



FALL - 2024

CSE351 - PROGRAMMING LANGUAGES

TERM PROJECT REPORT

NAME SURNAME :	ULAŞ CAN DEMİRBAĞ
ID :	20200702119
INSTRUCTOR :	Prof. Dr. GÜRHAN KÜÇÜK

Table of Contents

1. Introduction	4
2. Explanation of Algorithms	5
2.1 Constant Folding	5
2.2 Constant Propagation	5
2.3 Algebraic Simplification.....	5
2.4 Subexpression Elimination	6
2.5 Dead Code Elimination	6
3. Design and Implementation	7
3.1 Overview of the Folder Structure	7
3.2 Implementation Details for Each Algorithm	8
3.2.1 Constant Folding	8
3.2.2 Copy or Constant Propagation	10
3.2.3 Algebraic Simplification	11
3.2.4 Common Subexpression Elimination	13
3.2.5 Dead Code Elimination	14
4. Testing and Results	16
4.1 Test Inputs and Expected Outputs	16
4.2 Makefile to Run All Tests	18
4.3 Observations	18
5. Conclusion.....	19
6. References.....	20

Table of Figures

Figure 1 Folder Structure of Term Project	7
Figure 2 Helper Function is_number for Constant Folding	8
Figure 3 Prog and assgmt rules for Constant Folding	8
Figure 4 General rule for operators in Constant Folding	8
Figure 5 Special rule for the power operator in Constant Folding	8
Figure 6 Implentation unordered_map	10
Figure 7 assgmt rule for Copy or Constant Propagation	10
Figure 8 exp rule for Copy or Constant Propagation	10
Figure 9 assgmt rule for Algebraic Simplification Part 1	11
Figure 10 assgmt rule for Algebraic Simplification Part 2	12
Figure 11 Implementation of exps_map	13
Figure 12 assgmt rule of Common Subexpression Elimination	13
Figure 13 Function to eliminate dead code.	14
Figure 14 Helper function to extract identifiers.	14
Figure 15 Rule for parsing assignments.	15
Figure 16 Main function calling dead code elimination.	15
Figure 17 Makefile of Constant Folding.....	16
Figure 18 Makefile to run all tests.....	18
Figure 19 Execution of Make Check and Make Clear Rules.....	18

Table of Tables

Table 1 Inputs and Expected Outputs of Constant Folding	16
Table 2 All tests and result	17

1. Introduction

Optimizing compilers apply a series of source code transformations (or its intermediate representations) to maximize runtime efficiency, to minimize code size, and to reduce its complexity. This project considers five fundamental compiler optimization techniques, each implemented as a separate module using Flex for lexical analysis and Yacc/Bison for parsing and transformation. The optimizations done are:

1. **Constant Folding**
2. **Copy or Constant Propagation**
3. **Algebraic Simplification**
4. **Common Subexpression Elimination**
5. **Dead Code Elimination**

For each technique, we implemented specific parsers and transformers and tested them on sample inputs in order to prove their correctness by comparing the obtained results with the expected outputs. The structure of the report is as follows:

- **Section 2:** Provides an overview of the concepts underlying each of the five optimization techniques.
- **Section 3:** Outlines the design and implementation process, including the role of the lexer, parser, and the specific approach for each optimization.
- **Section 4:** Describes the testing methodology and presents the results obtained from the sample cases.
- **Section 5:** Concludes with a summary of the project's findings.

This report demonstrates real-life application and effectiveness of such optimization techniques, and thus it adds to a better understanding of compiler design and functionality.

2. Explanation of Algorithms

2.1 Constant Folding

Constant Folding is a technique of optimization where constant expressions are evaluated at compile time instead of runtime. In a statement like $a = 2 + 3$;, the compiler replaces $2 + 3$ with 5. This saves the computer from doing unnecessary calculations when the program is executed.

Key Concept: If both operands of an operation are constants, then evaluate the result at compile-time and replace the expression with the computed value.

2.2 Constant Propagation

Constant Propagation. This optimization replaces variables by their known constant values at compile time. For example, if a variable is assigned a constant ($d = 4$;) and later used in an expression ($c = d + 5$;) , the compiler may substitute d by 4, making the expression $c = 4 + 5$;

Key Concept: Maintain a record of variables assigned constant values and substitute their occurrences in expressions by the corresponding constants.

2.3 Algebraic Simplification

Algebraic Simplification: This pattern uses simple mathematical identities to simplify expressions. Operations such as adding or subtracting zero, multiplying by one, or similar adjustments can be used to lower the complexity of code:

Examples:

- $x + 0 \rightarrow x$
- $x - 0 \rightarrow x$
- $x * 1 \rightarrow x$
- $0 * x \rightarrow 0$
- $0 / x \rightarrow 0$
- $x^2 \rightarrow x * x$

Key Concept: Look for expressions that match standard algebraic rules and simplify them to make the code smaller and less complex.

2.4 Subexpression Elimination

Subexpression Elimination, or Common Subexpression Elimination, CSE, finds repeated expressions in the source code and replaces them with a single precomputed value stored in a temporary variable.

Example:

- $x = y + z;$
- $w = y + z;$
- The expression $y + z$ is repeated twice here. One can replace the second occurrence with x and eliminate the repetition of computation by using an intermediate variable:.

Key Concept: Keep track (e.g., via map or dictionary) of expressions that have been computed and reuse the result whenever possible to save redundant computations.

2.5 Dead Code Elimination

Dead Code Elimination eliminates the assignments to variables that are never used in the program.

Example:

- $a = b;$
- $x = 2 * b;$ // used later
- $p = z + y;$ // never used afterward
- In this example, the statement $p = z + y;$ does not affect the program's outcome because p is never used. It can be safely removed.

Key Concept: Analyze the program backward, starting from the final statements, to identify variables that are still "live" (used in subsequent computations). Discard assignments to variables that do not contribute to any live value.

Together, these optimizations improve code efficiency and redundancy and enhance the overall performance of compiled programs.

3. Design and Implementation

3.1 Overview of the Folder Structure

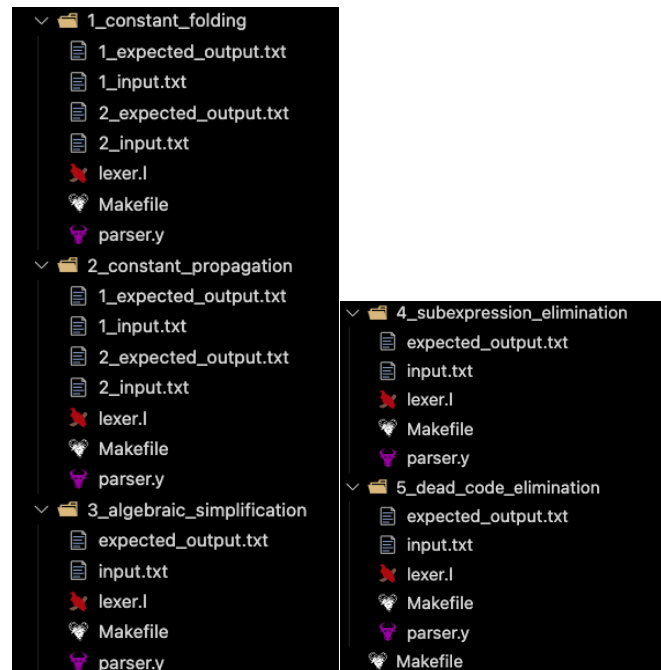


Figure 1 Folder Structure of Term Project

1. Lexical Analysis (Flex):

- A single `lexer.l` file is provided (or nearly identical across all optimizations).
- It recognizes tokens such as variables (`TOKEN_VAR`), numbers (`TOKEN_NUM`), and operators (`+`, `-`, `*`, `/`, `^`, `=`).

2. Parsing and Transformation (Yacc/Bison):

- Each optimization has its own `parser.y`, which contains the grammar rules.
- Within these grammar rules, we embed the transformation logic:
 - **Constant Folding:** Evaluate expressions if both operands are numeric.
 - **Copy or Constant Propagation:** Maintain a map of variable-to-constant and substitute when possible.
 - **Algebraic Simplification:** Match patterns (`x+0`, etc.) and output a simplified assignment.
 - **Common Subexpression Elimination:** Maintain a map from expression strings to a variable holding that expression. If it appears again, replace with the variable.
 - **Dead Code Elimination:** Store all assignments in a list, then perform a backward pass to check which assignments are used before printing them.

3. Makefiles for Each Optimization:

- Each directory (1 to 5) has a dedicated Makefile to build and run the corresponding optimization.
- A final “master” Makefile (makefile for all tests) runs each project’s tests and displays the result.

3.2 Implementation Details for Each Algorithm

3.2.1 Constant Folding

```
bool is_number(const std::string& s) {
    if (s.empty()) return false;
    for (size_t i = 0; i < s.size(); ++i) {
        if (!std::isdigit(s[i]) && s[i] != '.') return false;
    }
    return true;
}
```

Figure 2 Helper Function is_number for Constant Folding

```
prog:
    prog assgmt
    | assgmt
    ;

assgmt:
    TOKEN_VAR TOKEN_ASSIGN exps TOKEN_SEMICOLON {
        std::cout << $1 << "=" << $3 << ";" << std::endl;
    }
    ;
```

Figure 3 Prog and assgmt rules for Constant Folding

```
exps:
    exp TOKEN_PLUS exp {
        if (is_number($1) && is_number($3)) {
            $$ = strdup(std::to_string(std::stoi($1) + std::stoi($3)).c_str());
        } else {
            $$ = strdup((std::string($1) + "+" + std::string($3)).c_str());
        }
    }
}
```

Figure 4 General rule for operators in Constant Folding

```
exp TOKEN_POWER exp {
    if (is_number($1) && is_number($3)) {
        int temp = std::stoi($1);
        for (int i = 1; i < std::stoi($3); i++) {
            temp *= std::stoi($1);
        }
        $$ = strdup(std::to_string(temp).c_str());
    } else {
        $$ = strdup((std::string($1) + "^" + std::string($3)).c_str());
    }
}
```

Figure 5 Special rule for the power operator in Constant Folding

Constant folding has been integrated into the parser in order to optimize arithmetic expressions as they are parsed by evaluating constant sub-expressions at compile time. The non-terminal `exprs` manages operations including addition, subtraction, multiplication, division, and exponentiation. A helper function, `is_number`, is used to check if both operands are numeric. If they are, then the parser evaluates the result immediately and substitutes the expression with its constant value. Otherwise, it constructs a symbolic string representation, which allows the program to manipulate mixed expressions dynamically. This approach avoids most of the calculations at runtime, precomputing the constants, hence it is efficient in execution but still flexible enough for expressions involving variables or symbolic terms.

3.2.2 Copy or Constant Propagation

```
std::unordered_map<std::string, std::string> idents;
```

Figure 6 Implementation unordered_map

```
assgmt:
    TOKEN_VAR TOKEN_ASSIGN exps TOKEN_SEMICOLON {
        std::string input = std::string($3);
        std::string ident = std::string($1);
        idents[ident] = input;
        std::cout << ident << "=" << input << ";" << std::endl;
    }
;
```

Figure 7 assgmt rule for Copy or Constant Propagation

```
exp:
    TOKEN_VAR {
        std::string ident = std::string($1);
        if (idents.find(ident) != idents.end()) {
            ident = idents[ident];
        }
        $$ = strdup(ident.c_str());
    }
    | TOKEN_NUM {
        $$ = strdup(std::to_string($1).c_str());
    }
;
```

Figure 8 exp rule for Copy or Constant Propagation

Constant propagation is implemented as part of the parser to improve performance by substituting variable references with their constant values wherever possible. This optimization makes use of a global `unordered_map` called `idents` to keep track of mappings of variables to their values assigned. Similarly, when parsing an expression (`exp`), a variable (`TOKEN_VAR`) is looked up in the `idents` map. If it is found, its value is substituted before continuing to parse the rest of the expression; in this way, expressions with constants are simplified during parsing. This reduces lookup and calculation at runtime, but the design also supports variable reassignments by delivering the latest values to the optimization to capture any changes, and by making the expression flexible if it is not a constant.

3.2.3 Algebraic Simplification

```
assgmt:
TOKEN_VAR TOKEN_ASSIGN exps TOKEN_SEMICOLON {
    std::set<char> operators;
    operators.insert('+');
    operators.insert('-');
    operators.insert('*');
    operators.insert('/');
    operators.insert('^');
    std::string input = std::string($3);
    std::string ident = std::string($1);
    std::string beforeOperator;
    std::string afterOperator;
    char operatorFound = '\0';

    for (size_t i = 0; i < input.size(); ++i) {
        if (operators.count(input[i]) > 0) {
            operatorFound = input[i];
            beforeOperator = input.substr(0, i);
            afterOperator = input.substr(i + 1);
            break;
        }
    }

    if (operatorFound != '\0') {
        if (operatorFound == '+') {
            if (beforeOperator == "0") {
                if (ident != afterOperator) {
                    std::cout << ident << "=" << afterOperator << ";" << std::endl;
                }
            }
            if (afterOperator == "0") {
                if (ident != beforeOperator) {
                    std::cout << ident << "=" << beforeOperator << ";" << std::endl;
                }
            }
            if (beforeOperator != "0" && afterOperator != "0") {
                std::cout << ident << "=" << input << ";" << std::endl;
            }
        }
        if (operatorFound == '-') {
            if (beforeOperator == "0") {
                if (ident != afterOperator) {
                    std::cout << ident << "=" << afterOperator << ";" << std::endl;
                }
            }
        }
    }
}
```

Figure 9 assgmt rule for Algebraic Simplification Part 1

```

        if(afterOperator == "0"){
            if(ident != beforeOperator){
                std::cout << ident << "=" << beforeOperator <<";"<< std::endl;
            }
        }
        if(beforeOperator != "0" && afterOperator != "0"){
            std::cout << ident << "=" << input <<";"<< std::endl;
        }
    }
    if(operatorFound == '*'){
        if(beforeOperator == "1"){
            if(ident != afterOperator){
                std::cout << ident << "=" << afterOperator <<";"<< std::endl;
            }
        }
        if(afterOperator == "1"){
            if(ident != beforeOperator){
                std::cout << ident << "=" << beforeOperator <<";"<< std::endl;
            }
        }
        if(beforeOperator == "0" || afterOperator == "0"){
            std::cout << ident << "=" << 0 <<";"<< std::endl;
        }
        if(beforeOperator != "1" && afterOperator != "1" && beforeOperator != "0" && afterOperator != "0"){
            std::cout << ident << "=" << input <<";"<< std::endl;
        }
    }
    if(operatorFound == '/'){
        if(beforeOperator == "0"){
            std::cout << ident << "=" << 0 <<";"<< std::endl;
        }else{
            std::cout << ident << "=" << input <<";"<< std::endl;
        }
    }
    if(operatorFound == '^'){
        if(afterOperator == "2"){
            std::cout << ident << "=" << beforeOperator << "*" << beforeOperator << ";"<< std::endl;
        }else{
            std::cout << ident << "=" << input <<";"<< std::endl;
        }
    }
    } else{
        std::cout << ident << "=" << input <<";"<< std::endl;
    }
}
;

```

Figure 10 assgmt rule for Algebraic Simplification Part 2

There's also algebraic simplification being done in the parser; arithmetic expressions are simplified following common rules of mathematics. That is done as part of the assgmt (assignment) rule. The rule parses an expression containing operators such as +, -, *, / and ^ in such a way that it can recognize certain simplifications. Particular patterns, such as addition or subtraction of zero, multiplication by one, multiplication by zero, or squaring of a variable, are detected and simplified: for example, $x + 0$ simplifies to x , $x * 1$ remains x , and $x * 0$ becomes 0 . Likewise, an expression such as x^2 is transformed into $x * x$. That kind of simplifications decrease the complexity of generated expressions and improve their execution performance by preventing redundant calculations. At the same time, non-redundant expressions are not changed, so the output of the program is preserved correct and only meaningful optimizations are applied.

3.2.4 Common Subexpression Elimination

```
std::unordered_map<std::string, std::string> exps_map;
```

Figure 11 Implementation of exps_map

```
assgmt:
    TOKEN_VAR TOKEN_ASSIGN exps TOKEN_SEMICOLON {
        std::string expression_return($3);
        if (exps_map.find(expression_return) != exps_map.end()) {
            std::cout << $1 << "=" << exps_map[expression_return].c_str() << ";" << std::endl;
        } else {
            std::cout << $1 << "=" << $3 << ";" << std::endl;
            exps_map[expression_return] = $1;
        }
    }
;
```

Figure 12 assgmt rule of Common Subexpression Elimination

The CSE is realized by means of a built-in parser in order to evaluate the same expression only once. It relies, therefore, on the following global `unordered_map<exp/var_set> exps_map` to keep track of expressions along with their variables; then, when parsing any kind of assignment `assgmt`, its right-hand-side expression `exps` is run against this map. If the expression already exists, instead of computing/assigning the same expression, the parser reuses the associated variable name. The parser stores new expressions along with their associated variables into the map. This makes the approach efficient and reduces the consumption of memory if there are complex or repetitive computations involved. The implementation keeps computational overhead minimal while obtaining precise results.

3.2.5 Dead Code Elimination

```
std::vector<std::pair<std::string, std::string> > all_assgmt;
static std::unordered_set<std::string> extract_ids(const std::string &expr);

void eliminate_dead_code() {
    std::vector<bool> keep(all_assgmt.size(), false);
    std::unordered_set<std::string> used;
    if (!all_assgmt.empty()) {
        keep[all_assgmt.size() - 1] = true;
        used.insert(all_assgmt.back().first);
        std::unordered_set<std::string> lastExprVars = extract_ids(all_assgmt.back().second);
        for (std::unordered_set<std::string>::iterator it = lastExprVars.begin(); it != lastExprVars.end(); ++it) {
            used.insert(*it);
        }
    }

    for (int i = (int)all_assgmt.size() - 2; i >= 0; i--) {
        const std::string &var = all_assgmt[i].first;
        const std::string &expr = all_assgmt[i].second;
        if (used.find(var) != used.end()) {
            keep[i] = true;
            std::unordered_set<std::string> rhsIds = extract_ids(expr);
            for (std::unordered_set<std::string>::iterator it = rhsIds.begin(); it != rhsIds.end(); ++it) {
                used.insert(*it);
            }
        }
    }

    for (size_t i = 0; i < all_assgmt.size(); i++) {
        if (keep[i]) {
            std::cout << all_assgmt[i].first << "=" << all_assgmt[i].second << ";" << std::endl;
        }
    }
}
```

Figure 13 Function to eliminate dead code.

```
static std::unordered_set<std::string> extract_ids(const std::string &expr) {
    std::unordered_set<std::string> ids;
    std::string token;
    for (size_t i = 0; i < expr.size(); i++) {
        char c = expr[i];
        if (std::isalnum(static_cast<unsigned char>(c)) || c == '_') {
            token.push_back(c);
        } else {
            if (!token.empty()) {
                if (std::isalpha(static_cast<unsigned char>(token[0])) || token[0] == '_') {
                    ids.insert(token);
                }
                token.clear();
            }
        }
    }

    if (!token.empty()) {
        if (std::isalpha(static_cast<unsigned char>(token[0])) || token[0] == '_') {
            ids.insert(token);
        }
        token.clear();
    }

    return ids;
}
```

Figure 14 Helper function to extract identifiers.

```

assgmt:
    TOKEN_VAR TOKEN_ASSIGN exprs TOKEN_SEMICOLON
    {
        std::string var($1);
        std::string expr($3);
        all_assgmt.push_back(std::make_pair(var, expr));
    }
;

```

Figure 15 Rule for parsing assignments.

```

int main() {
    if (yyparse() == 0) {
        eliminate_dead_code();
    }
    return 0;
}

```

Figure 16 Main function calling dead code elimination.

Dead code elimination implemented in the parser is the elimination of unused assignments, ensuring that the generated code only includes computations that are effectively needed. This optimization uses a global vector, `all_assgmt`, to keep track of variable assignments in the order they are parsed. After parsing, the `eliminate_dead_code` function processes these assignments in reverse, determining which variables are actually used in later computations. An auxiliary function, `extract_idents`, discovers variable references within expressions and is complemented by using a `std::unordered_set` to keep track of used variables. The assignment is erased if its value is no longer needed. This ensures that only the assignments contributing to either the outputs or to further computations are retained, which in turn reduces memory usage, making execution more efficient. Correctness is ensured through careful consideration of all the dependencies, making this implementation reliable for large and complex computations.

4. Testing and Results

4.1 Test Inputs and Expected Outputs

Each optimization includes one or more sample input files and corresponding expected outputs. Below is an example for **Constant Folding**:

1_input.txt:	1_expected_output.txt:	2_input.txt:	2_expected_output.txt:
d=4; a=2+2; b=2^9; c=d^3; e=5; f=3*4; g=6/2; h=m; p=0; j=j+p; r=e*p; s=a;	d=4; a=4; b=512; c=d^3; e=5; f=12; g=3; h=m; p=0; j=j+p; r=e*p; s=a;	d=4; a=2+2; b=2^9; c=4^3; e=5; f=3*4; g=6/2; h=m; p=0; j=j+0; r=5*0; s=a;	d=4; a=4; b=512; c=64; e=5; f=12; g=3; h=m; p=0; j=j+0; r=0; s=a;

Table 1 Inputs and Expected Outputs of Constant Folding

```

Red    = \033[0;31m
Green  = \033[0;32m
NoColor= \033[0m

all: lex yacc
    g++ lex.yy.c y.tab.c -ll -o constant_folding -Wno-deprecated

yacc: parser.y
    yacc -d -v parser.y

lex: lexer.l
    lex lexer.l

run: all
    ./constant_folding < 1_input.txt > 1_output.txt
    ./constant_folding < 2_input.txt > 2_output.txt
    if diff -q 1_output.txt 1_expected_output.txt && diff -q 2_output.txt 2_expected_output.txt; then \
        echo "$(Green)🎉🎉 Constant Folding Algorithm is working correctly 🎉🎉$(NoColor)"; \
    else \
        echo "$(Red)💣💣 Constant Folding Algorithm is not working correctly 💣💣$(NoColor)"; \
    fi

clean:
    rm -f lex.yy.c y.tab.c y.tab.h constant_folding y.output 1_output.txt 2_output.txt

```

Figure 17 Makefile of Constant Folding

After compiling (using `make run`) and running the optimizer (`./constant_folding < 1_input.txt > 1_output.txt`), the script compares `1_output.txt` with the expected output to confirm correctness.

Similar test input-output pairs are provided for Constant Propagation, Algebraic Simplification, Subexpression Elimination, and Dead Code Elimination. Each directory has a dedicated `make run` command that performs the transformation, compares the result with the expected output, and prints success/failure messages.

Copy or Constant Propagation			
1_input.txt: d=4; a=2+2; b=2^9; c=d^3; e=5; f=3*4; g=6/2; h=m; p=0; j=j+p; r=e*p; s=a;	1_expected_output.txt: d=4; a=2+2; b=2^9; c=d^3; e=5; f=3*4; g=6/2; h=m; p=0; j=j+0; r=5*0; s=2+2;	2_input.txt: d=4; a=4; b=512; c=d^3; e=5; f=12; g=3; h=m; p=0; j=j+p; r=e*p; s=a;	2_expected_output.txt: d=4; a=4; b=512; c=d^3; e=5; f=12; g=3; h=m; p=0; j=j+0; r=5*0; s=4;
Algebraic Simplification		Subexpression Elimination	
input.txt: d=4; a=4; b=512; c=64; e=5; f=12; g=3; h=m; p=0; j=j+0; r=0; s=4; x=x*1; x=1*x; z=x*1; y=y+0; y=0+y; z=y+0; o=20*0; p=0/20; y=y-0; y=0-y; z=y-0;	expected_output.txt: d=4; a=4; b=512; c=64; e=5; f=12; g=3; h=m; p=0; r=0; s=4; z=x; z=y; o=0; p=0; z=y;	input.txt: d=4; a=4; b=512; c=d^3; e=5; f=12; g=3; h=m; p=0; j=j+p; r=e*p; s=a; x=y+z; w=y+z; q=d^3; u=512; q2=5; q3=m; q4=a;	expected_output.txt: d=4; a=d; b=512; c=d^3; e=5; f=12; g=3; h=m; p=0; j=j+p; r=e*p; s=a; x=y+z; w=x; q=c; u=b; q2=e; q3=h; q4=s;
Dead Code Elimination		Output of Test Results	
input.txt: b=z+y; a=b; x=2*b; p=z+y; x=3*x; p1=x*4; p2=b; p3=4+p1; p4=20-x; p5=20-x; p6=p5*5; p7=p3*p6;	expected_output.txt: b=z+y; x=2*b; x=3*x; p1=x*4; p3=4+p1; p5=20-x; p6=p5*5; p7=p3*p6;	<pre> >>> Checking < 1_constant_folding > 🚀 Constant Folding Algorithm is working correctly 🚀 >>> Checking < 2_constant_propagation > 🚀 Constant Propagation Algorithm is working correctly 🚀 >>> Checking < 3_algebraic_simplification > 🚀 Algebraic Simplification Algorithm is working correctly 🚀 >>> Checking < 4_subexpression_elimination > 🚀 Subexpression Elimination Algorithm is working correctly 🚀 >>> Checking < 5_dead_code_elimination > 🚀 Dead Code Elimination Algorithm is working correctly 🚀 </pre>	

Table 2 All tests and result

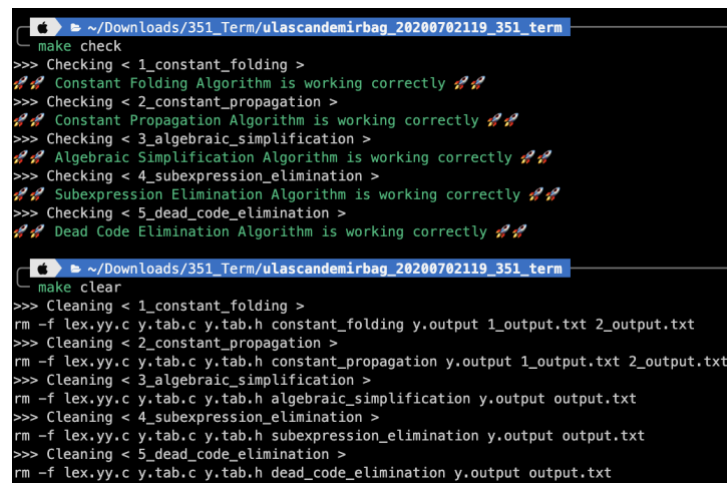
4.2 Makefile to Run All Tests

```
PROJECTS := 1_constant_folding \
            2_constant_propagation \
            3_algebraic_simplification \
            4_subexpression_elimination \
            5_dead_code_elimination

check:
    @for dir in $(PROJECTS); do \
        echo ">>> Checking < $$dir >"; \
        (cd $$dir && make run) | tail -n 1; \
    done

clear:
    @for dir in $(PROJECTS); do \
        echo ">>> Cleaning < $$dir >"; \
        (cd $$dir && make clean) \
    done
```

Figure 18 Makefile to run all tests



```
~/Downloads/351_Term/ulascandemirbag_20200702119_351_term
make check
>>> Checking < 1_constant_folding >
Constant Folding Algorithm is working correctly
>>> Checking < 2_constant_propagation >
Constant Propagation Algorithm is working correctly
>>> Checking < 3_algebraic_simplification >
Algebraic Simplification Algorithm is working correctly
>>> Checking < 4_subexpression_elimination >
Subexpression Elimination Algorithm is working correctly
>>> Checking < 5_dead_code_elimination >
Dead Code Elimination Algorithm is working correctly

~/Downloads/351_Term/ulascandemirbag_20200702119_351_term
make clear
>>> Cleaning < 1_constant_folding >
rm -f lex.yy.c y.tab.c y.tab.h constant_folding y.output 1_output.txt 2_output.txt
>>> Cleaning < 2_constant_propagation >
rm -f lex.yy.c y.tab.c y.tab.h constant_propagation y.output 1_output.txt 2_output.txt
>>> Cleaning < 3_algebraic_simplification >
rm -f lex.yy.c y.tab.c y.tab.h algebraic_simplification y.output output.txt
>>> Cleaning < 4_subexpression_elimination >
rm -f lex.yy.c y.tab.c y.tab.h subexpression_elimination y.output output.txt
>>> Cleaning < 5_dead_code_elimination >
rm -f lex.yy.c y.tab.c y.tab.h dead_code_elimination y.output output.txt
```

Figure 19 Execution of Make Check and Make Clear Rules

Make check: Loops through projects, runs make run in each directory, and shows the last line of output.

Make clear: Iterates over projects, runs make clean in each directory to remove any build artifacts.

4.3 Observations

All the transformations behaved as expected when tested. The constant expressions got simplified, known constants got replaced, simple algebraic rules were applied, redundant expressions were combined and unused assignments got eliminated. In order to keep the code manageable and well-organized, every transformation was placed in a separate directory and used the same lexical specification `lexer.l` thus being modular and easy to extend or debug.

5. Conclusion

The present project has achieved and evaluated the efficiency of five crucial compiler optimization techniques namely: Constant Folding, Constant Propagation, Algebraic Simplification, Common Subexpression Elimination and Dead Code Elimination. All the optimization produced significant enhancement in the code efficiency where redundancy has been cut down and runtime improved.

This is due to the fact that the design of the system was modular where Flex was used for lexical analysis and Yacc/Bison for parsing which made the implementations clear and organized, and also helped with testing and debugging. The output of the test cases were in line with what was expected indicating that the optimizations that were used were correct and efficient.

This work thus demonstrates the relevance of compiler optimizations in the current practice of programming, thus enhancing the appreciation of the importance of optimizations done during the compilation stage on the performance and resource consumption of software. Future work can also involve the addition of new optimization techniques and the assessment of their performance on bigger programs.

6. References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, & Tools*. Pearson.

Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann.