Assignment 3: Semantics & Intermediate Code

CS 464/5607: Compiler Design

Spring 2026

1 Introduction

In this assignment, you will implement two critical phases of the compiler pipeline:

1. **Phase 3: Semantic Analysis**: You will traverse the Abstract Syntax Tree (AST) to verify context-sensitive properties. This includes ensuring variables are declared before use, checking type compatibility in assignments, and verifying function arguments.
2. **Phase 4: Intermediate Representation (IR) Generation**: Once the program is validated, you will translate the AST into a linear, machine-independent Intermediate Representation (Three-Address Code) that serves as the bridge to final code generation.

2 Setup and Files

The complete implementation of the lexer and the parser is provided for this assignment

- `lib/`: Contains the **provided reference implementation** of Phase 1 and 2.
  - `lexer.l`: The reference Lexer.
  - `parser.y`: The reference Bison parser.
  - `ast.c / ast.h`: The reference AST implementation.

Part 1: Semantic Analysis

3 Error Reporting

To facilitate automated testing, semantic errors must be reported in a strictly standardized format.

3.1 The `errors.h` Interface

We have provided `errors.h` and `errors.c`, which define the `ErrorType` enum. **Note:** The specific error codes (e.g., `ERR_UNDECLARED_VAR`, `ERR_TYPE_MISMATCH_OP`) are documented in the comments within `errors.h`. Please review that file to understand which code corresponds to which semantic violation.

3.2 Using `report_error`

You must strictly use the provided helper function for all error output:

```
void report_error(ErrorType type, int lineno);
```

Crucial: Do not use `printf` manually for errors. The test suite relies on the specific string format generated by this function. Passing the correct `ErrorType` ensures your compiler matches the "Golden Output" required for grading.

4 The Symbol Table

The Symbol Table is the central data structure for Semantic Analysis. It tracks identifiers (variables, functions, arrays) and their associated attributes (types, scope, parameters) throughout the program.

4.1 Files

- `symbol_table.h`: Defines the API and structures.
- `symbol_table.c`: You must implement the logic here.

4.2 API Overview

You are required to implement the following functions to manage scope and symbols:

- `init_symbol_table()`: Initializes the global scope and data structures.
- `push_scope()`: Enters a new scope (e.g., entering a function body or `if` block).
- `pop_scope()`: Exits the current scope and removes local symbols.
- `insert_symbol(...)`: Adds a new symbol to the current scope. You must handle checks for redeclaration here (e.g., declaring `int x` twice in the same block).
- `lookup_symbol(...)`: Searches for a symbol starting from the current scope and moving outward to global scope.
- `lookup_local_symbol(...)`: Searches for a symbol *only* in the current scope (useful for detecting redeclarations).

5 Semantic Analysis

5.1 Entry Point

```c
int semantic_analysis(ASTNode* root);
```

This function is called by the driver after parsing is complete. It should traverse the AST, perform all necessary checks, and return the **total number of semantic errors** found.

5.2 Implementation Hints

You will need to write a recursive traversal function that visits every node in the AST.

- **Scope Management:** Ensure you call `push_scope()` and `pop_scope()` when entering/leaving blocks (functions, loops, conditionals).
- **Type Checking:** For binary operations (like `+` or `<`), ensure both operands are of compatible types (e.g., both `int`).
- **Function Calls:** Verify that the function exists, and that the number and types of arguments match the definition.

Note: For a complete list of the specific semantic rules you must enforce, please inspect the `ErrorType` enum in `errors.h`. The comments within that file provide detailed descriptions of the errors you are expected to handle.

Suggested Helpers: It is highly recommended to create helper functions to keep your code clean. Consider helpers for:

- Determining the result type of an expression node.
- Checking compatibility between two types.
- Validating function arguments against formal parameters.

Part 2: Intermediate Representation (IR) Generation

Once the AST has been validated and confirmed to be free of semantic errors, the compiler proceeds to the final phase of this assignment: generating the **Intermediate Representation (IR)**.

6 Overview of IR

An Intermediate Representation is a machine-independent version of the code that sits between the high-level source and the low-level machine code. IR decomposes complex expressions into a sequence of simple instructions. This linear structure makes it much easier to perform optimization and final code generation.

7 The IR Structure (`ir.h`)

The interface for the IR is defined in `ir.h`. You will need to inspect this file closely to understand the available operations.

7.1 Operations and Structure

The `IROp` enum defines the instruction set for our virtual machine. It includes arithmetic (`IR_ADD`, `IR_SUB`), control flow (`IR_IFZ`, `IR_GOTO`), and function management (`IR_CALL`, `IR_RET`). **Note:** Please refer to the comments in `ir.h` for the comprehensive list of supported operations. The instructions are stored as a linked list of `IRInst` structures:

```c
typedef struct IRInst {
    IROp op;              // The operation (e.g., IR_ADD)
    char *arg1;           // First operand (e.g., "x")
    char *arg2;           // Second operand (e.g., "5")
    char *result;         // Result destination (e.g., "t0")
    struct IRInst *next;
} IRInst;
```

7.1.1 Example Translation

A high-level statement like `x = a + 5;` might be represented internally as:

```
// Represents: t0 = a + 5 (ADD a 5 t0)
create_instruction(IR_ADD, "a", "5", "t0");

// Represents: x = t0 (ASSIGN t0 x)
create_instruction(IR_ASSIGN, "t0", NULL, "x");
```

7.2 API Implementation (ir.c)

You must implement the helper functions declared in `ir.h` inside the `ir.c` file. These include:

- `create_instruction`: Allocates and initializes a new `IRInst`.
- `append_instruction`: Adds an instruction to the end of the linked list.
- `new_temp()`: Generates a unique temporary variable name (e.g., `t0`, `t1`).
- `new_label()`: Generates a unique label name (e.g., `L0`, `L1`).

Tip: You are strongly encouraged to inspect the provided test cases in `tests/`. Comparing the input source code with the expected output files will give you a clear understanding of how specific constructs (like `while` loops or array access) should be translated.

8 Generating IR (ir_generator.c)

The core logic for this phase resides in `ir_generator.c`. You must implement the function:

```
IRInst* generate_ir_from_ast(ASTNode* root);
```

8.1 Implementation Hints

- **Traversal:** Similar to semantic analysis, you will traverse the AST recursively. However, instead of checking for errors, you will be building a list of instructions.
- **Temporaries:** For binary operations (like `3 * x`), you cannot store the result directly in the final variable. You must generate a temporary variable using `new_temp()` to hold the intermediate result.
- **Control Flow:** For `if` and `while` nodes, you will need to generate labels using `new_label()` and insert `IR_IFZ` (Branch if Zero) and `IR_GOTO` instructions to manage the execution flow.

Note: For Part B (IR Generation), you may assume that all variable names in the source program are globally unique. You do not need to handle complex name mangling for scoping; simply using the variable name provided in the AST node (e.g., `node->data`) is sufficient.

9 Building and Testing

9.1 Running Tests

We have provided a Python automation script. To build your project and run the tests, simply execute:

```
python run_tests.py
```

This script will automatically run the `makefile` to compile your code and then execute the binary against the test suite.

9.2 Testing with Custom Inputs

You are encouraged to create your own test cases to debug specific issues or test edge cases.

1. Create a text file (e.g., `my_test.txt`) with your source code.
2. Open your terminal and run `make` to ensure your compiler is built.
3. Run the compiler manually by passing the file path as an argument:

```
./build/compiler my_test.txt
```

This will print the output of all phases (Syntax, Semantics, IR) directly to your terminal, allowing you to debug specific errors.

9.3 Platform Specifics (Windows)

If you are developing natively on Windows (using MinGW/Git Bash) rather than WSL or Linux, you must make two small modifications to handle file extensions correctly.

1. Modify `Makefile` (Line 7)

Windows executables require the `.exe` extension. Update the `TARGET` variable:

```
# Change:
TARGET = $(BUILD_DIR)/compiler

# To:
TARGET = $(BUILD_DIR)/compiler.exe
```

2. Modify `run_tests.py` (Line 9)

The Python automation script needs to know the exact name of the executable to launch it.

```
# Change:
EXECUTABLE = os.path.join("build", "compiler")

# To:
EXECUTABLE = os.path.join("build", "compiler.exe")
```

9.4 Grading Distribution

- **70% Visible Test Cases:** These tests are provided in the `tests/` folder. Passing them ensures your compiler handles standard semantic rules correctly.
- **30% Hidden Test Cases:** These test edge cases, robust error recovery, and complex scoping scenarios.

10 Submission Instructions

...