YEDİTEPE ÜNİVERSİTESİ

CSE 351 Programming Languages Term Project Report

Dead Code Elimination Algorithm

Ege Huriel

Assigned by Prof. Dr. Gürhan Küçük

January 12, 2026

Contents

PROJECT OVERVIEW

Introduction

Dead Code Elimination is a fundamental optimization technique used in compilers to remove program. Such statements increase code size and reduce efficiency without affecting program semantics.

In this project, a Dead Code Elimination algorithm is implemented for a simplified Intermediate Language (IL). The implementation uses Lex for lexical analysis and Yacc fir syntax analysis. After parsing the IL program, the DCE algorithm is applied to eliminate unnecessary assignment statements based on variable liveness analysis.

Objectives

The main objective of this project is to design and implement a Dead Code Elimination algorithm for a simplified Intermediate Language (IL) using Lex and Yacc. The project aims to demonstrate how classical compiler optimization techniques can be applied after lexical and syntactic analysis.

More specifically, the objectives of this project are:

1. To perform **lexical analysis** of the given IL program using Flex.
2. To perform **syntax analysis** and construct an internal representation of the program using Bison.
3. To implement a **liveness-based dead code elimination algorithm** that removes assignment statements whose result are never used.
4. To ensure that the optimized program preserves the original program semantics while eliminating unnecessary computations.
5. To provide a clean and modular compiler-like pipeline separating parsing and optimization logic.

By completing this project, the goal is practical experience with compiler construction tools and understand how optimization passes operate on an intermediate representation rather than raw source code.

Notes

The implementation in this project follows several important design considerations and assumptions:

1. The supported Intermediate Language (IL) allows only **simple assignment statements** with at the most one binary operator.
2. Expressions containing more than two operands are considered invalid and are not supported.
3. Each assignment statement has exactly **one destination variable**.
4. Source operands may be either **variable or signed integer constants.**
5. The final line of the program explicitly specifies the set of live variables at program termination, which is used as the starting point for liveness analysis.

Lex and Yacc are used strictly for **lexical and syntactic processing**, while the dead code elimination logic itself is implemented in C++. This separation ensures better readability, maintainability and adherence to standart compiler design principles.

All generated and optimized outputs strictly follow the syntax of the defined IL, ensuring consistency between input and output formats.

Technical Requirements

This project is implemented in accordance with the technical requirements specified in the course term project instructions. The system is designed as a small compiler-like pipeline consisting of lexical analysis, syntax analysis, intermediate representation construction and optimization.

The main technical requirements of the project are as follows:

1. The project must utilize Lex to perform lexical analysis of the input IL program.
2. The project must utilize Yacc to perform syntax analysis and validate the grammatical structure of the program.

3. The implementation must correctly parse assignment statements, binary expressions and the final live variable block.
4. A Dead Code Elimination algorithm must be implemented to remove assignment statements whose result are not used int the computation of live variables.
5. The project must support signed integer constants and variable operands.
6. The algorithm must process the program by traversing the statement list in reverse order, as required by standart liveness analysis techniques.
7. The final optimized output must preserve the original program semantics while eliminating dead code.
8. The submission must include all source files (.l, .y and .cpp) sample input files and clear instructions or commands to compile and execute the program.

In addition to these requirements, the implementation follows modular design principles. Lex and Yacc are responsible only for parsing-related tasks, while the optimization logic implemented separately in C++. This design choice improves clarity, maintainability and aligns with real-world compiler architectures.

Intermediate Language Specification

The Dead Code Elimination algorithm implemented in this project operates on a simplified **Intermediate Language (IL)** specifically designed to facilitate static and analysis and optimization. This language abstracts away high-level programming constructs and focusses on assignment-based computations.

The IL is intentionally minimal, allowing the optimization logic to be clearly demonstrated without additional semantic complexity.

Language Syntax

An IL program consist of two main parts:

1. A sequence of assignment statements.
2. A final live variable block specifying variables that are live at program termination.

Each assignment statement follows one of the forms below:

```
variable = operand;
variable = operand operator operand;
```

Where:

variable is a valid identifier.

operand is either a variable or a signed integer constant.

operator is one of the following binary operators: + - * / ^

The program must end with a live variable block enclosed in braces:

{v1, v2, ... , vn}

This block explicitly defined the set of variables considered live at the end of program execution and serves as the starting point for liveness analysis.

Valid and Invalid Statements

The intermediate language enforces several constraints to ensure simplicity and determinism:

Valid Statements Include:

```
a = 5;
b = a;
c = b + -3;
x = y * z;
```

Invalid Statements Include:

```
a = b + c * d;
x = (a + b);
y = a + b + c;
```

Only one operator per expression is allowed and expressions with nested or chained operations are explicitly disallowed.

These restrictions ensure that each assignment statement can be analyzed independently during reverse traversal and liveness analysis, simplifying the implementation of the Dead Code Elimination algorithm.

Systems Design and Architecture

The system is designed following a **compiler-like module architecture**, where each phase of processing is clearly separated. This design choice improves readability, maintainability and aligns with standart compiler construction principles.

The overall workflow consist of lexical analysis, syntax analysis, intermediate representation construction and an optimization pass implementing dead code elimination.

Overall System Flow

The system processes an IL program through the following stages:

1. Lexical Analysis

The input IL file is first processed by the lexer implemented using Flex. The lexer converts the raw character stream into a sequence into a sequence of tokens such as identifiers, integer constants, operators and punctuation symbols.

2. Syntax Analysis

The token stream generated by the lexer is passed to the parser implemented using Bison. The parser validates the program structure according to the defined grammar rules and identifies assignment statements and the final live variable block.

3. Intermediate Representation Construction

During parsing, each valid assignment statement is converted into an internal data structure representing the programs intermediate form. This representation captures destination variables, operands and operators in a structured manner suitable for analysis.

4. Dead Code Elimination Pass

After parsing is completed, the dead code elimination algorithm is applied. The algorithm performs reverse traversal of the statement list and uses liveness analysis to determine which assignments contribute to the final live variables.

5. Output Generation

The optimized set of statements is printed in the original order, forming the final IL program after dead code elimination.

This sequential pipeline ensures that each stage has a single responsibility and interacts with other stages through well-defined interfaces.

Intermediate Representation Design

The intermediate representation is designed to be simple yet expressive enough to support dead code elimination. It consist of two main components:

Operand:

Represents either a variable or a signed integer constant. This abstraction allows the algorithm to uniformly handle variable references and constant values.

Statement:

Represents a single assignment instruction including:

- Destination variable
- Source operands
- Binary operator

The IR is shared between the parser and the optimization logic through a common header file. This ensures type consistency across different compilation units and avoids duplication of data structures.

By operating on this intermediate representation renter than raw source text, the dead code elimination algorithm remains independent of parsing details and can be extended or modified more easily.

Lexical Analysis

Lexical analysis is the first stage of the system pipeline and is responsible for converting the raw input character stream into a sequence of meaningful token. In this project, lexical analysis is implemented using Flex.

The lever reads the input IL program and identifies language elements such as identifiers, integer constants, operators and punctuation symbols while ignoring whitespace and comments.

Token Definitions

The lever recognizes the following categories of tokens:

- Identifiers **(ID):**

    Variable names consisting of alphabetic characters and underscores, optionally followed by alphabetical characters.

Integer Constants **(INT):**

Integer represented as sequences of digits. Negative integers are handled at parser level using the minus operator

Operators:

Binary arithmetic operators supported by the IL: + - * / ^

Lexer Implementation

The lexer is defined int the ***dce.l*** file using Flex syntax. Each token pattern is associated with an action that returns the corresponding token to the parser.

When an identifier token is recognized, its lexeme is duplicated and passed to the parser to ensure safe memory handling. Integer constants are converted from their textual representation into integer values before being passed to the parser.

The lexer is configured with the following options to simplify integration.

noyywarp to avoid requiring a wrap up function.

noinput and ***nounput*** to disable unused input/output functions.

yylineno to enable line number tracking for error reporting.

By isolating all lexical concerns within the lexer, the system ensures that the parser and optimization stages operate on a clean and well defined token stream.

Syntax Analysis

Syntax analysis is responsible for validating the structural correctness of the input IL program according to formally defined grammar. In this project, syntax analysis is implemented using Yacc.

The parser processes the token stream generated by the lexer and ensures that the program follows the rules of the defined Intermediate Language. Upon successful parsing, the program is transformed into an internal representation suitable for further analysis and optimization.

Grammar Rules

The grammar defines the structure of a valid IL program as sequence of assignment statements followed by a final live variable block.

At a high level, the grammar enforces the following rules:

A program consist of zero or more assignment statements.

Each assignment statement assigns a value to exactly one destination variable.

Expression may consist of either:

A single operand

or Two operands connected by a single binary operator.

The program must terminate with a live variable block enclosed in braces.

This grammar ensures that complex expressions and ambiguous constructs are disallowed, simplifying both parsing and subsequent optimization.

Parser Actions

During parsing, semantic actions associated with grammar rules are used to construct the intermediate representation **(IR)** of the program.

For each assignment statement:

A Statement object is created.

The destination variable name is stored.

Source operands and operators are recorded.

The constructed statement is appended to a global list representing the program order.

For the live variable block:

Each listed variable is added to a separate list representing variables that are live at program termination.

Memory management for dynamically allocated tokens is handled carefully to prevent leaks, ensuring that all identifier strings are freed once they are stored in the IR.

By combining strict grammar validation with immediate IR construction, the parser provides a clean and reliable bridge between lexical analysis and the dead code elimination phase.

Dead Code Elimination Algorithm

Dead Code Elimination is an optimization technique that removes program stamens whose computed values are never used. In this project, DCE is implemented using a liveness based analysis applied to the intermediate representation of the IL program.

The algorithm determines whether an assignment contributes to the computation of variables that are live at the end of program execution.

Liveness Analysis

Liveness analysis identifies variables whose values may be used in the future. A variable is considered live at a given point if its current value can be used later in the program.

In this project liveness analysis is simplified by explicitly providing the set of variables t-hat are live at program termination through the final live variable block. This set serves as the initial live set for the analysis.

During the analysis:

A variable becomes live if it appears as a source operand in a statement whose result is required.

A variable ceases to be live once its required value has been defined.

This information is used to decide whether each assignment statement is necessary or can be safely eliminated.

Reverse Traversal Strategy

The DCE algorithm processes the program statements in reverse order. Reverse traversal is essential because the liveness of a variable depends on its future use rather than its past definitions.

The algorithm proceeds as follows:

1. Initialize the live variable set using the variables list in the final live block.
2. Traverse the list of statements from the last statement to the first.
3. For each statement, check whether its destination variable is present in the live set

This reverse processing ensures that dependencies between statements are correctly captured.

Statement Elimination Criteria

For each assignment statement encountered during reverse traversal, the following rules are applied:

- If the destination variable of the statement is **not** in the live set, the statement is considered dead and is removed.
- If the destination variable is in the live set:
    - The statement is retained in the optimized program.
    - The destination variable is removed from the live set, as its required value has now been defined.
    - Any source variables used in the statement are added to the live set.

After all statements have been processed, the retained statements are reversed again to restore the original execution order.

This method guarantees that all retained statements contribute directly or indirectly to the computation of the live variables while safely eliminating unnecessary assignments.

Implementation Details

This section describes the concrete implementation choices made during the development of the project, focusing on the integration of Lex and Yacc, the intermediate representation and the dead code elimination logic.

Lex and Yacc Integration

Lex and Yacc are integrated in a standart compiler pipeline configuration. The lexer is responsible for tokenizing the input IL program, while the parser validates the program structure and constructs the intermediate representation.

The parser actions directly populate shared data structures representing the program:

- A vector of Statement objects storing assignment instructions in program order.
- A vector of strings storing the final live variable list.

These data structures are declared as global and shared between the parser and the main driver program via common header file.

This design avoids duplication and ensures consistency across different compilation units.

Memory management is carefully handled during parsing. Identifier tokens are dynamically allocated by the lexer and freed once the their values are safely stored in the intermediate representation.

Dead Code Elimination Implementation

The dead code elimination logic is implemented is the main drive program (main.cpp). After parsing is completed, the algorithm performs the following steps:

1. The final live variable list is converted into a set to allow efficient membership.
2. The list of statement is traversed in reverse order.
3. Statements whose destination variables are not present in the live set are discard.
4. Statements whose destination variables are live are retained and is updated.
5. The retained statement are reversed to restore the original execution order.

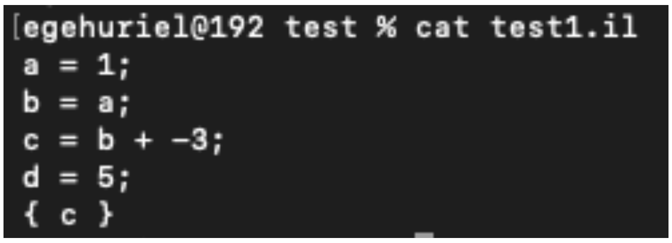
The algorithm operates exclusively on the intermediate representation rather than raw text, which simplifies the implementation and improves robustness. The final optimized program is printed in valid IL syntax, preserving the structure of the original input.

Examples and Test Cases

Several test cases were prepared to verify the correctness of the dead code elimination algorithm under different scenarios. These test cases demonstrate how the algorithm handles variable dependencies, unused assignements and live variable propagation.

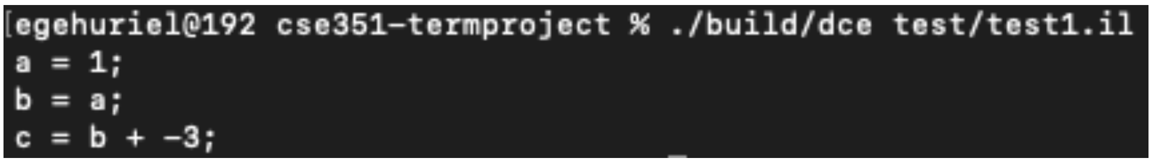
Example 1: Basic Dead Code Elimination

Input:



```
[egehuriel@192 test % cat test1.il
 a = 1;
 b = a;
 c = b + -3;
 d = 5;
 { c }
```

Output:

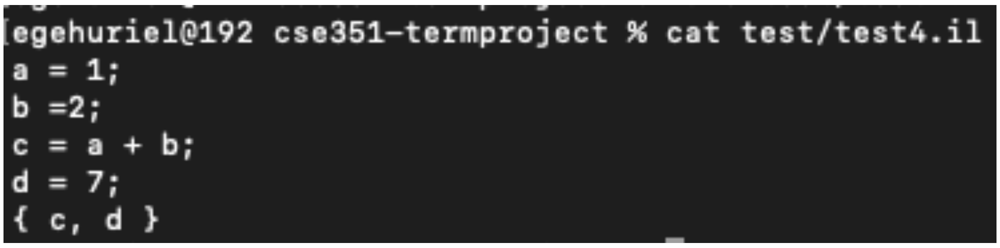


```
[egehuriel@192 cse351-termproject % ./build/dce test/test1.il
 a = 1;
 b = a;
 c = b + -3;
```

The assignment d = 5; eliminated because the variable d is not live at the end of the program and is not used by any other statement.

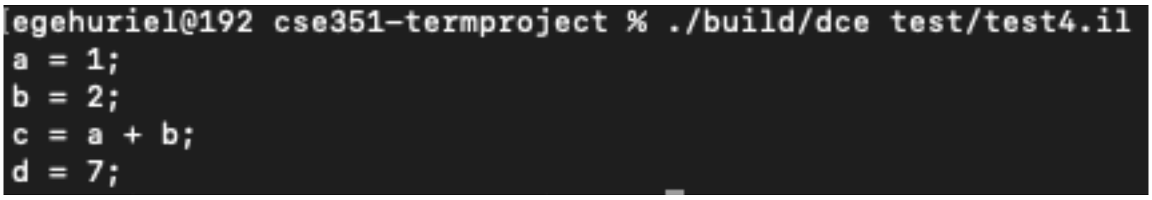
Example 2: Multiple Live Variables

Input:



```
[egehuriel@192 cse351-termproject % cat test/test4.il
a = 1;
b =2;
c = a + b;
d = 7;
{ c, d }
```

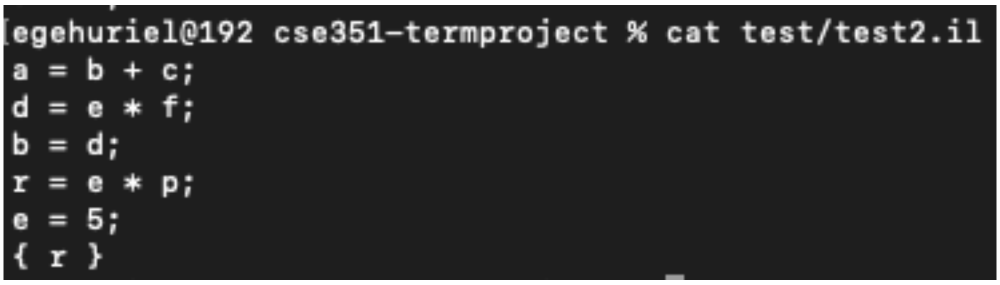
Output:



```
[egehuriel@192 cse351-termproject % ./build/dce test/test4.il
a = 1;
b = 2;
c = a + b;
d = 7;
```

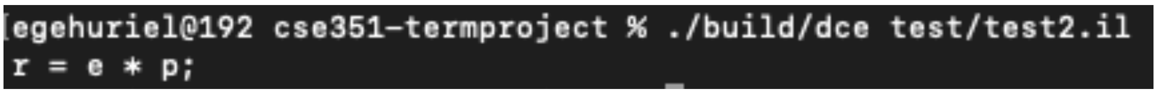
Example 3:

Input:



```
egehuriel@192 cse351-termproject % cat test/test2.il
a = b + c;
d = e * f;
b = d;
r = e * p;
e = 5;
{ r }
```

Output:



```
egehuriel@192 cse351-termproject % ./build/dce test/test2.il
r = e * p;
```

Compilation and Execution

The project can be compiled and executed using standart Flex, Bison and C++ compilat-ions tools. The following commands demonstrate the complete build and execution process.

bison -d src/dce.y -o build/dce.tab.c

flex -o build/lex.yy.c src/dce.l

g++ src/main.cpp build/dce.tab.c build/lex.yy.c -I build -I src -o build/dce

To run the program on a sample input file:

./build/dce test/test1.il

The program reads the IL file, performs dead code elimination and prints the optimized IL code to standart output.

Conclusion

In this project, a complete Dead Code Elimination algorithm was successfully impleme-nted using Lex and Yacc. The system parses a simplified intermediate language, constructs an internal representation and applies a liveness based optimization to remove unnecessary assignment statements.

The modular architecture of the project clearly separates lexical analysis, intermediate representation construction and optimization logic. The design not only improves code readability and maintainability but also reflects real world compiler construction practices.

The result demonstrate that dead code can be safely eliminated without altering program semantics, leading to more efficient and cleaner code.

References

1. CSE 351 Programming Languages Term Project Instruction PDF File
2. CSE 351 Programming Languages Lecture Slides / Notes
3. Stack Overflow. "Stack Overflow - Where Developers Learn, Share, & Build Careers." Stack Overflow, 2019, stackoverflow.com.
4. "How to Put Header File to .Tab.h in Bison?" Stack Overflow, stackoverflow.com/questions/52905235/how-to-compile-a-yacc-file-that-contains-a-header-file.
5. "In Lex and Yacc Code Printf Not Working in Yacc File." Stack Overflow, stackoverflow.com/questions/43900607/in-lex-and-yacc-code-printf-not-working-in-yacc-file.
6. GeeksforGeeks. "Dead Code Elimination." GeeksforGeeks, 21 Sept. 2023, www.geeksforgeeks.org/compiler-design/dead-code-elimination/.
7. arnav58. "GitHub - Arnav58/Deadcode-Removal-Complier: System Programming Academic Project." GitHub, 2025, github.com/arnav58/Deadcode-Removal-Complier. Accessed 15 Dec. 2025.
8. codelearner3012. "GitHub - Codelearner3012/C-Parser-For-Dead-Code-Removal." GitHub, 2025, github.com/codelearner3012/C-Parser-for-dead-code-removal. Accessed 15 Dec. 2025.