# CSE351 TERM PROJECT REPORT
# Dead Code Elimination Using Lex and Yacc

## Student:Gökay Ertuğrul
## Student ID: 20220702010

## 1. INTRODUCTION

In this project, I implemented a Dead Code Elimination  algorithm for an intermediate language using Lex and Yacc tools. Dead code elimination is one of the most important optimization techniques used in compilers.

The main goal of this project was to analyze a given intermediate language code and remove the assignment statements that do not contribute to the final output. These unnecessary statements are called **dead code** because their results are never used by any other part of the program.

For example, if we have a variable that is assigned a value but never used afterwards, that assignment is considered dead code and can be safely removed. This optimization makes the code shorter and potentially faster to execute.

I chose to implement this project using Lex for lexical analysis (tokenization) and Yacc for parsing and semantic analysis. This approach allowed me to focus on the algorithm itself rather than dealing with low-level string processing.

## 2. DESIGN OVERVIEW

### 2.1 Intermediate Language Syntax

The intermediate language  that my program accepts has the following characteristics:

- Assignment statements in the form: variable = expression;
- Expressions can be simple (single operand) or binary (two operands with an operator)
- Supported operators: + (addition), - (subtraction), * (multiplication), / (division), ^ (power)
- Operands can be either variable names or integer constants (including negative numbers)
- The last line contains the live variables list in curly braces: { var1, var2, ... }

Here is an example of valid IL code:

```
a = b + c;
x = 5;
y = x * 2;
{ y }
```

In this example, only the variable 'y' is live at the end. This means 'a = b + c' is dead code because 'a' is never used.

## 2.2 Algorithm Explanation

The DCE algorithm I implemented follows three main steps as described in the project requirements:

**STEP 1: Parse and Store**
First, I parse all the assignment statements and store them in memory. I also read the live variables list from the last line. This gives me the initial set of variables that are "live" (needed) at the end of the program.

**STEP 2: Backward Analysis**
This is the core of the algorithm. I process the statements in REVERSE order (from last to first). For each statement, I check:
- Is the destination variable in the live set?
- If yes: This statement is needed. I keep it and add its source operands to the live set.
- If no: This statement is dead code. I skip it.

The key insight is that when we find a live statement, its source variables must also become live (because we need their values) but the destination variable becomes dead before this point (because this is where it gets its value).

**STEP 3: Output Generation**
After processing all statements backwards, I have a list of needed statements (in reverse order). I simply reverse this list and print it to get the correct output order.

# 3. IMPLEMENTATION

## 3.1 Lexer (lexer.l)

My lexer recognizes the following tokens:
- **ID**: Variable names starting with a letter, followed by letters or digits
- **NUMBER**: Integer constants (positive or negative)
- **Operators**: PLUS (+), MINUS (-), MULT (*), DIV (/), POWER (^)
- **Special characters**: ASSIGN (=), SEMICOLON (;), LBRACE ({), RBRACE (}), COMMA (,)

The lexer ignores whitespace and newlines, which makes the input format flexible.

## 3.2 Parser (parser.y)

For the parser, I defined the following grammar:

```
program    -> statements liveset
statements -> statements statement | (empty)
statement  -> ID = expression ;
```

```
expression -> operand | operand OP operand
operand    -> ID | NUMBER
liveset    -> { varlist }
varlist    -> ID | varlist , ID
```

I used a Statement structure to store each parsed statement:

- **dest**: the destination variable name
- **src1**, **src2**: the source operands
- **op**: the operator (or 0 for simple assignments)
- **is_src1_num, is_src2_num:** flags to track if operands are numbers
- **original: s**tores the original statement string to preserve formatting for output

For the live variable set, I used a simple array-based approach. The main functions are:

- **is_live()**: checks if a variable is in the live set
- **add_to_live()**: adds a variable to the live set
- **remove_from_live()**: removes a variable from the live set
- **perform_dce()**: the main function that implements the DCE algorithm

## 4. COMPILATION & USAGE

Build and Run (using shell script):

```
./run.sh
```

This will clean previous build files, compile the project using Lex and Yacc and run both test cases automatically .

Manual Build:

```
make
```

Manual Run:

```
./dce < input.il
```

Clean:

```
make clean
```

# 5. TEST RESULTS

I tested my implementation with two test cases from the project description.

## Test 1:

Input:

```
  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;
  { r, s }
```

Output:

```
  a=2+2;
  e=5;
  p=0;
  r=e*p;
  s=a;
```

Analysis: Out of 11 statements, only 5 are needed. The variables b, c, f, g, h, and j are never used to compute r or s, so they are eliminated.

**Result: PASS**

## Test 2:

Input:

```
  b=z+y;
  a=b;
  x=2*b;
  { x }
```

Output:

```
  b=z+y;
  x=2*b;
```

Analysis: Variable 'a' is assigned the value of 'b', but 'a' is never used. Only 'x' is live, and it depends on 'b'. So 'a=b' is dead code.

**Result: PASS**

# 6. CHALLENGES AND SOLUTIONS

During this project, I faced a few challenges:

**1. Handling the reverse order processing:** First, I tried to process statements forward, but this made tracking live variables difficult. After understanding the algorithm better, I realized that backward processing is much more natural for this problem.

**2. Distinguishing variables from numbers:** I needed to make sure that numeric constants don't get added to the live set. I solved this by keeping flags (is_src1_num, is_src2_num) for each statement.

**3. Memory management:** I had to be careful with strdup() and free() calls to avoid memory leaks, especially in the Yacc semantic actions.

# 7. CONCLUSION

In this project, I developed a dead code elimination tool for an intermediate language using Lex and Yacc. The main components I implemented are:

- A lexer (lexer.l) that tokenizes the input IL code, recognizing variables, numbers, operators, and special characters like assignment, semicolon, and braces
- A parser (parser.y) that processes the grammar, stores each statement with its destination, source operands, operator, and original text format
- A DCE algorithm that analyzes statements in reverse order using a live variable set to determine which statements are necessary
- Helper functions (is_live, add_to_live, remove_from_live, perform_dce) for managing the live variable set and executing the elimination process

The program takes IL code as input, reads the live variables from the last line, and processes statements backwards. For each statement, it checks if the destination variable is in the live set. If yes, the statement is kept and its source operands are added to the live set. If no, the statement is dead code and is eliminated. Finally, the remaining statements are printed in the correct order.

Both test cases from the project description pass successfully. Test 1 reduces 11 statements to 5, and Test 2 reduces 3 statements to 2, producing the expected optimized output in both cases.

Through this project, I gained practical experience with Lex and Yacc tools and developed a better understanding of how backward dataflow analysis works in compiler optimization.