Purpose of the Project

This project aims to optimize the given intermediate code by applying compiler optimization algorithms to make it more efficient. The main objectives of the project are as follows:

1. Implementing **Constant Folding** and **Constant Propagation** algorithms on the intermediate code.
2. Detecting and removing unnecessary expressions after code optimization (e.g., **Dead Code Elimination**).
3. Performing more advanced optimizations by applying the **Algebraic Simplification** algorithm.

To achieve these objectives, a system for analysis and optimization has been designed using **lex** and **yacc** tools. This system aims to analyze the given intermediate code and transform it into a faster, more readable, and resource-efficient form.

What is a Compiler?

A compiler is software that translates source code written in a programming language into another language, typically machine code or intermediate code. This process is essential for the computer to understand and execute the written program. The compiler performs the translation in multiple stages:

1. **Lexical Analysis**: This stage breaks the code into smaller components called tokens.
2. **Syntax Analysis**: Checks if the tokens comply with the rules of the language and generates a syntax tree.
3. **Semantic Analysis**: Ensures variables are defined and checks type compatibility.
4. **Intermediate Code Generation**: Produces an intermediate representation before translating it into machine code.
5. **Optimization**: Improves the code's performance and removes unnecessary operations.
6. **Code Generation**: Produces the final code that can be understood by the machine or the target platform.

What is Compiler Optimization?

Compiler optimization refers to the process of improving the generated code to increase its speed, reduce memory consumption, and enhance resource efficiency. These optimizations impact both the runtime performance of the code and the compilation process. Optimization techniques are generally categorized into two types:

1. **Local Optimization**
This type of optimization is applied to a specific code block or a small section of code, considering only the logic and data within that block. Examples include:

- **Constant Folding**: Computing the results of operations with constant values during compilation.
- **Copy Propagation**: Replacing variables with their previously assigned values.
- **Dead Code Elimination**: Removing unused code.

2. **Global Optimization**
This type of optimization considers the entire program or larger sections of code. Examples include:

- **Loop Unrolling**: Optimizing loops to reduce execution overhead.
- **Global Common Subexpression Elimination**: Removing redundant expressions across the program.

What are Local Optimization Algorithms?

Local Optimization Algorithms refer to optimization processes applied to a specific region or block of code within a program. These algorithms focus solely on the expressions and operations within that particular block, without considering the overall structure of the program. As a result, the code becomes more efficient, faster, and consumes fewer resources.

The local optimization algorithms used in my project are as follows:

1. Constant Folding
Constant Folding is an optimization technique that evaluates operations based on constant values during the compilation process and replaces these operations with their results. This optimization aims to make the program run faster by performing calculations at compile time instead of runtime.
Example:
Input Code:
```
x = 2 + 3;
y = 5 * 4;
```
After Constant Folding:
```
x = 5;
y = 20;
```

2. Copy or Constant Propagation
Copy Propagation refers to the process of replacing a variable with its assigned value (a constant or another variable) in other parts of the code. This optimization eliminates unnecessary variables and produces simpler, more readable code.
Example:
Input Code:
```
a = 5;
x = a + 2;
```

After Constant Propagation:

```
x = 5 + 2;
```

After Applying Constant Folding:

```
x = 7;
```

3. Algebraic Simplification

This algorithm simplifies algebraic operations or eliminates meaningless expressions. By optimizing mathematically unnecessary statements, the code's readability and performance are improved.

Example:

Input Code:

```
x = x + 0;
y = y * 1;
z = z * 0;
```

After Algebraic Simplification:

```
// x = x + 0; and y = y * 1; are removed.
z = 0;
```

4. Dead Code Elimination

Dead Code Elimination removes code that has no effect on the program's execution or is unused. This optimization cleans up unused variables and unnecessary assignments, making the code more efficient and compact.

Example:

Input Code:

```
a = 5;
x = a + 2;
y = 3;
z = y + 4;
```

After Dead Code Elimination:

```
a = 5;
x = a + 2;
z = 7; // y is removed because it is unused.
```

Implementation

1. Constant Folding Implementation

Constant Folding is an optimization technique that calculates operations between constants at compile time and replaces them with their constant results. The part of the code where this operation is applied is defined in the expression production rules.

Code Snippet

```
expression:
    NUMBER PLUS NUMBER {
        $$.value = $1 + $3;  // Calculate the sum of two constants.
        asprintf(&$$.expr, "%d", $$.value);  // Replace the expression
with the constant value.
    }
    | NUMBER MINUS NUMBER {
        $$.value = $1 - $3;
        asprintf(&$$.expr, "%d", $$.value);
    }
    | NUMBER MULTIPLY NUMBER {
        $$.value = $1 * $3;
        asprintf(&$$.expr, "%d", $$.value);
    }
    | NUMBER DIVIDE NUMBER {
        if ($3 == 0) {
            yyerror("Division by zero");  // Check for division by zero
error
        } else {
            $$.value = $1 / $3;
            asprintf(&$$.expr, "%d", $$.value);
        }
    }
;
```

Explanation

1. **Defining Constants**:
These rules apply to NUMBER tokens, which represent constant integer values.

2. **Performing Operations**:
Basic operations such as addition (PLUS), subtraction (MINUS), multiplication (MULTIPLY), division (DIVIDE) and exponantial (EXPONENT)are defined.

- The constants are replaced by a single constant result after the operation. For example:

• **Input**: x = 3 + 5;

• **Output**: x = 8;

3. **Updating the Result**:

- The result of the operation is stored in $$.value and can be used in subsequent operations.
- The asprintf function formats the optimized expression into a text format, such as "8", representing the constant value.

4. **Error Handling**:

- A division by zero error is checked during division operations to ensure safety.

Example Input and Execution Flow

Input:

```
x = 4 + 2;
y = 3 * 5;
z = 10 / 2;
```

1. x = 4 + 2;

- The rule NUMBER PLUS NUMBER is triggered.
- Result: $$.value = 6 → Output: x = 6;

2. y = 3 * 5;

- The rule NUMBER MULTIPLY NUMBER is triggered.
- Result: $$.value = 15 → Output: y = 15;

3. z = 10 / 2;

- The rule NUMBER DIVIDE NUMBER is triggered.
- Result: $$.value = 5 → Output: z = 5;

Final Result

```
x = 6;
y = 15;
z = 5;
```

This implementation optimizes the expressions by computing constant operations at compile time, reducing runtime calculations and improving performance.

2. Copy or Constant Propagation Implementation

Copy Propagation is applied when a variable is assigned the value of another variable. In this implementation, the process is done through a symbol table, which keeps track of variable definitions and their values.

Code Snippet

```
expression:
    ID {
        Node *node = lookup($1);  // The variable is looked up in the
symbol table.

        if (node) {
            $$.value = node->value;  // The value of the variable is
retrieved.
            asprintf(&$$.expr, "%s", $1);  // The variable's name is
stored.
            node->usage_count++;  // The usage count of the variable is
incremented.
        } else {
            $$.value = 0;  // Default value
            asprintf(&$$.expr, "%s", $1);
        }
    }
;
```

Explanation

1. **Using the Symbol Table**:

- The lookup function checks if the variable is defined in the symbol table.
- If the variable is found, its value is retrieved.

2. **Value Propagation**:

- When a variable is assigned the value of another variable, the value from the symbol table is propagated.
- Example:

• **Input**: a = b;

• **Propagation**: The value of b is used in place of a.

3. **Updating the Usage Count**:

- The usage_count is incremented for the variable, which helps track how many times each variable is used.

Example Input and Execution Flow

Input:

```
a = 10;
```

```
b = a;
c = b + 5;
```

Steps:

1. a = 10;

- a is added to the symbol table with the value 10.

2. b = a;

- The symbol table is checked for a, and its value is found (10).
- b is assigned the value 10 from a.

3. c = b + 5;

- The symbol table is checked for b, and its value is found (10).
- The calculation c = 10 + 5 is performed, resulting in c = 15.

Final Result

```
a = 10;
b = 10;
c = 15;
```

In this implementation, **Copy Propagation** ensures that once a variable is assigned the value of another, the value is propagated throughout the code, allowing for optimizations such as reducing unnecessary variable assignments.

3. Algebraic Simplification Implementation

Algebraic Simplification optimizes code by removing or simplifying mathematically redundant or ineffective expressions. In this implementation, the process is applied in the statement production

rules, where certain operations are checked for mathematical ineffectiveness, and optimized outputs are produced.

Code Snippet

```
statement:
    ID ASSIGN expression {
        Node *node = lookup($1);

        if (node) {
            // If the same value is assigned again, this operation is
meaningless.
            if ($3.value == node->value) {
                printf("Deleted: %s = %s; Because the code has no
meaningful effect.\n", $1, $3.expr); // Algebraic Simplification
            } else if (valid) {
                insert($1, $3.value); // The new value is added to the
symbol table.
                printf("%s = %s;   %d\n", $1, $3.expr, $3.value);
            }
        } else if (valid) {
            insert($1, $3.value); // If the variable is used for the
first time, it is added to the symbol table.
            printf("%s = %s;   %d\n", $1, $3.expr, $3.value);
        }

        valid = 1; // Reset the error check flag.
    }
;
```

Explanation

1. **Checking for Meaningless Assignments**

- If a variable is assigned its own value or if the operation has no mathematical effect (such as x = x + 0), the operation is removed.
- Example:

• x = x;

• x = x + 0;

These assignments are deemed redundant and are deleted.

2. **Updating Symbols and Values**

- If the expression cannot be simplified, the new value is added to the symbol table, and the optimized version of the assignment is kept in the code.

3. **Using the Symbol Table**

- The lookup function checks if the variable is in the symbol table.

- If the variable exists, it compares the new value with the old value to see if any simplification can be made.

Simplification Logic

Algebraic Simplification applies the following patterns to simplify the code:

1. **Removing Ineffective Operations**

- x = x + 0; → x
- x = x * 1; → x

2. **Simplifying Meaningless Operations**

- x = x * 0; → x = 0
- y = y ^ 2; → y = y * y

3. **Eliminating Redundant Assignments**

- x = x;
- x = 5; x = 5;

These assignments are removed as they don't change the state of the program.

Example Input and Execution Flow

Input:

```
a = a + 0;
b = b * 1;
c = c * 0;
d = d ^ 2;
e = e;
```

Steps:

1. a = a + 0;

- The condition x = x + 0 is identified as redundant, so this statement is removed.

2. b = b * 1;

- The condition x = x * 1 is identified as redundant, so this statement is removed.

3. c = c * 0;

- The condition x = x * 0 is simplified to c = 0.

4. d = d ^ 2;

- The expression x = x ^ 2 is simplified to d = d * d.

5. e = e;

- The assignment e = e is redundant and removed.

Final Result

```
c = 0;
```

```
d = d * d;
```

Expression Patterns for Algebraic Simplification

In the code snippet, algebraic simplifications are applied through specific patterns for addition and multiplication in expressions:

1. Addition Simplification (PLUS)

```
expression PLUS expression {
    if ($1.value == 0) {  // x + 0 → x
        $$.value = $3.value;
        asprintf(&$$.expr, "%s", $3.expr);
    } else if ($3.value == 0) {  // 0 + x → x
        $$.value = $1.value;
        asprintf(&$$.expr, "%s", $1.expr);
    } else {
        $$.value = $1.value + $3.value;
        asprintf(&$$.expr, "%s + %s", $1.expr, $3.expr);
    }
}
```

• **Optimization**:

- x + 0 → x: If the first operand is 0, the result is just the second operand.
- 0 + x → x: If the second operand is 0, the result is just the first operand.

2. Multiplication Simplification (MULTIPLY)

```
expression MULTIPLY expression {
    if ($1.value == 0 || $3.value == 0) {  // x * 0 → 0
        $$.value = 0;
        asprintf(&$$.expr, "0");
    } else if ($1.value == 1) {  // x * 1 → x
        $$.value = $3.value;
        asprintf(&$$.expr, "%s", $3.expr);
    } else if ($3.value == 1) {  // 1 * x → x
        $$.value = $1.value;
        asprintf(&$$.expr, "%s", $1.expr);
    } else {
        $$.value = $1.value * $3.value;
        asprintf(&$$.expr, "%s * %s", $1.expr, $3.expr);
    }
}
```

• **Optimization**:

- x * 0 → 0: Multiplying by 0 results in 0.
- x * 1 → x: Multiplying by 1 results in the original value.
- 1 * x → x: Multiplying by 1 results in the original value.

Summary of the Algebraic Simplification Process:

1. **Eliminates Ineffective Operations**:

- For example, x + 0 and x * 1 are removed or simplified.

2. **Simplifies Redundant Operations**:

- For example, x * 0 is simplified to 0.

By following these patterns, the code is optimized by removing unnecessary operations and simplifying others to make the program more efficient and easier to read.

Dead Code Elimination (DCE) Implementation

Dead Code Elimination is an optimization technique used to identify and remove code that has no effect on the program's execution. In the project, this process is implemented using a **symbol table** to track the usage of variables and eliminate those with no consumers.

Code Implementation

```
void print_table() {
    // Display Symbol Table
    printf("\nSymbol Table:\n");
    printf("----------\n");
    for (int i = 0; i < symbol_count; i++) {
        if (symbol_table[i].usage_count != 0) {
            printf("| %s = %d (Usage: %d)\n",
                symbol_table[i].id,
                symbol_table[i].value,
                symbol_table[i].usage_count);
        }
    }
    printf("----------\n");

    // Identify and Display Dead Code
    printf("\nEliminated by Dead Code Elimination\n");
    printf("----------\n");
    for (int i = 0; i < symbol_count; i++) {
        if (symbol_table[i].usage_count == 0) { // Dead Code Check
            printf("| %s = %d (No consumers)\n",
                symbol_table[i].id,
                symbol_table[i].value);
        }
    }
    printf("----------\n");
}
```

Explanation of Implementation

1. **Tracking Variable Usage**:

- The **symbol table** is used to maintain a record of all variables and their usage count (usage_count).
- Every time a variable is consumed (used in an expression or statement), its usage_count is incremented.

2. **Identifying Dead Code**:

- Variables with usage_count == 0 are considered **dead code** because they are assigned a value but are never used anywhere in the program.

3. **Eliminating Dead Code**:

- Variables with no consumers are flagged and excluded from the optimized version of the code.

4. **Debugging and Logging**:

- The print_table function lists both:

• **Active variables**: Those used in the program, along with their usage counts.

• **Dead variables**: Those that are never used, marked as "Eliminated by Dead Code Elimination."

Example Input and Execution

Input Code:

```
x = 5;   // Dead code (not used anywhere)
y = 10;  // Used
z = y + 2; // Dead code (not used anywhere)
```

Symbol Table (Before Optimization):

Variable	Value	Usage Count
x	5	0
y	10	2
z	12	0

Output:

```
Symbol Table:
----------
| y = 10 (Usage: 2)
----------
Eliminated by Dead Code Elimination
----------
| x = 5 (No consumers)
| z = 12 (No consumers)
----------
```

Steps of Dead Code Elimination

1. **Symbol Table Construction**:

- Variables are added to the symbol table during assignment.
- Their usage_count is updated whenever they are referenced.

2. **Optimization Process**:

- Variables with usage_count == 0 are removed as dead code.
- Remaining variables are included in the optimized output.

3. **Final Optimized Code**:

```
y = 10;
```

Benefits of Dead Code Elimination

1. **Reduced Code Size**:

- Removes unnecessary variables, leading to smaller and more efficient code.

2. **Improved Performance**:

- Eliminates redundant memory usage and processing.

3. **Better Readability**:

- Simplifies code by removing unused and irrelevant sections.

How to Run?

While in the project directory, you can run the code by using the following command in the terminal:

```
bash run.sh
```

You can then view the output of the program in the input.txt file.