

# Expanded Oral Exam Script: CFL Reachability & Datalog

Candidate

## I. Introduction (Approx. 2 Minutes)

**Speaker:** Good morning/afternoon. Today I will be presenting two fundamental frameworks used in static program analysis: **Context-Free Language (CFL) Reachability** and **Datalog**.

To give you some context, static analysis is all about reasoning about program behavior without actually running the code. We often model programs as graphs—where nodes are program points and edges represent control flow or data flow.

My agenda for this presentation is three-fold:

1. First, I will explain **CFL Reachability**. I will discuss why standard graph reachability isn't enough for precise analysis and how we use formal grammars to filter out invalid execution paths.
2. Second, I will introduce **Datalog**. This is a declarative logic language that allows us to specify these reachability problems very concisely, separating the specification of the analysis from its implementation.
3. Finally, I will touch upon the complexity of these algorithms and the expressiveness trade-offs we face.

I will use the whiteboard to illustrate the key algorithms and examples as we go.

---

## II. CFL Reachability (Approx. 6-7 Minutes)

**Speaker:** Let's start with the core problem: **Graph Reachability**. In a simple directed graph, we ask: *Is there a path from node  $s$  to node  $t$ ?* Standard algorithms like BFS or DFS can solve this in linear time,  $O(n + m)$ .

However, in program analysis, a "path" in the graph might not represent a valid execution. We need to filter out paths that are impossible at runtime.

*[Action: Go to the whiteboard] (Note: Draw a simple graph with nodes 1, 2, 3, 4. Draw edges representing function calls and returns.)*

**Speaker:** Consider a function call graph. If function `main` calls `foo`, and `foo` calls `bar`, when `bar` returns, it **must** return to `foo`. It cannot magically return to a different function. Standard reachability treats all edges equally and misses this constraint.

To fix this, we label the edges of our graph with symbols from an alphabet  $\Sigma$ . We then say a node  $t$  is reachable from  $s$  only if the string of labels along the path belongs to a specific Context-Free Language  $\mathcal{L}$ .

**Speaker:** The most important application of this is **Dyck Reachability**, or matched-parenthesis reachability. We define a grammar  $G$  that generates balanced parentheses. For example, a function call is an "open parenthesis"  $(_i$  and a return is a "close parenthesis"  $)_i$ .

[Action: Draw the Dyck Grammar on the board] (Note: Reference Slide 11:)

$$\mathcal{S} \rightarrow ({}_i\mathcal{S})_i \mid \mathcal{S}\mathcal{S} \mid \epsilon$$

This grammar says a valid path is either a matched pair  $(_i \dots)_i$ , a concatenation of valid paths  $\mathcal{S}\mathcal{S}$ , or an empty path  $\epsilon$ .

[Action: Draw the Graph Example from Slide 12] (Note: Draw nodes 1, 2, 3, 4. Edge 1-2 labeled  $(_1$ . Edge 2-3 labeled  $(_2$ . Edge 3-4 labeled  $)_2$ . Edge 4-1 labeled  $)_1$ .)

**Speaker:** Let's trace a path here. If we go from node 1 to 2, we "open" scope 1. From 2 to 3, we "open" scope 2. If we then traverse an edge labeled  $)_2$ , we "close" scope 2. This matches the inner parenthesis. Finally, if we see  $)_1$ , we close the outer scope. The path string is  $(_1(_2)_2)_1$ , which reduces to  $\epsilon$ . Therefore, node 1 reaches itself via a valid Dyck path.

If we had mismatched edges—say, opening scope 1 but trying to close scope 2—the path would be invalid and rejected by the analysis.

**Speaker:** We can apply this same logic to **Field Sensitivity** in pointer analysis. Imagine writing to a field 'x.f = y'. This is like "opening" a parenthesis for field 'f'. Reading from that field 'z = x.f' is like "closing" it. A data flow is valid only if writes and reads to fields match up correctly.

**Speaker: Complexity:** You might wonder, is this expensive? Yes. While standard reachability is linear, CFL reachability generally takes  $O(n^3)$  time. The algorithm is essentially a dynamic programming approach, very similar to the CYK parsing algorithm for strings. We basically add "summary edges" to the graph whenever we find a valid sub-path.

**The Limit:** We run into a hard theoretical limit if we try to be *too* precise. If we want to model **both** function calls (context sensitivity) AND heap accesses (field sensitivity) perfectly, we get **Interleaved Dyck Reachability**. This involves intersecting two Dyck languages. It is a known result that reachability for the intersection of two context-free languages is **undecidable**. So, in practice, we must approximate one of the two dimensions.

### III. Datalog (Approx. 6-7 Minutes)

**Speaker:** Now, let's shift gears to **Datalog**. While CFL reachability is the *concept* or the *algorithm*, Datalog is the **language** we use to express it. Datalog is a declarative logic programming language. In imperative languages like C++ or Python, we describe *how* to compute something. In Datalog, we only describe *what* implies what.

[Action: Erase board and write "Datalog Syntax"]

**Speaker:** A Datalog program is built from two things: **Facts** and **Rules**.

1. **Facts** represent our input data. For a graph, these are just the edges. For example, 'edge(1, 2)' is a fact.
2. **Rules** allow us to infer new information. A rule has a **Head** and a **Body**, separated by ':-', which is read as "if".

*[Action: Write the Transitive Closure example clearly] (Note: Reference Slide 12:)*

```
path(X, Y) :- edge(X, Y).  
path(X, Y) :- path(X, Z), edge(Z, Y).
```

**Speaker:** Let's break this down. The first rule is the base case: "There is a path from X to Y if there is an edge from X to Y". The second rule is the recursive step: "There is a path from X to Y if there is already a path to some node Z, and an edge from Z to Y".

**Speaker: Semantics:** How does the computer execute this? It uses **Least Fixed Point semantics**. It starts with the facts we know. It applies the rules to discover new facts. For example, if we know 'edge(1,2)' and 'edge(2,3)', the first rule tells us 'path(1,2)' and 'path(2,3)' are true. Then, the engine runs again. It sees 'path(1,2)' and 'edge(2,3)'. The second rule triggers, and we infer 'path(1,3)'. This repeats until no new facts can be generated.

**Speaker:** Now, what if we want to say something is reachable only if something else is **NOT** reachable? We need **Negation**. Consider this rule:

```
oneWay(X,Y) :- path(X,Y), not path(Y,X).
```

This says X is connected to Y, but Y cannot get back to X.

**Speaker:** Negation is dangerous. It can lead to paradoxes where there is no clear answer—like "Statement A is true if Statement A is false". To handle this, we restrict ourselves to **Stratified Datalog**. This means we organize the predicates into layers or "strata". We look at the **Precedence Graph** of the predicates. If a rule depends negatively on a predicate, that predicate must be in a lower layer. We fully compute the lower layer (the facts that are "not" true) *before* we move up to evaluate the rule containing the negation. This ensures the "not" is well-defined and fixed before we use it.

---

## IV. Conclusion (Approx. 1-2 Minutes)

**Speaker:** To wrap up, we have looked at two sides of static analysis.

1. **CFL Reachability** is the theoretical framework that allows us to make our graph traversals precise. It lets us model context-sensitivity (like function calls) and field-sensitivity by treating execution paths as strings in a language. It comes with a cost:  $O(n^3)$  complexity compared to linear time for standard reachability.
2. **Datalog** is the practical, declarative tool we use to implement these analyses. It is powerful enough to express any polynomial-time algorithm, including CFL reachability. While Datalog evaluation is generally exponential in the size of the rules, it is polynomial in the size of the data, which makes it efficient for program analysis where the graph is huge but the rules are few.

This combination—using formal grammars to define validity and Datalog to compute it—is the backbone of many modern static analysis tools.

Thank you for listening. I am ready for your questions.

# EXAM CHEAT SHEET

## Key Concepts & Definitions

### 1. CFL Reachability

- **Goal:** Filter invalid paths (e.g., mismatched calls/returns).
- **Standard Reachability:**  $O(n + m)$  (Linear).
- **CFL Reachability:**  $O(n^3)$  (Cubic, like CYK parsing).
- **Dyck Language:** Balanced parentheses. Call=(, Return=).
- **Undecidability:** Intersecting two Dyck languages (Context + Field sensitivity) is undecidable.

### 2. Datalog Syntax

- **Facts:**  $edge(1, 2)$  (Ground truth).
- **Rules:**  $Head : -Body$  (Inference).
- **Atom:**  $p(t_1, \dots, t_k)$ .
- **Ground Atom:** No variables (constants only).

### 3. Datalog Semantics

- **Herbrand Base:** All possible ground atoms.
- **Least Fixed Point:** Repeatedly apply rules until nothing changes.
- **Negation:** Requires *Stratified Datalog* to avoid paradoxes (no negation cycles).

### 4. Complexity

- **Data Complexity:** Polynomial in size of facts (Efficient).
- **Program Complexity:** Exponential in number of rules.
- **Expressiveness:** Capture all PTime algorithms.

## Drawings to Remember

### Dyck Graph (Context Sensitivity):

- $1 \xrightarrow{(1)} 2 \xrightarrow{(2)} 3 \xrightarrow{)2)} 4 \xrightarrow{)1)} 1$
- Path string:  $(1(2)2)_1 \rightarrow \epsilon$  (Valid).

### Transitive Closure (Datalog):

- $path(X, Y) : -edge(X, Y)$ .
- $path(X, Y) : -path(X, Z), edge(Z, Y) ..$