Inlining Design Report

1. Introduction

Inlining is a crucial transformation step performed on the Intermediate Representation (IR) *before* the final BASIC code emission. It corresponds to the first part of **Task 8**. Its purpose is to replace placeholder `CALL` instructions, generated during the initial code generation phase, with the actual IR code of the non-recursive functions and procedures being called. This process eliminates the overhead associated with function call mechanisms at runtime, aiming for faster execution at the cost of potentially larger code size.

2. Inliner Design

- **Approach:** The inlining process is implemented in the `Inliner` class (`inline.py`). It operates on a list of IR instruction strings.
- **Input Dependency:** The `Inliner` requires a pre-computed dictionary (`function_bodies`) that maps the unique IR names of non-recursive functions and procedures to their respective parameter lists (unique names) and their body IR code (as generated by the `CodeGenerator`).
- **Core Logic:** The `inline_calls` method iterates through the input IR, identifies `CALL` instructions using regular expressions, and performs the replacement based on the lecture slide method (`COS 341 L16.pdf`).
- **Assumptions:**
  - All variable, parameter, function, and procedure names in the input IR and `function_bodies` dictionary are already unique, as guaranteed by the preceding Semantic Analysis phase .
  - The functions/procedures being inlined are **not recursive**. Recursive calls are not handled by this inliner and would require different techniques (as mentioned in Chapter 9 of the textbook) .

---

3. Expected Input

The `Inliner` expects:

1. **Initial IR Code:** A `List[str]` containing the intermediate representation generated by the `CodeGenerator` , potentially including `target = CALL func_name(arg1, ...)` instructions.

2. **Function Body Information:** A `Dict[str, FunctionBodyInfo]`, where keys are the unique IR names of functions/procedures, and values are `FunctionBodyInfo` objects containing:
    - `params`: A `List[str]` of the unique IR names of the function/procedure's parameters.
    - `body_ir`: A `List[str]` of the IR instructions constituting the function/procedure's body (including `RETURN` statements for functions).

---

4. How Processing Works (`inline_calls` method)

1. **Iteration:** The method iterates through the input `ir_code` list.
2. **CALL Detection:** A regular expression identifies `CALL` instructions, capturing the target temporary variable (`tN`), the unique function/procedure name (`v_name_M`), and the argument list.
3. **Lookup & Precondition Check:** It looks up the `func_name` in the `function_bodies` dictionary. If not found (e.g., built-in or error), the `CALL` is skipped. Argument counts are checked against the stored parameter list. Recursion is assumed *not* to occur.
4. **Determine Return Target:**
    - It checks if the called function has `RETURN` statements in its body IR.
    - If it *is* a function (has `RETURN`s), it looks ahead at the *next* instruction in the main IR stream. If this next instruction assigns the `CALL`'s temporary target (`tN`) to a final variable (e.g., `v_result_P = tN`), then `v_result_P` becomes the `actual_return_target`. The subsequent assignment instruction will be skipped later.
    - If it's a function but no such assignment follows, the `CALL`'s temporary (`tN`) is the `actual_return_target`.
    - If it's a procedure (no `RETURN`s), the `actual_return_target` is `None`.
5. **Parameter Passing Code:** For each parameter `p` and corresponding argument `a` from the `CALL`, an assignment instruction `p = a` is generated .
6. **Adapt Body Code (`_adapt_body`):**
    - **Unique Labels:** The helper `_make_labels_unique` is called. It scans the copied body IR for `REM Lk` labels, generates new unique labels (e.g., `Lk_inline_X`), and replaces both the `REM` definitions and the corresponding `GOTO Lk` / `IF ... THEN Lk` targets within that copied block. This prevents label conflicts if the same function is inlined multiple times.
    - **Replace RETURN:** The uniquely-labeled body IR is scanned. Any `RETURN value` instruction is replaced with `actual_return_target = value`. If `actual_return_target` is `None` (procedure), `RETURN` instructions are simply omitted.

7. **Assemble Output:** The parameter assignment instructions and the adapted body code are appended to the output IR list.
8. **Skip Original Instructions:** The original `CALL` instruction is *not* copied. If a subsequent assignment was identified in step 4, it is also skipped by advancing the loop counter appropriately.
9. **Copy Non-CALL Instructions:** Any instruction that is not a `CALL` is copied directly to the output list.

---

5. Expected Output

The `inline_calls` method returns a **new** **`List[str]`** representing the intermediate representation where all valid, non-recursive `CALL` instructions have been replaced by the corresponding parameter assignments and adapted body code. This output IR is ready for the final BASIC emission phase.

Example (Simplified):

Input IR:

```
t1 = CALL v_add_1(5, 10)
v_result_2 = t1
STOP
```

Output IR (after inlining `v_add_1` with params `v_a_3`, `v_b_4` and body `t10 = v_a_3 + v_b_4; RETURN t10`):

```
v_a_3 = 5
v_b_4 = 10
t10 = v_a_3 + v_b_4
v_result_2 = t10  // RETURN replaced using final target
STOP
```

---

6. Key Logic & Limitations (from Slides)

- **Process:** Copy body, pass parameters via assignment, replace `RETURN` with assignment to target, remove `CALL` .
- **Limitation:** This method **does not handle recursion**. Recursive calls would need different handling (e.g., stack frames, as discussed in Chapter 9).

7. Advantages and Disadvantages (from Slides)

- **Advantages:** Generates potentially **faster** target code by eliminating call/return overhead (jumps, stack manipulation).
- **Disadvantages:** Requires complex **scope analysis** beforehand, **not applicable to recursion**, and can lead to significant **code size increase** due to repeated code blocks.