Code Generation Design Report

1. Introduction

The Code Generator is the fourth phase of the SPL compiler, following Semantic Analysis. Its primary responsibility is to translate the validated and type-annotated Abstract Syntax Tree (AST) into a lower-level Intermediate Representation (IR). This IR serves as a bridge between the source language structure and the final target code (BASIC). The IR format used resembles Three-Address Code (TAC), making it suitable for subsequent processing like function inlining and final code emission.

2. Code Generator Design

- **Approach:** A **syntax-directed translation** approach is implemented using a **visitor pattern**. The `CodeGenerator` class (`code_gen.py`) recursively traverses the AST provided by the Semantic Analyzer.
- **Structure:** For each relevant AST node type (defined in `ast_nodes.py`), there is a corresponding `_visit_NodeType` method within the `CodeGenerator` class. These methods generate the appropriate sequence of IR instructions.
- **Input Dependency:** The generator relies heavily on the information produced during semantic analysis:
  - A fully populated `SymbolTable` is required to look up the unique, generated IR names for variables, functions, and procedures.
  - Type annotations (`node_types` dictionary) attached to expression nodes during semantic analysis are used to guide the translation of operators and conditions.

---

3. Expected Input

The Code Generator expects the following inputs:

1. **Validated AST:** The root `ProgramNode` of the AST, which has already been successfully processed by the `SemanticAnalyzer` . This ensures all scope and type rules have been checked.
2. **Populated Symbol Table:** The `SymbolTable` instance returned by the `SemanticAnalyzer` , containing `SymbolInfo` for all declared identifiers, including their unique IR names (`unique_name`).

3. **Type Annotations:** A dictionary (`node_types`) mapping AST node IDs to their determined types ('numeric' or 'boolean'), generated during semantic analysis.

---

4. How Processing Works

The `generate` method initiates the process, focusing on the `main` program block as the entry point for execution.

1. **Recursive Traversal:** The `_visit` method dispatches calls to specific `_visit_NodeType` methods based on the current AST node's type.
2. **Temporary Generation:** The `_new_temp` method generates unique temporary variable names (e.g., `t1` , `t2` , ...) used to store intermediate results of expression evaluations.
3. **Label Generation:** The `_new_label` method generates unique label names (e.g., `L1` , `L2` , ...) required for control flow instructions.
4. **Instruction Emission:** The `_emit` method appends generated IR instruction strings to an internal list (`self.ir_code`).
5. **Expression Translation:** Expression nodes (`AtomNode` , `TermNode` , `ParenTermNode` , `UnaryOperationNode` , `BinaryOperationNode`) are flattened into sequences of IR instructions. Arithmetic and comparison operations generate instructions using the corresponding operators (`+` , `-` , `*` , `/` , `=` , `>`). Results are stored in temporary variables. Variable accesses use the `unique_name` from the symbol table.
6. **Control Flow Translation:**
   - `if` , `while` , and `do-until` statements are translated using the specific logic defined in `Phase 4 - code-gen.pdf` :
     - Labels are represented as `REM L_name` .
     - Conditional jumps (`IF...THEN L_target`) are generated based on evaluated conditions.
     - Unconditional jumps (`GOTO L_target`) are used to manage flow, particularly to implement the `if-then-else` structure without an explicit `ELSE` keyword and to loop in `while` statements.
7. **Logical Operator Translation:**
   - `and` and `or` operators are handled within `_generate_conditional_jump` using short-circuiting logic, generating multiple conditional jumps and intermediate labels as specified.
   - `not` is handled by swapping the target labels (`label_true` , `label_false`) in `_generate_conditional_jump` , effectively inverting the condition's jump behaviour without a dedicated IR operator.

8. **Function/Procedure Calls:** Calls generate a placeholder `CALL unique_name(args)` instruction. Function call results are assigned to a temporary variable; procedure calls use a dummy temporary target. Unique names are retrieved via `SymbolInfo`.
9. **Declarations:** Variable declarations (global, local, parameters) do not generate executable IR code; they are handled entirely during semantic analysis.

---

5. Expected Output (Intermediate Representation)

The `generate` method returns a `List[str]`, where each string is a single IR instruction. This IR resembles Three-Address Code but follows the specific syntax required by the project (e.g., `REM` for labels, `STOP` for halt, `PRINT` for output, `=` for assignment and equality).

Example Snippet (`x = (a plus 5)`):

```
t1 = 5
t2 = v_a_1 + t1
v_x_2 = t2
```

Example Snippet (`if (a > 0) { halt }`):

```
t1 = v_a_1
t2 = 0
IF t1 > t2 THEN L1
GOTO L2
REM L1
STOP
REM L2
```

---

6. Key Translation Logic (Summary from Phase 4)

The generator strictly follows the custom rules from `Phase 4 - code-gen.pdf`:

- Labels use `REM L_name` syntax.
- Assignment uses `=`.
- `halt` translates to `STOP`.
- `if-then-else` translation avoids `ELSE` by structuring jumps and code blocks.
- `if-then` translation follows a similar structure with jumps.
- `and` / `or` use short-circuiting conditional jumps.

- `not` is handled by swapping jump targets.
- `CALL` instructions are generated as placeholders.

---

7. Examples

==
EXAMPLE: Halt
==
--- Source Code ---
halt

--- Generated IR Code ---
STOP
==

==
EXAMPLE: Simple Assignment
==
--- Source Code ---
glob { counter }
proc {} func {} main { var {}
counter = 0
}

--- Generated IR Code ---
t1 = 0
v_counter_1 = t1
==

==
EXAMPLE: Arithmetic Expression
==
--- Source Code ---
glob { a b result }
proc {} func {} main { var {}
a = 10;

```
b = 5;
result = ((a mult 2) plus (b div 1))
}
```

--- Generated IR Code ---

```
t1 = 10
v_a_1 = t1
t2 = 5
v_b_1 = t2
t3 = 2
t4 = v_a_1 * t3
t5 = 1
t6 = v_b_1 / t5
t7 = t4 + t6
v_result_1 = t7
===========================================================================
```

```
=========================================================================
EXAMPLE: If-Then Statement
=========================================================================
```

--- Source Code ---

```
glob { x }
proc {} func {} main { var {}
x = 5;
if (x > 0) {
print "positive"
}
}
```

--- Generated IR Code ---

```
t1 = 5
v_x_1 = t1
t2 = 0
IF v_x_1 > t2 THEN L1
GOTO L2
REM L1
PRINT "positive"
```

```
REM L2
===========================================================================

=========================================================================
EXAMPLE: If-Then-Else Statement
=========================================================================
--- Source Code ---
glob { x status }
proc {} func {} main { var {}
x = 0;
if (x eq 0) {
status = 1
} else {
status = 0
}
}
```

```
--- Generated IR Code ---
t1 = 0
v_x_1 = t1
t2 = 0
IF v_x_1 = t2 THEN L1
t3 = 0
v_status_1 = t3
GOTO L3
REM L1
t4 = 1
v_status_1 = t4
REM L3
===========================================================================

=========================================================================
EXAMPLE: While Loop
=========================================================================
--- Source Code ---
glob { i }
proc {} func {} main { var {}
i = 5;
while (i > 0) {
print i;
```

```
i = (i minus 1)
}
}
```

```
--- Generated IR Code ---
t1 = 5
v_i_1 = t1
REM L1
t2 = 0
IF v_i_1 > t2 THEN L2
REM L2
PRINT v_i_1
t3 = 1
t4 = v_i_1 - t3
v_i_1 = t4
GOTO L1
REM L3
========================================================================

======================================================================
EXAMPLE: Do-Until Loop
======================================================================
--- Source Code ---
glob { count }
proc {} func {} main { var {}
count = 0;
do {
count = (count plus 1);
print count
} until (count > 5)
}
```

```
--- Generated IR Code ---
t1 = 0
v_count_1 = t1
REM L1
t2 = 1
t3 = v_count_1 + t2
```

v_count_1 = t3
PRINT v_count_1
t4 = 5
t5 = v_count_1 > t4
IF t5 = 0 THEN L1
==

==
EXAMPLE: Logical AND
==
--- Source Code ---
glob { a b flag }
proc {} func {} main { var {}
a = 1; b = 0; flag = 0;
if ((a > 0) and (b > 0)) {
flag = 1
}
}

--- Generated IR Code ---
t1 = 1
v_a_1 = t1
t2 = 0
v_b_1 = t2
t3 = 0
v_flag_1 = t3
t4 = 0
IF v_a_1 > t4 THEN L3
REM L3
t5 = 0
IF v_b_1 > t5 THEN L1
GOTO L2
REM L1
t6 = 1
v_flag_1 = t6
REM L2
==

==
EXAMPLE: Logical OR

===

--- Source Code ---

```
glob { valid error }
proc {} func {} main { var {}
valid = 0; error = 1;
if ((valid eq 1) or (error eq 1)) {
print "check needed"
}
}
```

--- Generated IR Code ---

```
t1 = 0
v_valid_1 = t1
t2 = 1
v_error_1 = t2
t3 = 1
IF v_valid_1 = t3 THEN L1
REM L3
t4 = 1
IF v_error_1 = t4 THEN L1
GOTO L2
REM L1
PRINT "check needed"
REM L2
```

===

===

EXAMPLE: Logical NOT

===

--- Source Code ---

```
glob { active }
proc {} func {} main { var {}
active = 0;
if (not (active eq 1)) {
print "inactive"
}
}
```

```
--- Generated IR Code ---
t1 = 0
v_active_1 = t1
t2 = 1
IF v_active_1 = t2 THEN L2
GOTO L2
REM L1
PRINT "inactive"
REM L2
================================================================================

==============================================================================
EXAMPLE: Procedure Call
==============================================================================
--- Source Code ---
glob { g }
proc {
pdef setg(val) { local {} g = val }
}
func {}
main { var { localval }
localval = 42;
setg(localval)
}
```

```
--- Generated IR Code ---
t1 = 42
v_localval_1 = t1
t2 = CALL v_setg_1(v_localval_1)
================================================================================

==============================================================================
EXAMPLE: Function Call
==============================================================================
--- Source Code ---
glob { result input }
proc {}
func {
fdef square(n) { local { sq }
sq = (n mult n);
```

```
halt; // Need one instruction before return
return sq
}
}
main { var {}
input = 7;
result = square(input)
}
```

```
--- Generated IR Code ---
t1 = 7
v_input_1 = t1
t2 = CALL v_square_1(v_input_1)
v_result_1 = t2
=============================================================================
```

```
===========================================================================
EXAMPLE: Complex Example
===========================================================================
--- Source Code ---
glob { i sum }
proc {}
func {}
main { var {}
sum = 0;
i = 1;
while (i > 6) { // Equivalent to while i < 6, using > for test
sum = (sum plus i);
if (sum > 10) {
print "sum exceeded 10";
halt // Use halt instead of break
} else {
print "sum is ok"
};
i = (i plus 1)
};
print "final sum";
print sum
}
```

```
--- Generated IR Code ---
t1 = 0
v_sum_1 = t1
t2 = 1
v_i_1 = t2
REM L1
t3 = 6
IF v_i_1 > t3 THEN L2
REM L2
t4 = v_sum_1 + v_i_1
v_sum_1 = t4
t5 = 10
IF v_sum_1 > t5 THEN L4
PRINT "sum is ok"
GOTO L6
REM L4
PRINT "sum exceeded 10"
STOP
REM L6
t6 = 1
t7 = v_i_1 + t6
v_i_1 = t7
GOTO L1
REM L3
PRINT "final sum"
PRINT v_sum_1
============================================================
```