌈N/2⌉
colsSeq ← GenerateGrayCodeSequences(colsVar)
rowsSeq ← GenerateGrayCodeSequences(rowsVar)
cols ← length of colsSeq
rows ← length of rowsSeq
kTable ← array of size cols × rows, initialized to 0
for i = 0 to cols × rows - 1 do
    x ← i%cols
    y ← ⌊i/cols⌋
    colVals ← Split(colsSeq[x]) and map each value to boolean
    rowVals ← Split(rowsSeq[y]) and map each value to boolean
    kTable[i] ← EvaluateLogicalExpression(logicalExpressions, rowVals ∪ colVals)
end for
return kTable
```

### *D. K-Map Pattern Recognition*

Pattern recognition begins by defining functions that generate a specific pattern that can simplify the logics. Algorithm 5 can generate Karnaugh patterns such as square fields, horizontal fields, vertical fields, etc.

**Algorithm 5** Karnaugh Map Pattern Lookup Generator

**Require:** Number of variables `n`, logical condition pattern `v`
**Ensure:** Generated lookup fields for the Karnaugh table

```
function GenerateGroupSequences(n)
    if n = 0 then
        return empty list
    else if n = 1 then
        return list {"1", "0", "X"}
    else
        prev ← GenerateGroupSequences(n - 1)
        sequences ← concatenate:
          Append "0" to each element in prev
          Append "1" to each element in prev
          Append "X" to each element in prev
        return sequences
    end if
end function
function GenerateField(v)
    colsSeq ← GenerateGrayCodeSequences(colsVar)
    rowsSeq ← GenerateGrayCodeSequences(rowsVar)
    cols ← length of colsSeq
    rows ← length of rowsSeq
    field ← array of size cols × rows, initialized to 0
    for i = 0 to cols × rows - 1 do
        x ← i%cols
        y ← ⌊i/cols⌋
        colVals ← Split(colsSeq[x]) and map to boolean
        rowVals ← Split(rowsSeq[y]) and map to boolean
        values ← rowVals ∪ colVals
        field[i] ← true if all conditions in v are met:
          For each character c in v, check:
            If c = "X", continue
            Else, compare c to corresponding value in values
    end for
    return field
end function
```

With all the helper algorithms defined, finally the Karnaugh table can be solved. The Algorithm 6 solves Karnaugh by first generating all possible fields for each pair `x` and `y`. These fields are then sorted from the biggest to the smallest, and reduced by removing all unnecessary fields.

**Algorithm 6** Karnaugh Map Solver

**Require:** Karnaugh table `karnaughTable`, number of variables `N`
**Ensure:** Simplified Karnaugh map

```
function FieldCheck(field)
    for each element v in field do
        if v is true then
            return false if karnaughTable[i] is false
        end if
    end for
    return true
end function
function FieldSize(field)
    count ← 0
    for each element v in field do
        if v is true then
            count ← count + 1
        end if
    end for
    return count
end function
function FieldIntersect(fieldA, fieldB)
    result ← array of size rows × cols, initialized to 0
    for each index i do
        result[i] ← fieldA[i] AND fieldB[i]
    end for
    return result
end function
function FieldUnion(fieldA, fieldB)
    result ← array of size rows × cols, initialized to
```

```
0
    for each index i do
        result[i] ← fieldA[i] OR fieldB[i]
    end for
    return result
end function
fields ← empty list
for each group in GenerateGroupSequences(N) do
    field ← GenerateField(group)
    if FieldCheck(field) then
        fields ← fields ∪ field
    end if
end for
fields ← fields sorted by decreasing FieldSize
for i = fields.length - 1 down to 0 do
    joinField ← array of size rows × cols, initialized to 0
    for j = fields.length - 1 down to 0 do
        if i is not j then
            joinField ← FieldUnion(joinField, fields[j])
        end if
    end for
    intersected ← FieldIntersect(fields[i], joinField)
    if FieldSize(intersected) = FieldSize(fields[i]) then
        Remove fields[i] from fields
    end if
end for
```

### E. Code Reassemble

This part finally converts the optimized logical conditions to code. This step involves generating AST which then will be transformed into code respecting the code style guides. An alternative approach can be achieved by directly generating the code string. While the latter approach is simpler, this however, will put a burden on a developer since they have to restyle the code. This additional task can lead to increased development time and potentially introduce errors or inconsistencies, making the AST-based approach more scalable and reliable in the long run, especially for larger and more complex projects. Furthermore, the use of ASTs allows for more flexibility in code transformation, enabling automatic optimizations and modifications that may not be possible when generating raw code strings.

## V. Results and Discussion

To make the implementation more clear, let's take a look at one example. This code snippet is an example that has potential optimization for logical conditions.

Fig. 2: Code Snippet with Potential Optimizations

```
function determineAccess(user) {
    const isAdmin = user.isAdmin;
    const isEditor = user.isEditor;
    const hasPaidSubscription =
        user.hasPaidSubscription;
    const isTrialActive = user.isTrialActive;
    const isVerified = user.isVerified;

    // Complex logical condition
    // to simulate access
    if ((isAdmin && isVerified) ||
        (isAdmin && hasPaidSubscription) ||
        (isEditor && isVerified &&
            isTrialActive) ||
        (isEditor && hasPaidSubscription &&
            isVerified) ||
        (hasPaidSubscription && isTrialActive &&
            !isVerified) ||
        (!isAdmin && isEditor && isVerified)) {
        return true;
    }
    return false;
}
```

Figure 3 shows the logical condition extracted from the previous example. The variables inside the logical conditions will the get extracted and identified. Those variables are `isAdmin`, `isEditor`, `hasPaidSubscription`, `isTrialActive`, and `isVerified`. In total, there are 5 variables, meaning the Karnaugh table will have 8 rows and 4 columns. For visual reason, `isAdmin`, `isEditor`, `hasPaidSubscription`, `isTrialActive`, and `isVerified` will be referred as `A`, `B`, `C`, `D`, `E`, respectively.

Fig. 4: Grouped Karnaugh Table

| DE \ ABC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 000 | 0 | 0 | 0 | 0 |
| 001 | 0 | 0 | 0 | 1 |
| 011 | 0 | 1 | 1 | 1 |
| 010 | 0 | 1 | 1 | 0 |
| 110 | 0 | 1 | 1 | 0 |
| 111 | 1 | 1 | 1 | 1 |
| 101 | 1 | 1 | 1 | 1 |
| 100 | 0 | 1 | 1 | 0 |

Fig. 3: Logical Condition Abstract Syntax Tree

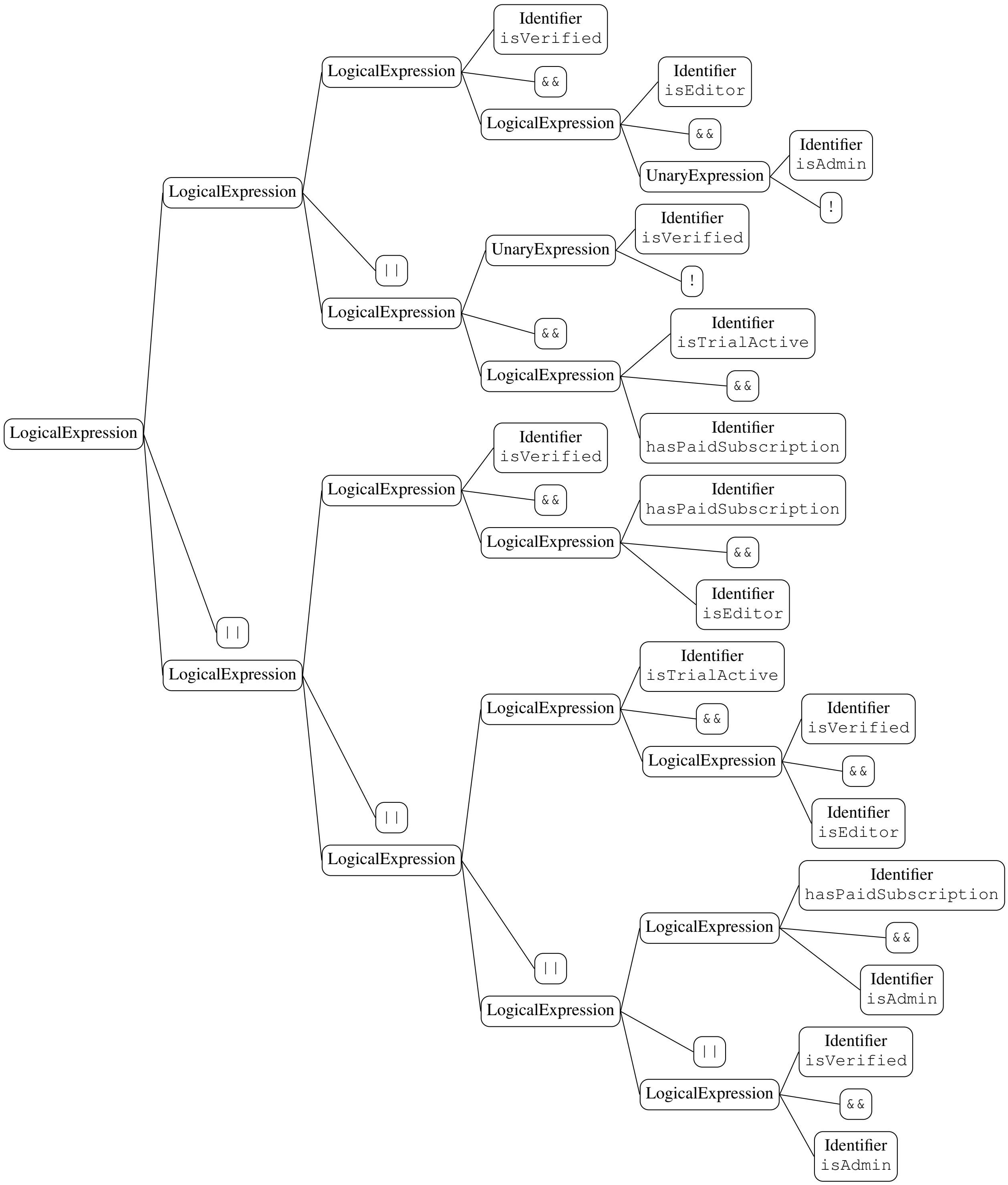

Figure 4 shows the optimized conditional logic by grouping 4 regions indicated by different colors. This result will then get reassembled to code again.

Fig. 5: Optimized Code Snippet

```
function determineAccess(user) {
    const isAdmin = user.isAdmin;
    const isEditor = user.isEditor;
    const hasPaidSubscription =
        user.hasPaidSubscription;
    const isTrialActive = user.isTrialActive;
    const isVerified = user.isVerified;

    // Complex logical condition
    // to simulate access
    if ((isEditor && isVerified) ||
        (isAdmin && isVerified) ||
        (isAdmin && hasPaidSubscription) ||
        (hasPaidSubscription && isTrialActive
            && !isVerified)) {
        return true;
    }
    return false;
}
```

The optimized logic condition can also be expressed in logic gates.

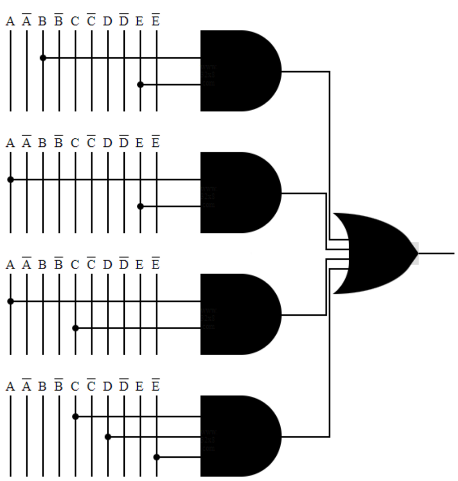


Fig. 6: Optimized Logic Gate Representation

## VI. Conclusion

In conclusion, the application of Karnaugh map logic to streamline code design presents a significant improvement in simplifying logical expressions. By integrating Karnaugh maps with abstract syntax tree (AST) parsing, this methodology enhances the readability and maintainability of code while ensuring efficient algorithm development. The process of extracting logical conditions, constructing Karnaugh tables, recognizing simplification patterns, and reassembling optimized code leads to cleaner and more performant codebases. Moreover, the use of ASTs ensures that code transformations are scalable and consistent, reducing the potential for errors. This approach not only supports the goals of maintainability but also contributes to improved developer productivity and code optimization. While the methods outlined in this paper were demonstrated through JavaScript, the principles can be generalized to other programming languages, paving the way for broader application in software engineering practices.

## Appendix

The code implementation for the methods and experiments discussed in this paper can be found at the following GitHub repository: https://github.com/NadhifRadityo/code-design-karnaugh.

Please do explore the repository for a deeper understanding of the implementation details and for any potential extensions of the methodology. for issues or questions regarding the repository, please refer to the provided documentation or contact the repository maintainer directly.

## Acknowledgment

The author would like to express profound gratitude to Dr. Ir. Rinaldi Munir, M.T. for his invaluable guidance and insights as IF1220 Discrete Mathematics course lecturer, which greatly contributed to the development of this paper.

The author also extends apologies for any shortcomings that may remain in this work. It is sincerely hoped that this paper will serve as a useful reference for future studies and research purposes.

## References

[1] Munir, R. 2023. Aljabar Boolean.
[2] Fluri, B., Wursch, M., Pinzger, M., Gall, H. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering. 33 (11): 725–743.
[3] Koschke, R., Falke, R., Frenzel, P. 2006. Clone Detection Using Abstract Syntax Suffix Trees. Working Conference on Reverse Engineering. IEEE.
[4] Karnaugh, M. 1953. The Map Method for Synthesis of Combinational Logic Circuits. Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics. 72 (5): 593–599. doi:10.1109/TCE.1953.6371932.
[5] Veitch, E. W. 1952. A chart method for simplifying truth functions. Proceedings of the 1952 ACM national meeting (Pittsburgh) on - ACM '52. Association for Computing Machinery. pp. 127–133. doi:10.1145/609784.609801.
[6] Dodge, N. B. 2015. Simplifying Logic Circuits with Karnaugh Maps. The University of Texas at Dallas, Erik Jonsson School of Engineering and Computer Science.

## 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, 31 Desember 2024

Nadhif Radityo N. (13523045)