Phase 3 Report — Intermediate Code Generation for the C-minus Compiler

1 Introduction

Phase 3 of the C-minus compiler project focuses on implementing the **Intermediate Code Generation (ICG)** stage. Compared to Phase 1 (lexical analysis) and Phase 2 (syntactic analysis), this phase is substantially more complex because it must integrate:

- syntactic structure,
- semantic correctness,
- memory layout and address management,
- and sequential generation of machine-independent code.

In this stage, the parse tree produced by the parser is translated into a sequence of **Three-Address Code (TAC)** instructions. These instructions form the intermediate representation that can be interpreted by the provided Tester tool, or potentially translated into real machine code.

2 Objectives of Intermediate Code Generation

Based on the project specification [**?**], and our implementation, the primary goals of this phase are:

2.1 1. Bridge Between Parsing and Execution

Intermediate code serves as a high-level machine-independent representation. It converts structured C-minus constructs into simple TAC operations.

2.2 2. Semantic Verification

While generating intermediate code, our compiler also performs **mandatory semantic checks**, including:

- use of undefined identifiers,
- illegal void variable declarations,
- mismatched argument counts,
- invalid use of `break`,

- operand type mismatches,
- mismatched argument types in function calls.

Detected semantic errors are written to `txt.errors_semantic`.

2.3 3. Memory Management

The code generator manages memory layout by allocating:

- global variable addresses,
- temporary addresses,
- runtime stack sections (where applicable),

with alignment rules (e.g., integer values require 4-byte alignment).

2.4 4. Intermediate Representation

Expressions, control flow constructs, and assignments are decomposed into TAC instructions such as:

```
(ASSIGN, ...)
(ADD, ...)
(SUB, ...)
(MULT, ...)
(LT, ...)
(JP, ...)
(JPF, ...)
(PRINT, ...)
```

This simplifies implementation, debugging, and potential optimization.

3 System Architecture and Implementation

ICG is integrated directly into the parser pipeline, making the compiler a **single-pass compiler**, as required by the project specification.

3.1 1. The Code Generator (code_gen.py)

This module implements the core of intermediate code generation.

Responsibilities

- Maintaining program memory for TAC instructions,
- Managing a semantic stack for operands, operators, and temporary addresses,
- Allocating memory for variables and temporaries,
- Communicating with supporting modules: `RuntimeStack`, `SymbolTable`, `ActionManager`, `RegisterFile`.

Key Functions

- **action(act, *args)**: execute the semantic routine associated with a grammar event,
- **instruction_push / instruction_insert**: append TAC instructions to program memory,
- **get_next_temp_address**: allocate new temporary memory,
- **get_next_data_address**: allocate new data memory,

The generator also supports an implicit built-in function `output(int)`.

3.2 2. TAC Instruction Definitions (instructions.py)

This module defines the TAC instruction set used by the compiler. Examples include:

```
ASSIGN, ADD, SUB, MULT, LT, EQ,
JP, JPF, PRINT
```

The instructions must follow the exact formatting required by the Tester tool.

3.3 3. Supporting Modules

ActionManager Maps action symbols from the grammar to semantic routines. Handles assignments, arithmetic operations, control flow, and function handling.

SymbolTable Stores metadata for variables, arrays, and functions. Responsible for semantic checks such as undefined identifiers and type matching.

RuntimeStack Simulates stack behavior for function calls. (Used in our extended implementation; not required in minimal version.)

RegisterFile Maintains temporary registers used during TAC generation.

4 Semantic Error Handling

Phase 3 requires reporting six categories of semantic errors, as described in the project documentation [?]. Our implementation supports all required errors, specifically:

- undefined identifiers or functions,
- illegal use of `void` for variables or arrays,
- mismatched number of arguments,
- invalid `break` usage,
- type mismatches between operands,
- mismatched types of actual and formal parameters.

If semantic errors exist:

- they are written to `txt.errors_semantic`, and
- no intermediate code is produced; instead, `txt.output` contains:

    The output code has not been generated

5 Three-Address Code Production

The TAC generated by the compiler follows these rules:

- each line begins with a line number, followed by a tab,
- operands use immediate values (#n), direct addressing (n), or indirect addressing (n@),
- all opcodes are uppercase.

Example:

```
0	(JP, 9, ,)
1	(ASSIGN, #1, 100, )
2	(MULT, 104, 100, 500)
...
```

The Tester uses this format and assigns the actual execution line numbers.

6 Challenges

- Designing and aligning semantic hooks with the parser state machine,
- Correct management of memory and temporary addresses,
- Handling nested control-flow structures,
- Ensuring semantic errors are detected and reported without interrupting the pass-one pipeline,
- Maintaining code correctness under the strict formatting rules of the Tester tool.

7 Conclusion

In Phase 3, the C-minus compiler was extended to support full intermediate code generation. The implementation:

- produces TAC for expressions, assignments, conditionals, loops, and function calls,
- performs all mandatory static semantic checks,
- manages memory allocation and addressing,
- integrates code generation into the parser in a single pass.

The result is a functioning pass-one compiler capable of generating correct, Tester-compatible intermediate code for the C-minus language.

Contributors

Mohammad-Saeed Arvanaghi, Herbod Pourali, and Ilya Farhangfar

References

[1] Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley.

[2] Terence Parr. *The ANTLR Parser Generator Documentation*.