# WHILE Modularity Analysis

Steven Shan, Hannah Sun, and Michelle Zhu

## I. INTRODUCTION

### A. *Motivation*

Modularity in code is important for maintainability, readability, and reusability. Code today is regularly updated, and having modular code isolates errors and allows for changes to be made independently of each other. Additionally, lack of modularity often results in duplicate code and redundancy that could be eliminated. Moreover, having independent functions allows clients and programmers to reuse them across different code. However, code can often become too long and/or difficult to understand by human inspection. We hope to create a tool that alleviates this burden through an analysis that groups together lines of code into snippets that could be suggested for refactoring in order to improve modularity.

## II. PROJECT DESCRIPTION AND OBJECTIVES

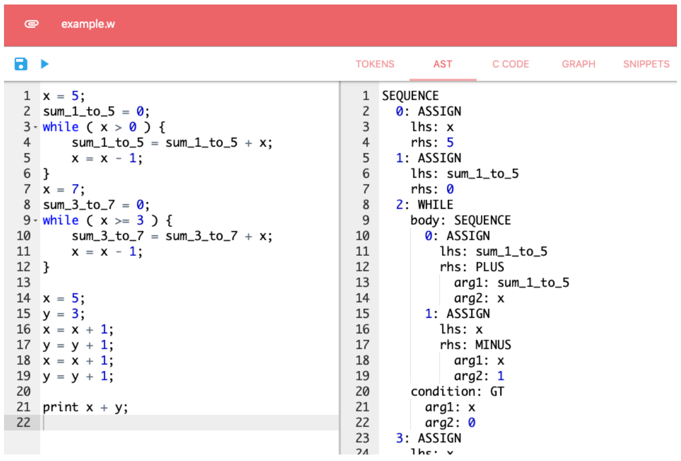
This project builds a static analysis tool that examines code without executing the program, providing groupings of code lines that could be used for refactoring to improve modularity. Our approach is to use variable access and dependencies to determine which parts of the code are suitable for refactoring. We will refer to a grouping of code lines (not necessarily contiguous) that access the same variables (either reading from or writing to) as code snippets, which are defined in detail in further on. The scope of this analysis is the WHILE language, which is defined in a subsequent section.

For this project, we wrote a web application that takes in a program written in WHILE and transpiles it to C. Our web application allows users to see how the compiler tokenizes the code, creates a abstract syntax tree, transpiles to C, creates the read-write graph, and the snippets our algorithm finds.

### A. *WHILE Language*

To limit the scope of this project to be maintainable, we decided to implement our analysis on the WHILE language, a simple programming language that was defined in class. Unlike modern programming languages, WHILE only supports the integer datatype and only has local variables. The lack of pointers and objects/structs simplifies the analysis because we do not have to worry about the side effects of each statement of code. This has some limitations that we will discuss later in the Limitations section.

Another benefit of the language is that it is easier to demonstrate the analysis. Each WHILE program could be considered to be the equivalent of the code running in the `main` function in C and each variable is globally scoped.



```
x = 5;
sum_1_to_5 = 0;
while ( x > 0 ) {
    sum_1_to_5 = sum_1_to_5 + x;
    x = x - 1;
}
x = 7;
sum_3_to_7 = 0;
while ( x >= 3 ) {
    sum_3_to_7 = sum_3_to_7 + x;
    x = x - 1;
}

x = 5;
y = 3;
x = x + 1;
y = y + 1;
x = x + 1;
y = y + 1;

print x + y;
```

```
SEQUENCE
  0: ASSIGN
    lhs: x
    rhs: 5
  1: ASSIGN
    lhs: sum_1_to_5
    rhs: 0
  2: WHILE
    body: SEQUENCE
      0: ASSIGN
        lhs: sum_1_to_5
        rhs: PLUS
          arg1: sum_1_to_5
          arg2: x
      1: ASSIGN
        lhs: x
        rhs: MINUS
          arg1: x
          arg2: 1
    condition: GT
      arg1: x
      arg2: 0
  3: ASSIGN
    lhs: x
```

Fig. 1: Screenshot of our web application

This helps show the purpose of our algorithm because it is meant to suggest refactoring targets within an excessively long function. Therefore, compared to a language such as C, we wouldn't need to include any boilerplate code.

### B. *WHILE Transpiler*

We based our implementation of the WHILE language off of the formal language syntax defined in class. The syntax only has a few minor modifications such as requiring semicolons at the end of each line and curly braces around `if` and `while` statements. The transpiler has two main components: the tokenizer and the parser.

The tokenizer uses a radix trie to match individual characters read from the source code file and output a stream of tokens, where each token is either a variable name or a keyword defined in the language rules. The token stream essentially separates the characters from the source code into groups, but does not have much meaning by itself.

The parser first uses recursive descent to build a parse tree from the stream of tokens output by the tokenizer. While the token stream has no meaning on its own, the parse tree generated by the parser directly resembles the recursive definition of the language's syntactic rules. After this step, the parser generates an abstract syntax tree (AST) from the parse tree. The AST represents higher level concepts of the program, such as whether each statement is an `if` or `while`, what its loop conditions are, and abstracts away language specific details such as whether each line ends in a semicolon or the parenthesis grouping in an expression.

The transpiler exposes an interface for our analysis to work directly with the AST so the analysis should be generalizable for languages with similar abstract features.

## III. Analysis Algorithm

### A. *Definitions*

- AST node
  - At the highest level, a program is a sequence of statements that are evaluated sequentially. Each statement in the WHILE language can either be an assignment, an `if` statement, or a `while` statement. We consider each of these to be an individual AST node. Note that there may also be a sequence of statements nested inside a `while` statement, but these are considered to be in the same AST node as the root-level `while` statement.
- Read set
  - A read set is the set of variables that an AST node reads from. For an AST node that represents a `while` or an `if` block, elements of the read set include those read in statements found within the `while` loop or the `if` block.
- Write set
  - A write set is a set of variables that an AST node writes to. For an AST node that represents a `while` or an `if` block, elements of the write set include those written to in statements found within the `while` loop or the `if` blocks.
- Read edge
  - A read edge is created for each element in an AST's read set. For each element, $r$, in the read set of an AST node, $a$, a read edge is created between $a$ and the AST node that represents the last, possible write to $r$. There may be more than one created for an element $r$ because of control flow.
- Write edge
  - A write edge is an edge between two AST nodes whose write sets share at least one element in common.
- Snippet
  - A snippet is a set of AST nodes that are on the same level of the AST graph, that depend on each other in order to correctly assign values to each variable. Snippets take into account the reads and writes of each node. It is important to note that we can find the first and last line of a snippet. The first line of the snippet refers to the line number of the AST with the smallest line number within the set of AST nodes in an given snippet. The last line of the snippet refers to the line number of the AST node with the largest line number within the set of AST nodes in a given snippet.
- Reaching definitions map
  - The reaching definitions map maps a variable, $v$ to a tuple containing the set of AST nodes, $LW$ where $v$ was last written to and the set of variables $RV$ that $v$ depends on in terms of reads. The set $LW$ only contains multiple elements if due to control flow $v$ could have been assigned differently in order to represent the possible lines from which $v$ was last written to, otherwise $LW$ is a singleton set. $RV$ is transitive in that if $x$ reads from $y$ and $y$'s $RV$ contains $z$, then $x$'s $RV$ will also contain $z$. This is achieved by updating the mapping sequentially as we iterate through the AST nodes to create read/write sets and edges (which is described in detail after the pseudocode).

### B. *Pseudocode*

```
snippet_list = new empty list
for (node in AST sequence) {
  generate read, write sets
  generate write edges
  generate read edges
  make a snippet only containing node
  add to snippet_list
}

for (node in AST sequence) {
  if (write-to != empty) {
    snippet = get_snippet(node)
    for (neighbor_node in write-to) {
      neighbor_snippet =
        get_snippet(neighbor_node)
      merge(snippet, neighbor_snippet)
    }
  }
}

for (node in AST sequence) {
  if (read-from != empty) {
    snippet = get_snippet(node)
    for (neighbor_node in read-from) {
      neighbor_snippet =
        get_snippet(neighbor_node)
      if (neighbor_node.get_line() >
                snippet.first_line() ||
          neighbor_node.last_line() >
                node.get_line()) {
        merge(snippet, neighbor_snippet)
      }
    }
  }
}
return snippet_list
```

Fig. 2: Pseudocode of our algorithm

We will now go more in depth into each section of algorithm.

### C. *Creating Read/Write Sets*

In order to create the read and write sets, we start by scanning the AST nodes. We create two sets for each node, the write-to set and the read-from set. The write-to set includes all variables the AST node modifies or reassigns, while the read-from set includes all the variables that the AST node reads from but does not modify. For `ASSIGN`, we simply put the written variable (the variable to the left of the =) in the write set, and all variables that appear in the right hand side of the = into the read set. For `WHILE` and `IF`, we recursively enter the block and union the read and write sets of all subsequent lines inside the block respectively.

## D. Read/Write Edges

After the write-to and read-from sets are created for each AST node recursively, we use them to create edges between the AST nodes. Edges are created only between AST nodes of the same level (for instance, an `ASSIGN` node on the same level as a `WHILE` node will not have edges to nodes inside of the `WHILE` block). As we iterate through the AST nodes sequentially and recursively, we use the reaching definitions mapping previously defined (that is constantly updated) to create read and write edges. Essentially, a write edge is created between two AST nodes if they share a variable in their write sets and a read edge is created between them if they share a variable in their read sets (one edge is created for each variable shared).

For `ASSIGN`, we first create the read edges. For each variable $v$ on the right hand side on the assignment, we create read edges to each node in $LW$ set for $v$ (in the reaching definitions map). While doing this, we also merge all the nodes in the $RV$ set for $v$ into the current node's dependencies.

In terms of write edges, there is a special case for `ASSIGN`; when the right hand side is only a numerical constant, we define the left hand side as a new variable, since it does not have any dependencies, even if the variable previously existed. This allows us to create more independent pieces of code. If the dependencies include the current node itself, we know it then depends on a previous version of itself, and is not a new variable. Thus if the dependencies of the current node includes itself, we create a write edge to all the nodes in the $LW$ set of the variable being written to in the assignment (on the left hand side) – since those are the other nodes that previously modified the same variable.

For `IF` and `WHILE`, we essentially do the same as `ASSIGN` for read edges. We create a read edge to each node in the $LW$ for each variable $v$ read in the loop body or the condition body (for `IF` this includes the code under both `IF` and `ELSE`). For write edges, we also follow the general steps from `ASSIGN`, except there may be multiple variables now in the write-to set of the AST node. For each of these variables, we find its $RV$ set. If the current AST node is in this set, then we find the variable's $LW$ set, and create write edges from the current AST node to all the nodes in that $LW$ set.

In each case, we also update the reaching definitions map. For each variable definitely written to, we set its $LW$ to the singleton set of the current node. In cases where we may or may not write to a variable (due to control flow), we add the current node into that variables $LW$ set. The $RV$ set for each variable is simply merged with the new dependencies in the current node.

## E. Creating Snippets

For each AST node, we create an individual snippet. So, to begin with a snippet only contains one AST node.

## F. Merging Snippets

Once we have created a snippet for each node, we first examine the write edges of all nodes. For any two nodes connected by a write edge, we will `merge` the two snippets that nodes are in together. `merge` takes in two snippets and will return one new snippet that contains all union of all the AST nodes in the two snippets fed in as parameters. We want to merge snippets that share a write edge between nodes within in the snippets because one of the future goals of creating snippets is to find lines of code that could be taken and made into a helper function to increase modularity and grouping lines of code that modify and redefine a certain variable would help us create a helper function that performs all needed operation on the variable within the same function. This is essentially what a write edge is, a connection between two lines that both modify the same variable. While we are not creating helper functions in our algorithm, we are looking to set up potential snippets of code that could be modularized (especially after comparison to other snippets to see if the two snippets are functionally equivalent, possibly with different variables).

After merging snippets based on write edges, we examine the read edges of each node. For every node $n$, we check each node $r$ in $n$'s read set. Let $ns$ and $rs$ represent $n$'s and $r$'s corresponding snippet respectively. We then check if the $r$'s line is greater than the first line of $ns$ or if $s$'s line less than $rs$'s last line, if either of these two predicates are true, we merge $ns$ and $rs$. When either of these conditions is true, we know that there is a dependency between $n$ and some node in $rs$ that dictates $n$ must always be in the relative order to $r$ as it currently is. Since there is a dependency, we must combine the snippets to preserve correctness of the program. We will examine the two examples of conditions to see why these two cases should be in the same snippet.

```
1   k = 2;
2   x = factorial(k);
3   k = k + 1;
```

In this example, while there is no function calls in WHILE language we can imagine that the call `factorial(k)` is a series of code lines that are all in one code snippet. Our graph would contain a write edge from line 1 to line 3 and a read edge from line 2 to line 3. This is the case where $s$'s line less than $rs$'s last line. It is important to note the dependency of the relative location of line 2 since the $k$ that line 2 reads must come from line 1 and not line 3 in order to calculate `factorial(2)` and not `factorial(3)`. Thus, all three lines must end in the same code snippet in order to preserve correctness.

```
1    y = 1;
2    x = y;
3    y = y + 1;
4    x = x + y;
```

In this example, we examine the case where $r$'s line is greater than the first line of $ns$. There are write edges between line 1 and line 3, and between line 2 and line 4. The read edges are between line 1 and line 2, between line 1 and line 3, between line 2 and line 4, and between line 3 and line 4. After merging from write edges, we will have two snippets; one containing line 1 and line 3 and a second containing line 2 and line 4. Once the algorithm detects the read edge between line 3 and 4, it recognizes that the two snippets must be merged. In this case, we want to merge because there is a dependency of line 4 and line 3 and it must be that line 3 occurs before line 4 but also after line 2 in order to preserve correctness of the program. So this dependency on relative ordering between the two snippets necessitates that this example becomes one snippet, which is why we need to merge the two snippets.

In general, these two conditions find snippets whose AST node line numbers have relative order that must be preserved in order to maintain functional correctness. We do this for all the individual snippets that begin as one AST node, and produce our final suggestions after the merging process is complete.

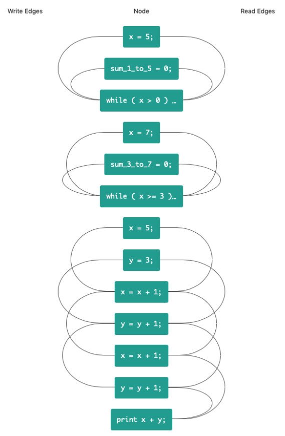
## IV. RESULTS & DISCUSSION

In this section, we will examine an example and our analysis's outputted snippets.

### A. *Example 1*

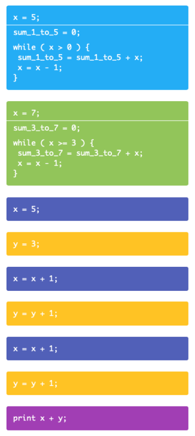
```
1    x = 5;
2    sum_1_to_5 = 0;
3    while ( x > 0 ) {
4     sum_1_to_5 = sum_1_to_5 + x;
5     x = x - 1;
6    }
7    x = 7;
8    sum_3_to_7 = 0;
9    while ( x >= 3 ) {
10    sum_3_to_7 = sum_3_to_7 + x;
11    x = x - 1;
12   }
13
14   x = 5;
15   y = 3;
16   x = x + 1;
17   y = y + 1;
18   x = x + 1;
19   y = y + 1;
20
21   print x + y;
```

In this example, we can examine the program manually and are able to deduce that the first 6 lines simply sum the integers 1 through 5, lines 6 through 12 sum the integers 3 though 7, lines 14-19 simply set $x$ and $y$ and alternately and independently increment $x$ and $y$ (so in alternating lines there are two independent sections of code), and then line 21 prints the sum of $x$ and $y$. Each of the code sections we just described are independent of each other and should therefore be considered as snippets. Our analysis finds the following edges:



```
x = 5;
sum_1_to_5 = 0;
while ( x > 0 ) …
x = 7;
sum_3_to_7 = 0;
while ( x >= 3 )…
x = 5;
y = 3;
x = x + 1;
y = y + 1;
x = x + 1;
y = y + 1;
print x + y;
```

(a) Write/Read Graph



```
x = 5;
sum_1_to_5 = 0;
while ( x > 0 ) {
 sum_1_to_5 = sum_1_to_5 + x;
 x = x - 1;
}

x = 7;
sum_3_to_7 = 0;
while ( x >= 3 ) {
 sum_3_to_7 = sum_3_to_7 + x;
 x = x - 1;
}

x = 5;

y = 3;

x = x + 1;

y = y + 1;

x = x + 1;

y = y + 1;

print x + y;
```

(b) Snippets Found

Fig. 3: Results of analysis on Example 1

We can see that the analysis is able to distinguish the dependencies between lines and correctly identify lines 1-6; lines 7-12; lines 14, 16, 18; lines 15, 17, 19; line 21 as snippets. It is important to note that snippets do not necessarily have to contiguous lines of code as seen with the alternating snippet block seen in lines 14-19. Another important detail is that line 21 is its own independent snippet because while does read from $x$ and $y$ defined in lines 18 and 19 it does not modify them and simply accesses. We determined that such relationships that only read and do not modify should not necessitate that the lines must be in the same blocks as this would create overly large snippets. Thus, this was a subjective optimization we made in order to have more meaningful snippets that could be used in future refactorings. In this example, we can also see that while $x$ is used throughout the program, it does not make the entire program into a singular snippet. This is because every time a variable is assigned to a constant we redefine a new variable since there are no previous dependencies, thus allowing for more modularity and less unnecessary dependencies that would create larger snippets.

More examples and intended graph and snippet generation can be found in the appendix.

## V. USAGE OF OUR TOOL

https://github.com/hannah-sun/while-modularity-analysis

See the above link for how to use the web application or how to use our tool locally.

## VI. LIMITATIONS

Currently, our approach only accomplishes the first step of the original goal: to find the snippets of the code that could potentially be separated or refactored into separating functions. Thus, as is, our analysis does not actually refactor the code; it simply generates potential snippets. For large programs, the examination of the output of our current analysis would still require a large amount of human labor in order to determine based on the snippets which parts of the program could be refactored and possibly which snippets could be made into separate functions. Moreover, we are currently working with a small subset of the C language that does not include memory allocation and usage, function declarations and calls, and control flow (i.e. `break, return`).

While we wrote and tested many examples to determine whether or not our algorithm was creating snippets the way we anticipated they would be created, we did not explicitly write a testing suite to continually run whenever we made changes to the algorithm. However, since we were developing an algorithm to create snippets and snippets were an idea we created, we continued to refine the definition of a snippet throughout the process and discovered that sometimes our algorithm found a more precise manner to create snippets than we could. One reason this was the case is that finding "snippets" and a way to partition the program is subjective and often the computer was able to be more objective than us. In this way, without having set test case expected output, we were actually able to refine our definition of a snippet and clarify exactly what we wanted to be considered to be a snippet.

## VII. FUTURE

In the future, we would considering limiting the number of lines of code that could be in a snippet, since many one or two line snippets breaks the code up too much or having only one or two large snippets breaks the code up too little and defeats the purpose of trying to find lines of code to possibly refactor. As mentioned in the Limitations section, our analysis can only be performed on a small subset of C transpiled from the WHILE language. It would be useful to extend our analysis to a larger subset of the C language to include function definitions, memory usage, and control flow beyond `if/else` statements.

Another future step would be to refactor the code based on the snippets the analysis currently finds. A basic, simple way to do this would be to compare the every snippet with every other snippet and check the operations that each snippet performs to see if the snippets are computing the same thing. This could also be improved to be more efficient.

As discussed in the Limitations section, snippet detection and creating modularity is subjective. Thus, finding a standardized metric for testing and evaluation of the suggested snippets would be another further step to take for this project.

In our algorithm currently, we only consider variable access and usage; however we could also consider creating snippets in terms of the specific functionality of each line. In other words, we could compare the operations performed in the code as an additional way to increase preciseness of the creation of snippets.

## VIII. Related Research and Tools

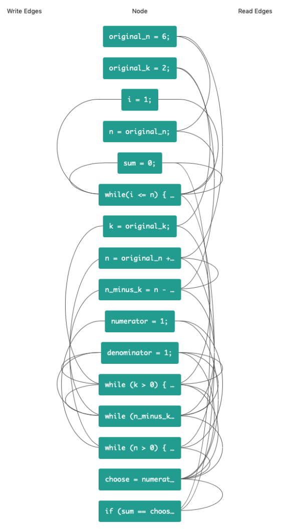
Extract method is one of the main approaches for making code more modular. It moves independent code to a separate new method/function and replaces the old code with a call to that function. While we are not implementing refactoring in our prototype (we are only identifying the independent pieces of code that could be extracted), this is similar to our goal. Currently, some IDEs support various degrees of refactoring by extract method, or have certain packages/plugins to download. However, many of these require that the programmer self identify the lines of code to extract, and only provide support for that process. For larger projects, it would be burdensome for the user to identify each case of possible method extraction themselves. Our goal here is to automate that process and be able to find candidates within the code for the user through static analysis. Currently, there a few approaches in the field: using program slicing, graph representations, scoring functions, and refactoring prioritization. Program slicing is difficult to fully automate because it typically requires user input to create the slicing criteria [1]. Scoring functions typically focus on length of code and and how many layers there are, ranking more complex code higher for extraction [1]. Prioritization focuses on which parts should be refactored first, so it is also not suitable for automation, since it doesn't present the full picture.

We used a graph representation of the code, since we wanted to focus on variable dependencies in order to find relatively independent snippets. Other research that uses graph representations also look at dependencies, such as the methods suggested by Sharma [3] and Kanemitsu et al. [2]. Sharma uses the length of dependency edges to determine where snippets or blocks of code should be split, whereas Kanemitsu et al. extracts all connected components. Our approach, like theirs, can extract non continuguous lines of code, but we differ from them because we used read and write edges, instead of generalized edges. Additionally, Sharma's approach chose to delete edges to create snippets, whereas we started with small snippets and merged them together to create larger ones instead of breaking up existing ones.

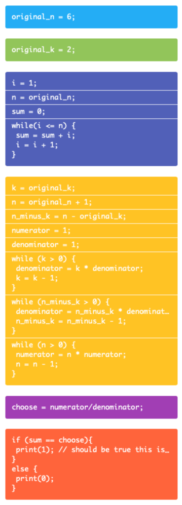
## APPENDIX

### A. *Example 2: Counting in two ways*

```
//counting in two ways
original_n = 6;
original_k = 2;

// first way to count
i = 1;
n = original_n;
sum = 0;

while(i <= n) {
    sum = sum + i;
    i = i + 1;
}

// second way to count
k = original_k;
n = original_n + 1;
n_minus_k = n - original_k;
numerator = 1;
denominator = 1;

// should calculate the denominator of n choose k
while (k > 0) {
    denominator = k * denominator;
    k = k - 1;
}

while (n_minus_k > 0) {
    denominator = n_minus_k * denominator;
    n_minus_k = n_minus_k - 1;
}

// should calculate the numerator of n choose k
while (n > 0) {
    numerator = n * numerator;
    n = n - 1;
}

choose = numerator/denominator;

// check if the two ways of counting are equivalent
if (sum == choose){
    print(1);
    // should be true this is the binomial theorem
}
else {
    print(0);
}
```



(a) Write/Read Graph



```
original_n = 6;
```
```
original_k = 2;
```
```
i = 1;
n = original_n;
sum = 0;
while(i <= n) {
 sum = sum + i;
 i = i + 1;
}
```
```
k = original_k;
n = original_n + 1;
n_minus_k = n - original_k;
numerator = 1;
denominator = 1;
while (k > 0) {
 denominator = k * denominator;
 k = k - 1;
}
while (n_minus_k > 0) {
 denominator = n_minus_k * denominat…
 n_minus_k = n_minus_k - 1;
}
while (n > 0) {
 numerator = n * numerator;
 n = n - 1;
}
```
```
choose = numerator/denominator;
```
```
if (sum == choose){
 print(1); // should be true this is…
}
else {
 print(0);
}
```

(b) Snippets Found

Fig. 4: Results of analysis on Example 2

## References

[1] Haas, R., & Hummel, B. (2015). Deriving Extract Method Refactoring Suggestions for Long Methods.

[2] Kanemitsu, T., Higo, Y., Kusumoto, S.: A Visualization Method of Program Dependency Graph for Identifying Extract Method Opportunity. In: Proceedings of the 4th Workshop on Refactoring Tools, pp. 814. ACM (2011)

[3] Sharma, T.: Identifying Extract-method Refactoring Candidates Automatically. In: Proceedings of the 5th Workshop on Refactoring Tools, pp. 5053. ACM (2012)