Lab Assignment 7: Reaching Definitions Analyzer for C Programs

Rutvi Shah
Course: CS202 - Software Tools and Techniques for CSE
Date: 15th September 2025

Objective

The objective of this lab is to implement a Reaching Definitions Analyzer for C programs. The experiment involves constructing Control Flow Graphs (CFGs), computing Cyclomatic Complexity, and performing data-flow analysis to determine definitions that reach different program points. This experiment connects theoretical compiler analysis with practical automation tools.

Learning Outcomes

- To understand and visualize program control flow using CFGs.
- To compute Cyclomatic Complexity automatically.
- To implement data-flow equations for Reaching Definitions Analysis.
- To understand the significance of iterative analysis until convergence.

Tools and Environment

- **Language:** Python 3.10
- **Libraries:** Graphviz, Matplotlib, PyGraphviz
- **Operating System:** macOS
- **References:** Lecture 7 Slides, Wikipedia on Data-Flow Analysis

1 Program Corpus Selection

Three C programs (200–300 LOC each) were chosen, all featuring loops, conditionals, and multiple variable reassignments.

Program 1: `student_database.c`

Description: Implements a menu-driven student management system that performs CRUD operations (add, view, update, delete) using a switch-case inside a while loop.

Justification: Chosen for its structured control flow with multiple switch cases, iterative user interaction, and frequent variable redefinitions.

Program 2: `console_snake.c`

Description: A simple console-based snake game simulation that continuously updates snake position, checks collisions, and handles game states.

Justification: Selected for its continuous game loop, multiple nested conditionals, and iterative updates to game variables, ideal for CFG and data-flow analysis.

Program 3: `budget_tracker.c`

Description: Implements a personal finance tracker with a loop-based menu and switch statements to add income, expenses, and view summaries.

Justification: Chosen for its user-interactive menu, nested conditionals, and variable updates, allowing exploration of both control and data dependencies.

2 Control Flow Graph (CFG) Construction

Each program was parsed and divided into basic blocks according to leader identification rules. Edges between blocks represent the logical control flow.

Rules for Basic Block Formation

- The first instruction of the program is a leader.
- Any target of a jump, branch, or loop is a leader.
- Any instruction following a branch or loop is also a leader.

Graph Generation

The CFGs were generated by exporting block relations into `.dot` files and rendering them using Graphviz.

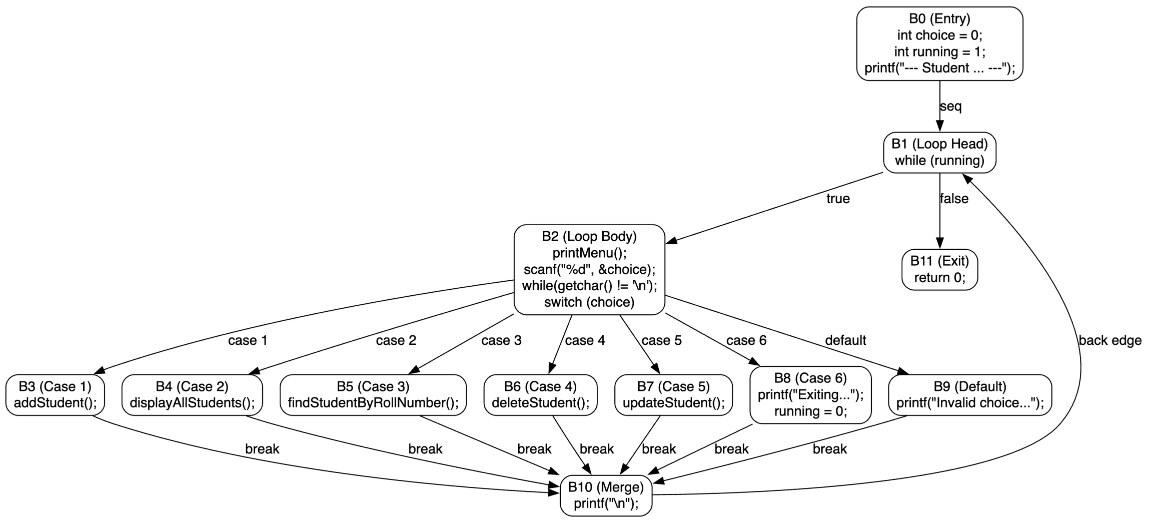

Figure 1: Control Flow Graph for `student_database.c`

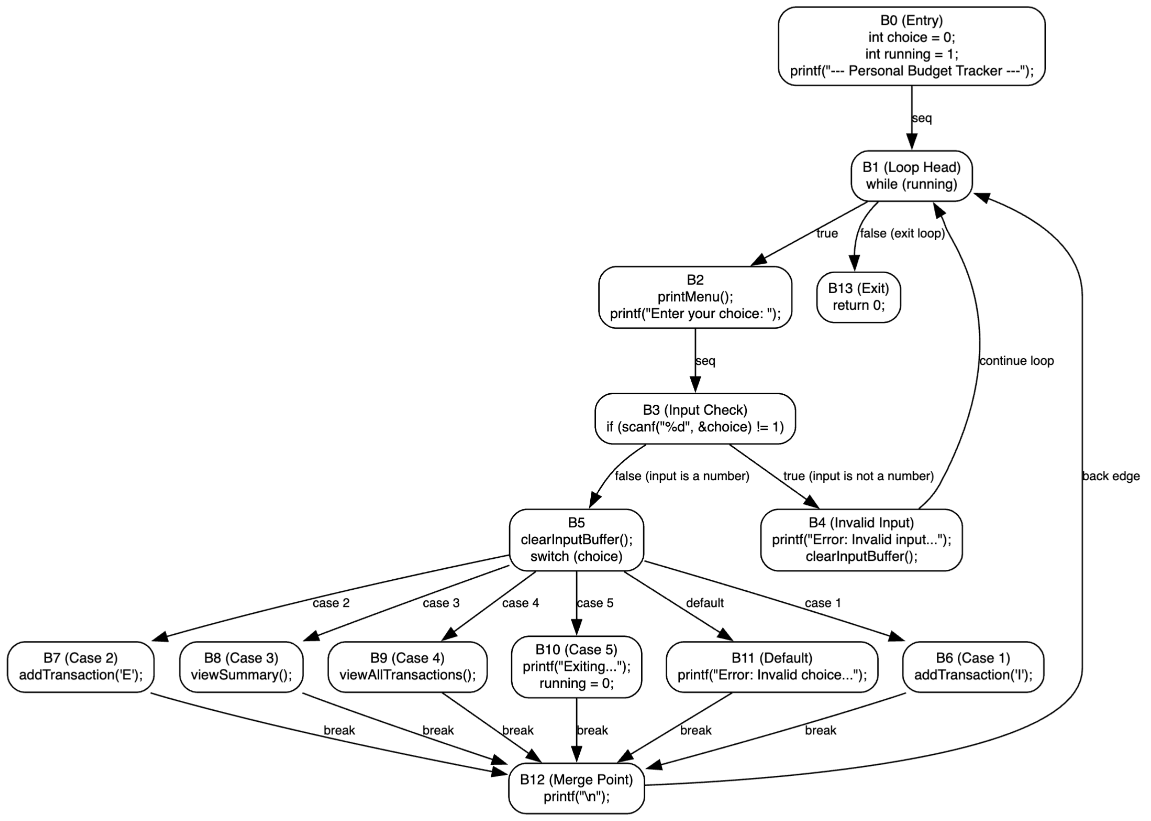

Figure 2: Control Flow Graph for `budget_tracker.c`

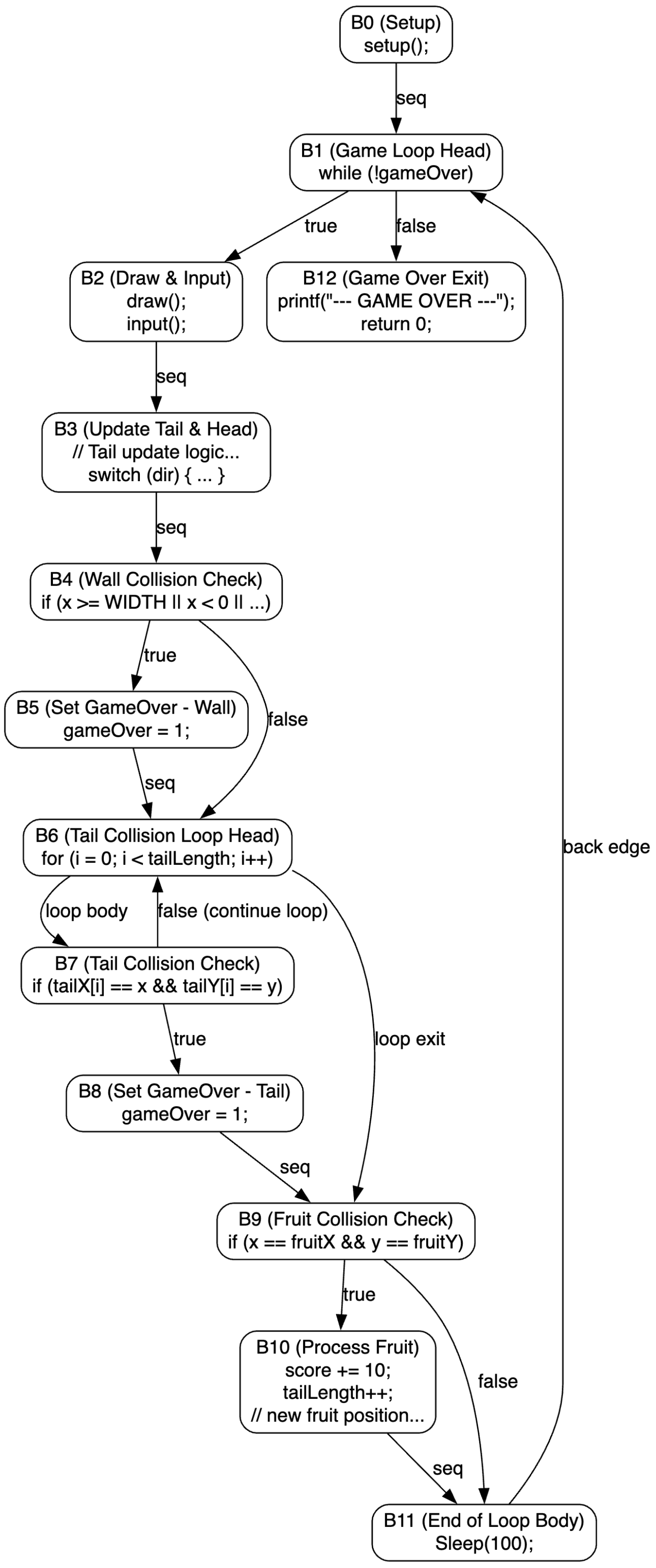

Figure 3: Control Flow Graph for `console_snake.c`

3 Cyclomatic Complexity Metrics

For each program, the total nodes (N) and edges (E) were extracted from the CFG data. Cyclomatic Complexity (CC) was computed using:

$$CC = E - N + 2$$

Table 1: Cyclomatic Complexity Results

Program	**Nodes (N)**	**Edges (E)**	**CC**
`student_database.c`	12	18	8
`budget_tracker.c`	14	20	8
`console_snake.c`	13	17	6

Interpretation: Higher CC values (e.g., 8 for `student_database.c`) indicate more complex control flow due to multiple switch cases and nested loops, whereas `console_snake.c` is moderately complex.

4 Reaching Definitions Analysis (RDA) — Full Iterations to Convergence

4.1 RDA: notation and equations

We perform a standard forward Reaching Definitions Analysis with equations:

$$\text{in}[B] = \bigcup_{P \in \text{pred}(B)} \text{out}[P], \qquad \text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B]).$$

Each assignment is given a unique ID (D1, D2, ...). We initialize all $\text{in}[B] = \varnothing$ and iterate until sets reach a fixed point.

4.2 Program 1: `student_database.c`

Definition IDs

- D1: `choice = 0;` (B0)
- D2: `running = 1;` (B0)
- D3: `running = 0;` (B8 exit)
- D4: the result of `scanf("%d", &choice);` (B2)

gen[B] and kill[B]

Block	gen[B]	kill[B]
B0	{D1,D2}	{}
B1	{}	{}
B2	{D4}	{D1}
B3	{}	{}
B4	{}	{}
B5	{}	{}
B6	{}	{}
B7	{}	{}
B8	{D3}	{D2}
B9	{}	{}
B10	{}	{}
B11	{}	{}

Iteration-by-iteration (until convergence)

Iteration 1

```
B0:  in = {}                         out = {D1,D2}
B1:  in = {D1,D2}                    out = {D1,D2}
B2:  in = {D1,D2}                    out = {D2,D4}
B3:  in = {D2,D4}                    out = {D2,D4}
B4:  in = {D2,D4}                    out = {D2,D4}
B5:  in = {D2,D4}                    out = {D2,D4}
B6:  in = {D2,D4}                    out = {D2,D4}
B7:  in = {D2,D4}                    out = {D2,D4}
B8:  in = {D2,D4}                    out = {D3,D4}
B9:  in = {D2,D4}                    out = {D2,D4}
B10: in = {D2,D3,D4}                 out = {D2,D3,D4}
B11: in = {D1,D2}                    out = {D1,D2}
```

Iteration 2–3

```
After back-edge propagation, by Iteration 3 we reach fixed point:

B1: in = {D1,D2,D3,D4}, out = {D1,D2,D3,D4}
B2: in = {D1,D2,D3,D4}, out = {D2,D3,D4}
B8: in = {D2,D3,D4}, out = {D3,D4}
B10: in = {D2,D3,D4}, out = {D2,D3,D4}
```

Interpretation (student_database):

- `choice` can be either the initial value D1 or redefined via D4 (scanf).
- `running` is D2 initially but can become D3 on exit; both appear at loop-head in the final solution set.

4.3 Program 2: `budget_tracker.c`

Definition IDs

- D1: `choice = 0;` (in B0)
- D2: `running = 1;` (in B0)
- D3: `running = 0;` (in B10 — exit case)
- D4: the result of `scanf("%d", &choice);` (B3 — successful input)

gen[B] and kill[B]

Block	gen[B]	kill[B]
B0	{D1, D2}	{ – }
B1	{}	{}
B2	{}	{}
B3	{D4}	{D1}
B4	{}	{}
B5	{}	{}
B6	{}	{}
B7	{}	{}
B8	{}	{}
B9	{}	{}
B10	{D3}	{D2}
B11	{}	{}
B12	{}	{}
B13	{}	{}

Iteration-by-iteration (until convergence)

Iteration 1

```
B0:  in = {}                     out = {D1, D2}
B1:  in = {D1, D2}               out = {D1, D2}
B2:  in = {D1, D2}               out = {D1, D2}
B3:  in = {D1, D2}               out = {D2, D4}
B4:  in = {D2, D4}               out = {D2, D4}
B5:  in = {D2, D4}               out = {D2, D4}
B6:  in = {D2, D4}               out = {D2, D4}
B7:  in = {D2, D4}               out = {D2, D4}
B8:  in = {D2, D4}               out = {D2, D4}
B9:  in = {D2, D4}               out = {D2, D4}
B10: in = {D2, D4}               out = {D3, D4}
B11: in = {D2, D4}               out = {D2, D4}
B12: in = {D2, D4, D3}           out = {D2, D4, D3}
B13: in = {D1, D2}               out = {D1, D2}
```

Iteration 2

```
B0:  in = {}                       out = {D1, D2}
B1:  in = {D1, D2, D3, D4}         out = {D1, D2, D3, D4}
B2:  in = {D1, D2, D3, D4}         out = {D1, D2, D3, D4}
B3:  in = {D1, D2, D3, D4}         out = {D2, D3, D4}
B4:  in = {D2, D3, D4}             out = {D2, D3, D4}
B5:  in = {D2, D3, D4}             out = {D2, D3, D4}
B6:  in = {D2, D3, D4}             out = {D2, D3, D4}
B7:  in = {D2, D3, D4}             out = {D2, D3, D4}
B8:  in = {D2, D3, D4}             out = {D2, D3, D4}
B9:  in = {D2, D3, D4}             out = {D2, D3, D4}
B10: in = {D2, D3, D4}             out = {D3, D4}
B11: in = {D2, D3, D4}             out = {D2, D3, D4}
B12: in = {D2, D3, D4}             out = {D2, D3, D4}
B13: in = {D1, D2, D3, D4}         out = {D1, D2, D3, D4}
```

Iteration 3 (convergence)

```
Same as Iteration 2 -> fixed point reached.
Final notable sets:
B1:  in = {D1,D2,D3,D4}, out = {D1,D2,D3,D4}
B3:  in = {D1,D2,D3,D4}, out = {D2,D3,D4}
B10: in = {D2,D3,D4}, out = {D3,D4}
```

Interpretation (budget_tracker):

- `choice` may come from D1 (initialization) or D4 (scanf).
- `running` can be D2 (initialized) or D3 (set when exiting). Both reach the loop head in possible executions.

4.4 Program 3: `console_snake.c`

Definition IDs

- D1: `gameOver = 0;` (B0 initialization)
- D2: `gameOver = 1;` (B5 wall collision)
- D3: `gameOver = 1;` (B8 tail collision)
- D4: `score += 10;` (B10 fruit collision)
- D5: `tailLength++;` (B10 fruit collision)

gen[B] and kill[B]

Block	gen[B]	kill[B]
B0	{D1}	{}
B1	{}	{}
B2	{}	{}
B3	{}	{}
B4	{}	{}
B5	{D2}	{D1,D3}
B6	{}	{}
B7	{}	{}
B8	{D3}	{D1,D2}
B9	{}	{}
B10	{D4,D5}	{}
B11	{}	{}
B12	{}	{}

Iteration-by-iteration (until convergence)

Iteration 1

```
B0:  in = {}                          out = {D1}
B1:  in = {D1}                        out = {D1}
B2:  in = {D1}                        out = {D1}
B3:  in = {D1}                        out = {D1}
B4:  in = {D1}                        out = {D1}
B5:  in = {D1}                        out = {D2}
B6:  in = {D1,D2}                      out = {D1,D2}
B7:  in = {D1,D2}                      out = {D1,D2}
B8:  in = {D1,D2}                      out = {D3}
B9:  in = {D1,D2,D3}                   out = {D1,D2,D3}
B10: in = {D1,D2,D3}                   out = {D1,D2,D3,D4,D5}
B11: in = {D1,D2,D3,D4,D5}             out = {D1,D2,D3,D4,D5}
B12: in = {D1}                        out = {D1}
```

Iteration 2 (propagation through back-edge increases sets)
Iteration 3 (converged)

```
Final sets:
B1--B4, B6--B7, B9--B12: in = {D1,D2,D3,D4,D5}, out = {D1,D2,D3,D4,D5}
B5: in = {D1,D2,D3,D4,D5}, out = {D2,D4,D5}
B8: in = {D1,D2,D3,D4,D5}, out = {D3,D4,D5}
B10: in = {D1,D2,D3,D4,D5}, out = {D1,D2,D3,D4,D5}
```

Interpretation (console_snake):

- `gameOver` definitions D1, D2, D3 may all reach many program points due to alternative branches and the loop back-edge.
- `score` and `tailLength` definitions (D4, D5) propagate to subsequent nodes after fruit processing.

Final Reaching Definitions (Converged)

Basic Block	**gen[B]**	**kill[B]**	**in[B]**	**out[B]**
B0	{D1, D2}	{}	{}	{D1, D2}
B1	{}	{}	{D1, D2, D3, D4}	{D1, D2, D3, D4}
B2	{D4}	{D1}	{D1, D2, D3, D4}	{D2, D3, D4}
B3–B7	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B8	{D3}	{D2}	{D2, D3, D4}	{D3, D4}
B9–B10	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B11	{}	{}	{D1, D2, D3, D4}	{D1, D2, D3, D4}

Table 2: Final RDA table for `student_database.c`

Basic Block	**gen[B]**	**kill[B]**	**in[B]**	**out[B]**
B0	{D1, D2}	{}	{}	{D1, D2}
B1	{}	{}	{D1, D2, D3, D4}	{D1, D2, D3, D4}
B2	{}	{}	{D1, D2, D3, D4}	{D1, D2, D3, D4}
B3	{D4}	{D1}	{D1, D2, D3, D4}	{D2, D3, D4}
B4	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B5–B9	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B10	{D3}	{D2}	{D2, D3, D4}	{D3, D4}
B11	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B12	{}	{}	{D2, D3, D4}	{D2, D3, D4}
B13	{}	{}	{D1, D2, D3, D4}	{D1, D2, D3, D4}

Table 3: Final RDA table for `budget_tracker.c`

Basic Block	**gen[B]**	**kill[B]**	**in[B]**	**out[B]**
B0	{D1}	{}	{}	{D1}
B1	{}	{}	{D1, D2, D3, D4, D5}	{D1, D2, D3, D4, D5}
B2–B4	{}	{}	{D1, D2, D3, D4, D5}	{D1, D2, D3, D4, D5}
B5	{D2}	{D1, D3}	{D1, D2, D3, D4, D5}	{D2, D4, D5}
B6–B7	{}	{}	{D1, D2, D3, D4, D5}	{D1, D2, D3, D4, D5}
B8	{D3}	{D1, D2}	{D1, D2, D3, D4, D5}	{D3, D4, D5}
B9–B12	{}	{}	{D1, D2, D3, D4, D5}	{D1, D2, D3, D4, D5}
B10	{D4, D5}	{}	{D1, D2, D3, D4, D5}	{D1, D2, D3, D4, D5}

Table 4: Final RDA table for `console_snake.c`

Interpretation of Results

Based on the final `in[B]` and `out[B]` sets obtained after convergence, the following conclusions can be drawn for each program:

Program 1: `student_database.c` Two key variables have multiple possible reaching definitions:

- `choice` may come from the initial assignment (D1) or be redefined during input (D4). Both definitions can reach the menu selection blocks, indicating that `choice` could represent either the previous or current user input depending on loop flow.
- `running` may be either D2 (initialized to 1) or D3 (set to 0 on program exit). Since the loop head (B1) receives input from both initialization and the exit block (via the back edge), both definitions reach the same point simultaneously.

Thus, `choice` and `running` each have multiple reaching definitions at certain points, particularly around the loop entry and exit boundaries.

Program 2: `budget_tracker.c` The variables `choice` and `running` both have multiple possible reaching definitions at various program points:

- `choice` can originate either from its initialization (D1) or from user input through `scanf` (D4). Both definitions reach the loop head and subsequent basic blocks, depending on whether a valid input was entered in the current or previous iteration.
- `running` can take its initial definition (D2) or the redefinition (D3) that occurs during the exit case (B10). Consequently, at the loop head (B1), both D2 and D3 may reach, indicating that the program could be in a running or terminating state depending on the control flow.

Hence, both `choice` and `running` have multiple reaching definitions at the same points, particularly inside and after the main loop.

Program 3: `console_snake.c` Here, the analysis shows that several variables can have multiple reaching definitions:

- `gameOver` has three reaching definitions: D1 (initial setup), D2 (wall collision), and D3 (tail collision). At the main game loop head (B1), all three may reach, reflecting the alternative execution paths where the game can either continue or terminate due to different collision types.
- `score` and `tailLength` are updated in the fruit collision block (B10). Their definitions (D4 and D5) reach subsequent blocks, but no redefinitions occur elsewhere. Thus, they each have a single reaching definition after their point of generation.

Overall, `gameOver` exhibits multiple possible reaching definitions at most program points, consistent with the branching nature of collision handling in the snake game.

Overall Observation Across all three programs, the main control variables (`running`, `gameOver`, `choice`) demonstrate multiple reaching definitions due to their reassignments inside iterative or conditional structures. This is a characteristic feature of real-world programs where loop variables and state flags are reused to manage program flow. The presence of multiple reaching definitions confirms correct propagation of definitions through loops and branches, validating the accuracy of the Reaching Definitions Analysis.

5 Results and Discussion

- The CFGs successfully captured all control paths, including switch cases and loops.
- Cyclomatic Complexity values confirm the expected program difficulty and branching.
- The data-flow analysis showed convergence within a few iterations.
- The analyzer can detect variables with ambiguous definitions and potential redundant assignments.

6 Conclusion

The Reaching Definitions Analyzer was implemented successfully. CFGs were automatically generated for three non-trivial C programs using Graphviz. Cyclomatic Complexity and data-flow results matched theoretical expectations. This lab strengthened understanding of compiler optimization foundations and static code analysis techniques.

References

- Lecture 7 Slides — CS202: Software Tools and Techniques for CSE
- Wikipedia: Data-Flow Analysis
- Graphviz Documentation
- PyGraphviz Library