MiniTS: A Compiler for a TypeScript-Inspired Class-Based Language: User Manual

**Team Members:**

**Russell Liu**

[russl8@my.yorku.ca](mailto:russl8@my.yorku.ca)

219616150

**Jeffrey Jiang**

[jeffj4@my.yorku.ca](mailto:jeffj4@my.yorku.ca)

218756668

# Table of Contents

[**Table of Contents 2**](#_dctt7qwujoq)

[***1 Input Languages 4***](#_qqux1ejg1ona)

[**1.1 Structure of a Program 4**](#_qezldr6lymza)

[*Input Language: 4*](#_iwzv53pmuq2i)

[*Key Constructs 4*](#_f391b3vqgw0g)

[**1.2 List of Advanced Programming Features 6**](#_l24amq1ru3na)

[*Feature 1: Inheritance 6*](#_9gippd2kyua)

[*Feature 2: For-Loops 6*](#_zcki1nhq1k19)

[*Feature 3: Scoping 6*](#_5k5l5fjtp2gb)

[*Feature 4: Function Declaration & Invocation 7*](#_ydptm6fe5ukz)

[*Feature 5: Lists 8*](#_oldjjba6oiz)

[***2 Output Structure 9***](#_2hkh3fbe3oq1)

[**Program has no semantic errors 9**](#_adzuobdtbjr4)

[**Program has semantic errors 9**](#_m4em5tahf6xx)

[***3 Compiler Workflow and Architecture 11***](#_weht0miwwlaj)

[VariableBindingVisitor 11](#_nxjk1rjv29k6)

[TypeCheckerVisitor 11](#_ecuvumixrdvi)

[***4 Justification of Output 12***](#_i5875yq1df63)

[**4.1 Variable declarations and assignments must match their declared type 12**](#_5f210sra1mnk)

[*4.1.1 Primitave Variables 12*](#_x9mkw9u009d5)

[*4.1.2 Lists 13*](#_1eopuv6jdffu)

[**4.2 Function return expressions must match their decleared return type 14**](#_44o7hije2p2l)

[**4.3 Function argument types must match parameter types 15**](#_m4cboo6yaxj)

[**4.4 Binary and unary expressions must use valid operand types 16**](#_yrat9ifbo2bx)

[*4.4.1 Arithmetic (+, -, /, \*, %): Both sides must be integers 16*](#_xiel6rpa10ke)

[*4.4.2 Logical (&&, ||, !): Operands must be boolean 17*](#_qsqjjkjzeyf)

[*4.4.3 Relational (<=, <, >, >=): Both sides must be integers. 17*](#_vkkssw5qkf6)

[*4.4.4 Equality (==, !=) – Both sides must be of the same type 18*](#_ykd819ewxriu)

[**4.5 If-statement and while-loop conditions must be boolean 18**](#_om8hqrhh0xne)

[**4.6 For-loop expressions must be of specific types 19**](#_tgoir2ubynwq)

[**4.7 Variables must be declared before use 20**](#_ozjcjwnlxyrh)

[**4.8 Variables cannot be declared more than once in the same scope 21**](#_8yjrmwwe4nq)

[**4.9 Functions and Classes cannot be redefined 21**](#_jysu9rg44tty)

[*4.9.1 Function redeclaration error 21*](#_xebl89mce2ea)

[*4.9.2 Class redeclaration error 22*](#_7tmkuqmtp55k)

[**4.10 Function calls must match their declarations 23**](#_e2q63sze5wcy)

[*4.10.1 Functions must be declared to be invoked 23*](#_owgbiwic4v5)

[*4.10.2 Function calls must include all parameters from declaration 23*](#_tusw9skd6ubp)

[*4.10.3 Function parameters must match declration parameter types 24*](#_chld71g1j101)

[***5 Summary of Submitted Examples 25***](#_yu3ve7grkov2)

[**5.1 Highlights of Example Input 1 25**](#_pcsk7vnbkzje)

[**5.2 Highlights of Example Input 2 25**](#_r7ol5zsitpmq)

[**5.3 Highlights of Example Input 3 25**](#_qpj6806gqed2)

[**5.4 Highlights of Example Input 4 25**](#_jzp9axhd3eoh)

[**5.5 Highlights of Example Input 5 25**](#_1y6gnf1yrcuj)

[**5.6 Highlights of Example Input 6 25**](#_t30ct4zewdzx)

[**5.7 Highlights of Example Input 7 25**](#_dyb4fzn28rsl)

[**5.8 Highlights of Example Input 8 25**](#_av2teecqdnlc)

[**5.9 Highlights of Example Input 9 25**](#_yt09qp59w4ts)

[**5.10 Highlights of Example Input 10 26**](#_mhqd2r7wlxjq)

[***6 Miscellaneous Features 26***](#_ag8gwccpsnc6)

[**6.1 Formatted HTML Report 26**](#_dhhl4a5jpgcm)

[**6.2 Multiple Input File Support 26**](#_3i82ozl52fem)

[**6.3 Error Highlighting 27**](#_5eptnfyadi8j)

[**6.4 Code Metrics 27**](#_cwhayycynsbb)

[**6.5 Log Retention 28**](#_i6fzb3w239tk)

[***7 Limitations 29***](#_hwflkgyxuls2)

[Not Supported: 29](#_f4cdy3bjd2yi)

[Just Out of Scope (due to time constraints): 29](#_ydkpwygunvr8)

# Input Languages

## Structure of a Program

| **class** Superclass {  c : list[char] = "hello";  i : int = 0;  } **class** Overview **extends** Superclass {   **function** **f** (i : int, j : int) : **int** {  **return** i + j;  }  **for** (j : int = 1; j <= 5; j = j + 1) {  i = f(i, j); *// i = 1+2+3+4+5 = 15*  }  j : int = 0;  **if**(i == 15) {  j = i;   } } |
| --- |

*Figure 1: Structure of input program*

### Input Language:

A typical input program accepted by the compiler is class-based. Meaning that it consists of one or more class definitions, each of which can inherit from a superclass. Within these classes, users can:

1. Declare and assign variables
2. Define functions
3. Use control flow statements such as loops and conditionals.

### Key Constructs

#### Classes and Inheritance

Programs are organized around classes. All statements and functions **must** be declared within a class. A class can inherit attributes and functions from another by extending it.

#### Variable Declarations and Assignments

Variables are strongly typed and can optionally be initialized during assignment. The supported types are ***int***, ***char***, ***bool***, and ***list*** parameterized by int, char, or bool.

#### Scoping

Variables follow lexical scoping rules: variables declared inside a function or block (such as a for loop or if statement) are local to that block. Class-level variables are accessible within all methods of the class and by its subclasses unless shadowed.

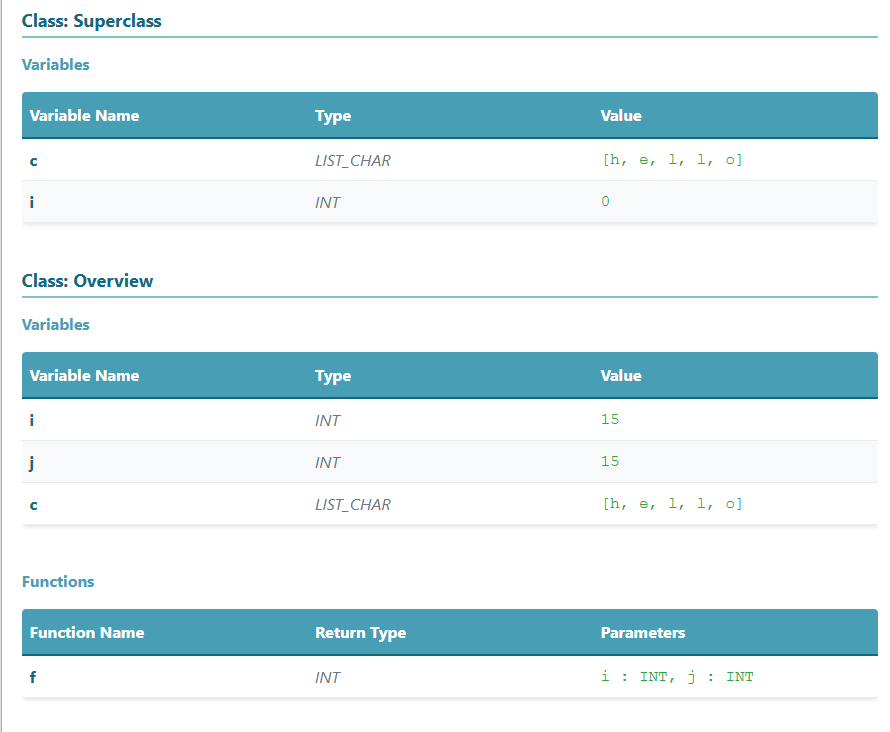
#### Functions

Functions support zero or more parameters and **must** specify a return type. All parameters and return values must be explicitly typed.

#### 

#### Control Flow

The language currently supports for loops, while loops, and if statements.



*Figure 2: Evaluated variables of input program (Refer to Figure 1)*

## List of Advanced Programming Features

### Feature 1: Inheritance

***Example file(s): inventory.txt***

Inheritance enables automatic access to ***all*** variables and functions defined in the superclass. To inherit attributes from another class, append the keyword *extends <ClassName>* to the class declaration.

| **class A {**  i : int = 0;  **}**  **class** B **extends** A { *// Inherits attributes from class A*  j : int = i + 1; *// j = 1*  } |
| --- |

### Feature 2: For-Loops

***Example file(s): task\_manager.txt***

The for loop follows the structure:

| **for** (<declaration>; <boolean\_expression>; <assignment>) {   *// loop body* } |
| --- |

Example:

| **j : int = 0;**  **for** (i: int = 0; i < 5; i = i + 1) {  k : int = j + i; } |
| --- |

### Feature 3: Scoping

***Example file(s): gaming.txt***

Variables are scoped to the block in which they are declared. Identifiers can be reused in different scopes without conflict. Functions and loops both introduce new scopes.

#### a) Scoping in Loops

| **class** Main {  x: int = 100;    **function** loopTest(): **int** {   **for** (x: int = 0; x < 10; x = x + 1) {   *// 'x' here is local to the loop*  }  }   x = 10; // refers to the class-level 'x' } |
| --- |

The loop-level x shadows the class-level x only within the loop body.

#### b) Scoping in Functions

| **class** Main {  x: int = 42;   **function** **print**(x: int): **int** {   **return** x; *// Refers to the parameter*   }  x = print(x); // x == 42 } |
| --- |

The parameter x shadows the outer x for the entire function body.

### Feature 4: Function Declaration & Invocation

***Example file(s): fleet.txt, student.txt***

Functions are declared using the following syntax:

| **function** functionName(param1: **type**, param2: **type**): **returnType** {   // function body } |
| --- |

Example:

| **function** f(i: int): **bool** {   **return** i == 0;  }  x: bool = f(1); |
| --- |

### Feature 5: Lists

***Example file(s): weather.txt***

Lists can be declared and initialized using the following syntax:

| **var**: list[**type**] = [...]; **var** = [...]; |
| --- |

Example:

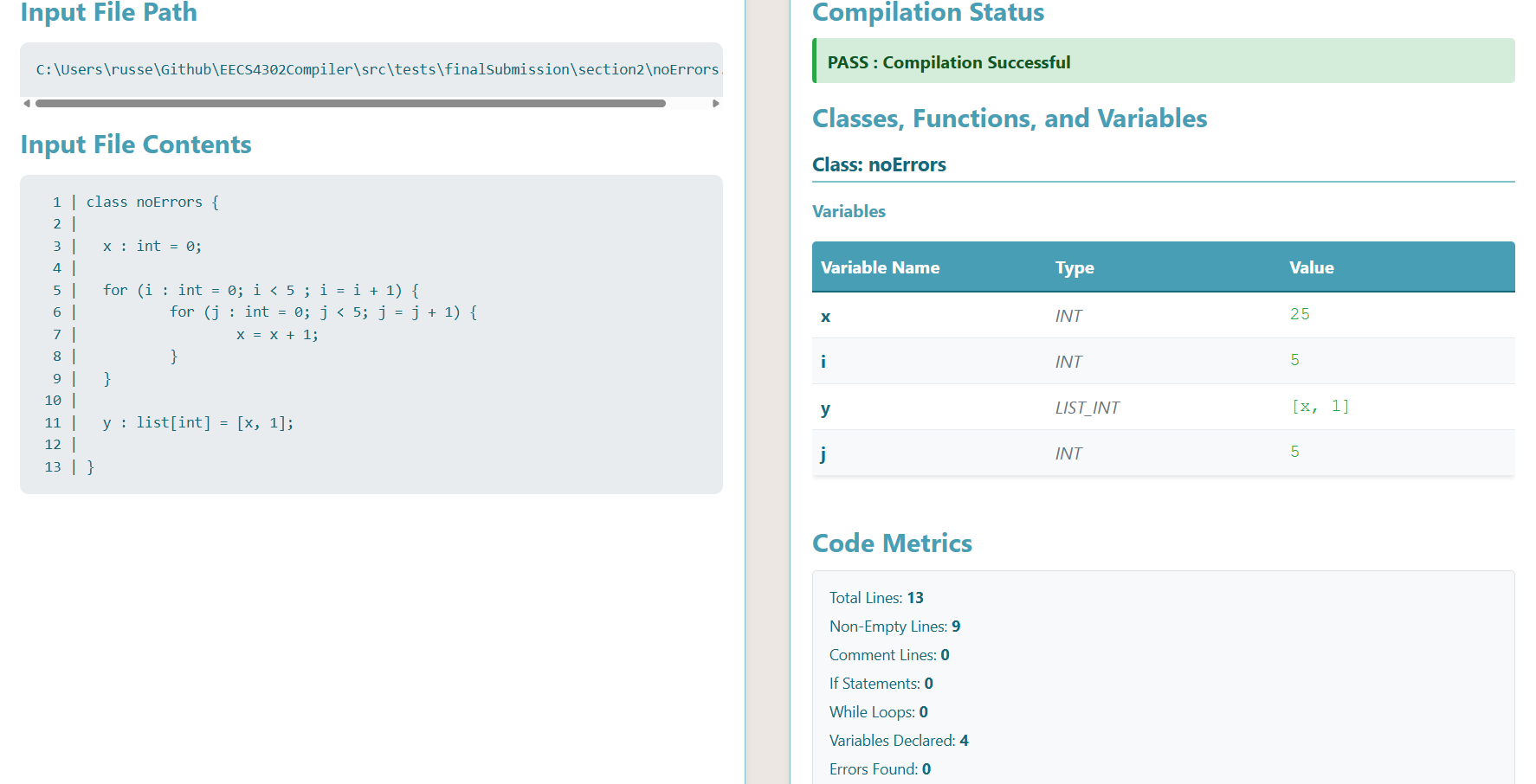
| a: list[int] = [1, 2, 4];  b: list[char] = "this is a string";  c: list[char] = ['s', 't', 'r', 'i', 'n', 'g']; |
| --- |

Supported list types include list[int], list[char], and list[bool]. Also, it is worth pointing out that a character list can be declared in two different ways.

# Output Structure

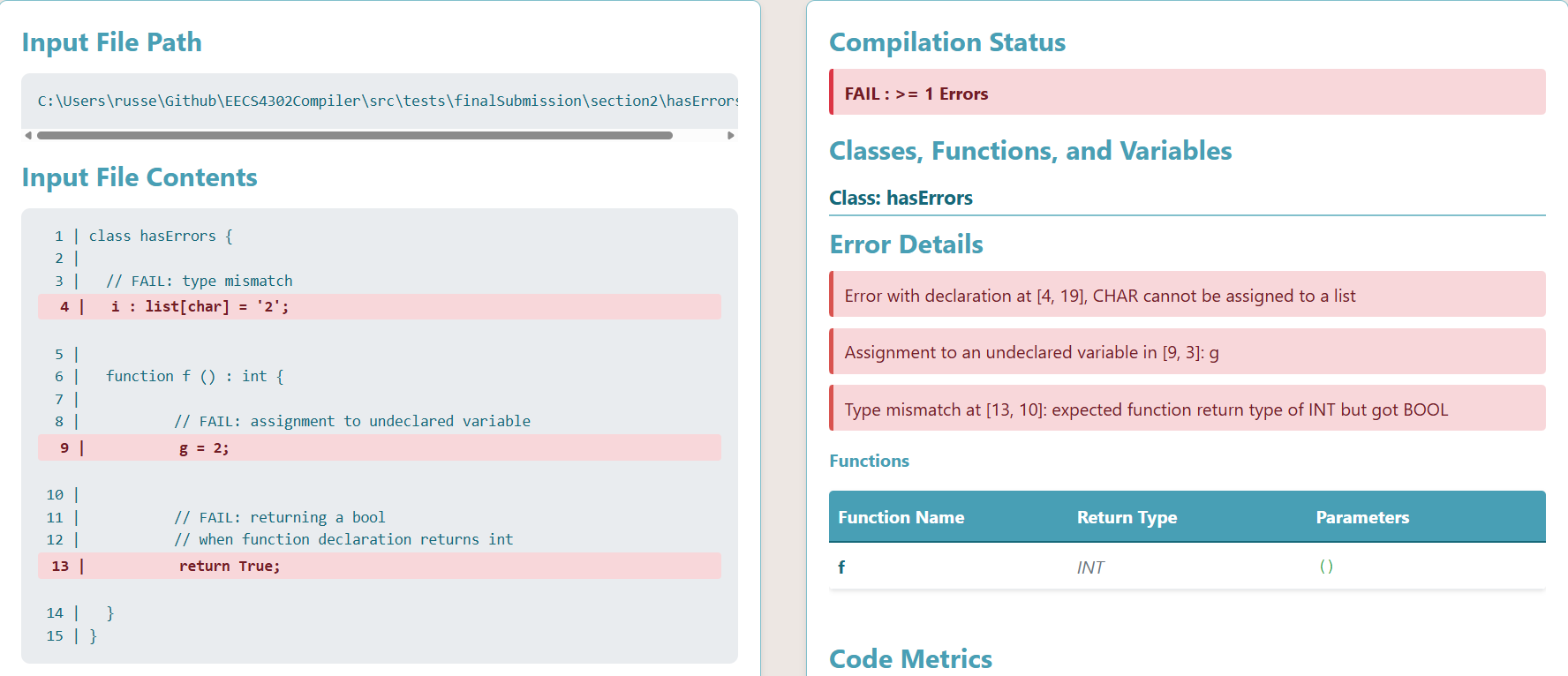
The HTML display differs depending on whether or not the program has semantic errors.

## Program has no semantic errors



If the program is free of semantic errors, the compilation status will display **"PASS"**. The output will also list the final evaluated values of all variables and functions.

## Program has semantic errors

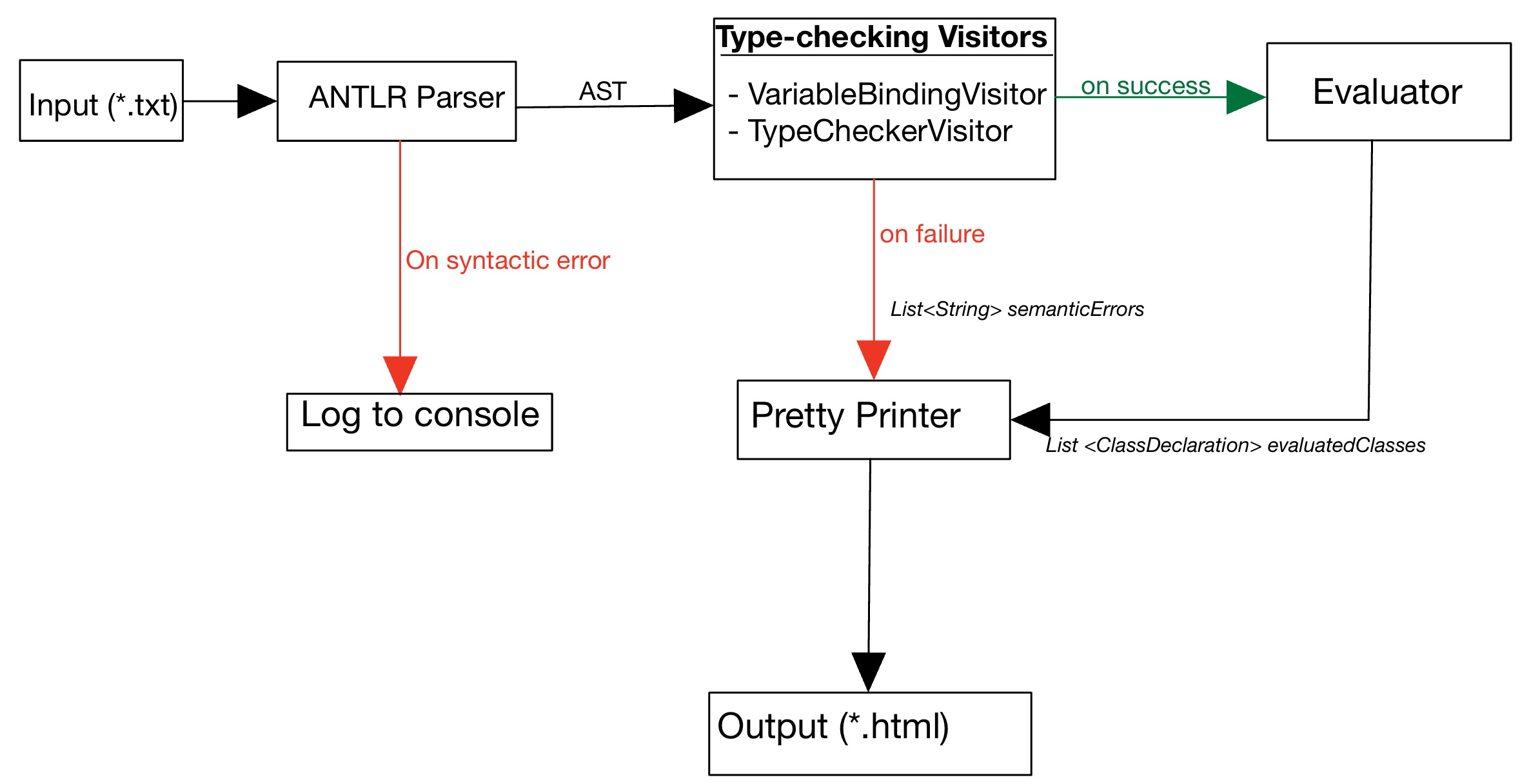


The compiler checks for semantic errors such as:

* Type mismatches
* References to undeclared variables
* Variable redeclarations
* Inconsistent list types
* Other issues (as covered in Section 4)

Any detected errors will be displayed on the right-hand side of the UI. Corresponding lines in the input will be highlighted on the left for easy reference.

# Compiler Workflow and Architecture

The compiler processes .txt files containing class-based programs and produces .html outputs. It begins with the ANTLR parser, which constructs an Abstract Syntax Tree (AST). If syntax errors are detected, the compiler logs them to the console and halts further processing.

If parsing succeeds, the AST (with the program node as root) is traversed by *two* semantic analysis visitors in sequence:

### VariableBindingVisitor

Records declared variables in memory and ensures no undeclared variables are used.

### TypeCheckerVisitor

Validates type correctness for each node, such as preventing boolean values from being assigned to integer variables.

If any semantic errors are found, evaluation is skipped, and the errors are passed to the Pretty Printer to generate an HTML summary.

If no errors are found, the AST is passed to the Evaluator, which assumes semantic validity and sends each class’s variable summary to the Pretty Printer for final HTML generation.

# Justification of Output

The output highlights several type checking criteria applied during compilation. In the code samples below, semantic errors are marked in red, and inline comments explain the reason behind each error.

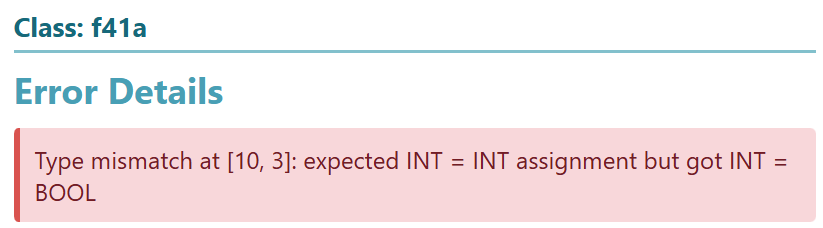
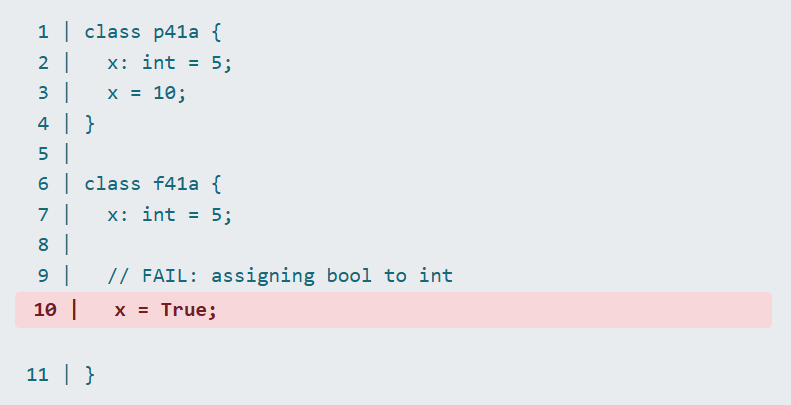
Below each code block, an error log is displayed, providing further detail about the type checking failure, such as:

* mismatched types
* undeclared variables
* incorrect usage of expressions

## Variable declarations and assignments must match their declared type

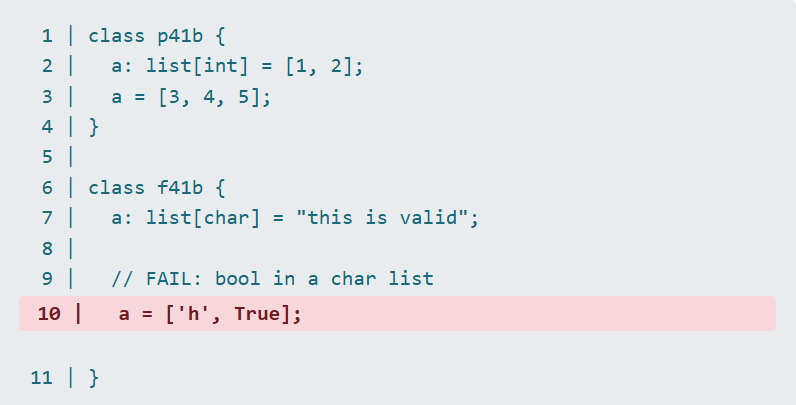
### Primitave Variables

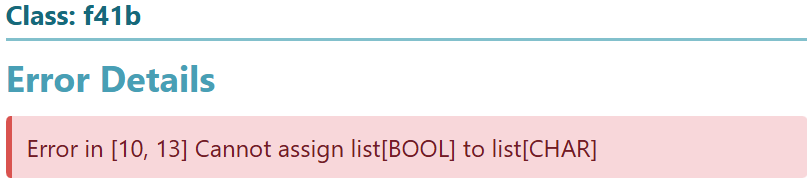
Declarations and assignments must match their declared types. For example, boolean expressions such as == and <= can only be assigned to boolean variables, and arithmetic expressions can only be assigned to integer variables.



### Lists

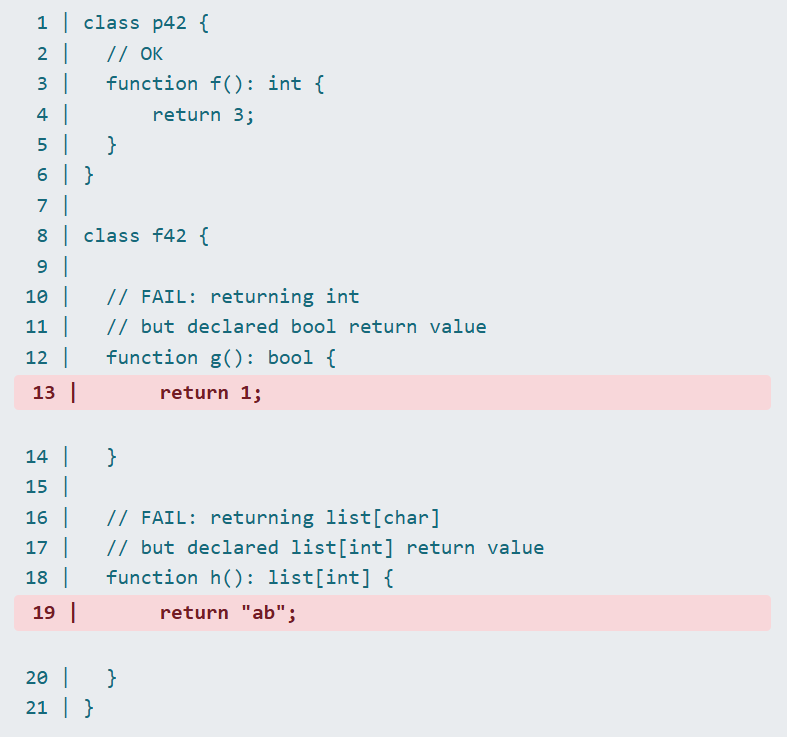
The same concept applies to list declarations and assignment statements. One difference is that for ensuring list type-safety, each item in the list must be checked against the declared type.

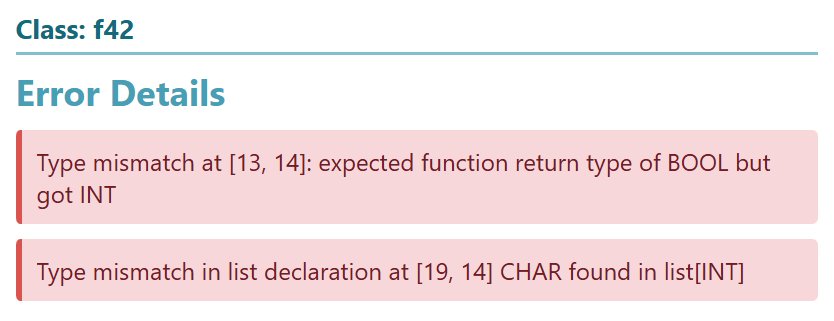




## Function return expressions must match their decleared return type

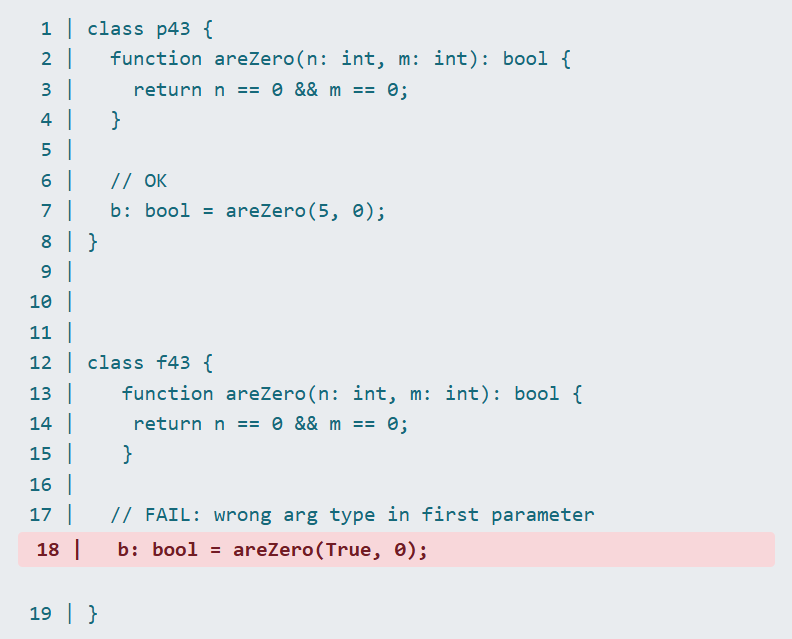
To ensure the type-safety of functions, the expression following their return statement is checked against the type it was declared with.

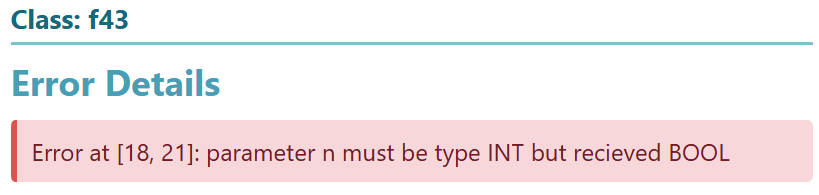




## Function argument types must match parameter types

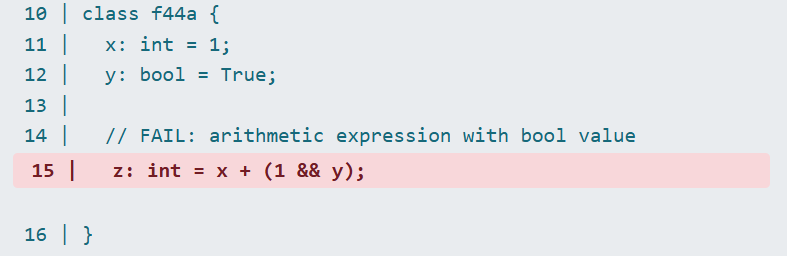
This check ensures that each parameter used in a function invocation matches the type specified in the declaration.

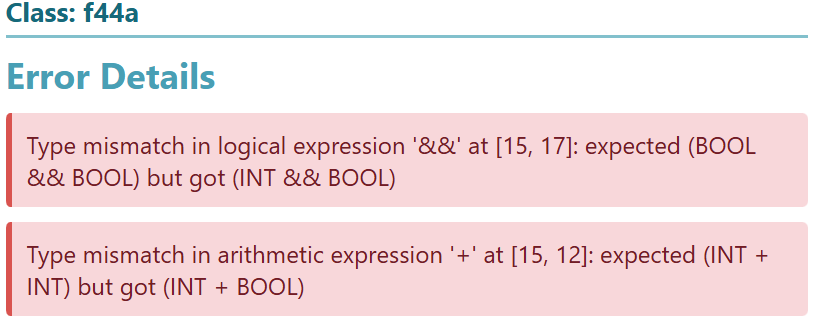




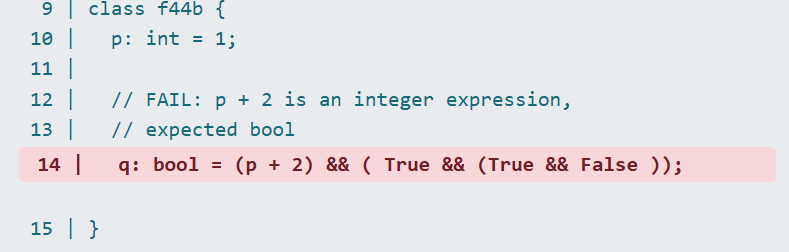
## Binary and unary expressions must use valid operand types

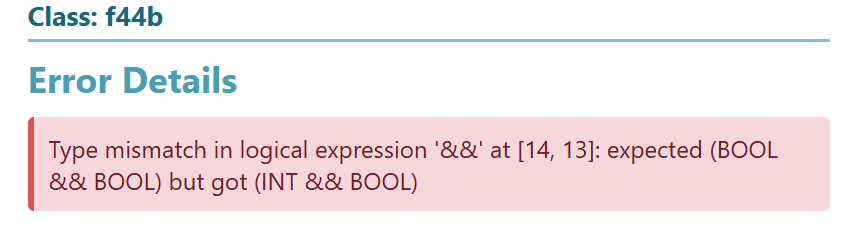
### Arithmetic (+, -, /, \*, %): Both sides must be integers





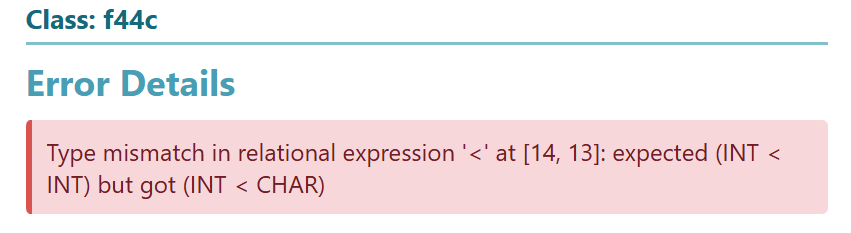
### Logical (&&, ||, !): Operands must be boolean



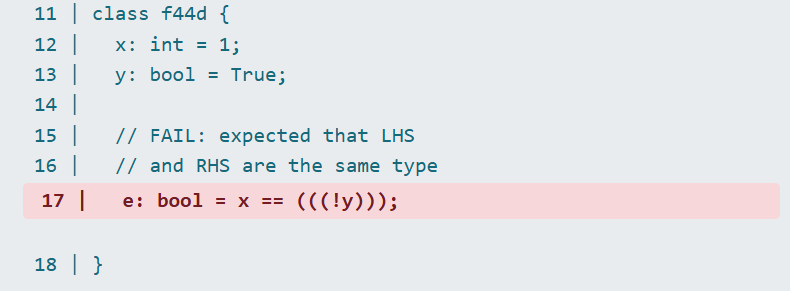


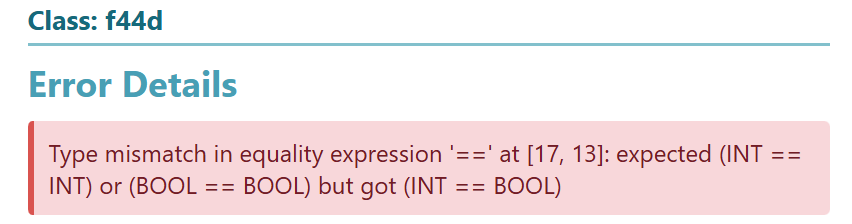
### Relational (<=, <, >, >=): Both sides must be integers.

### 

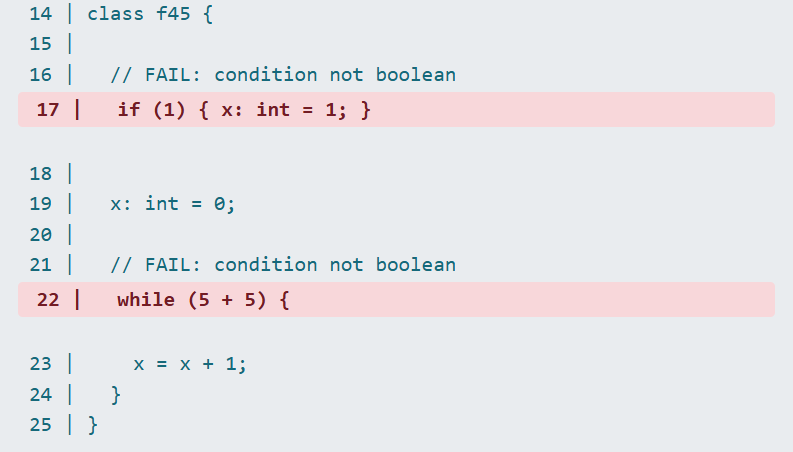


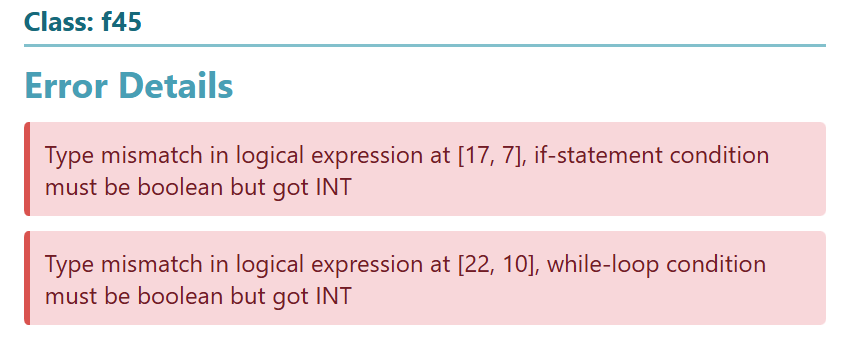
### Equality (==, !=) – Both sides must be of the same type



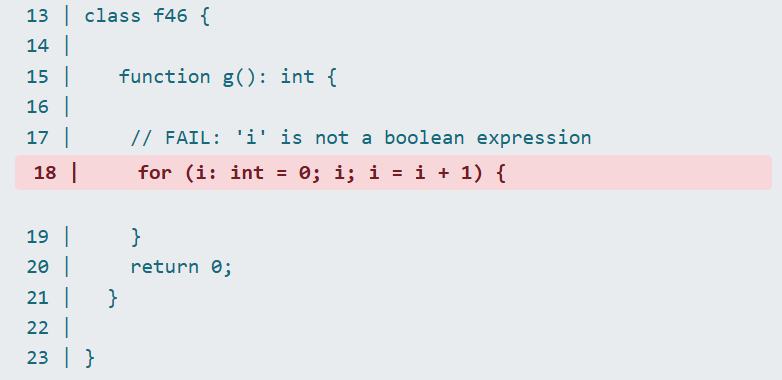


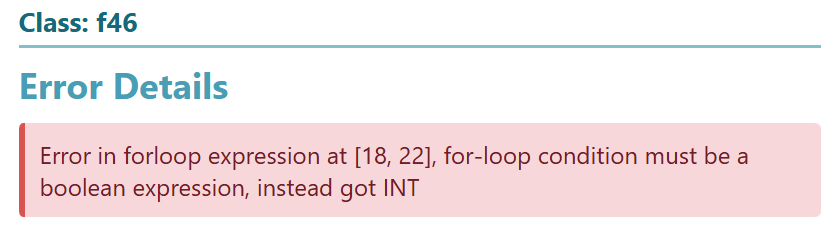
## If-statement and while-loop conditions must be boolean



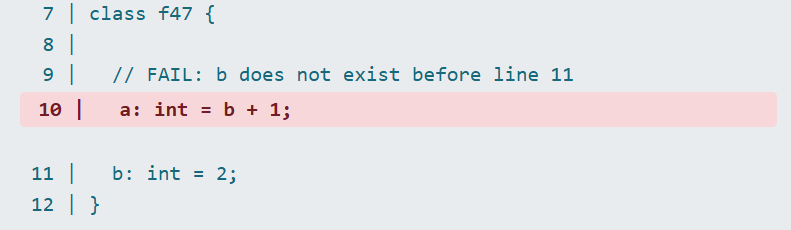


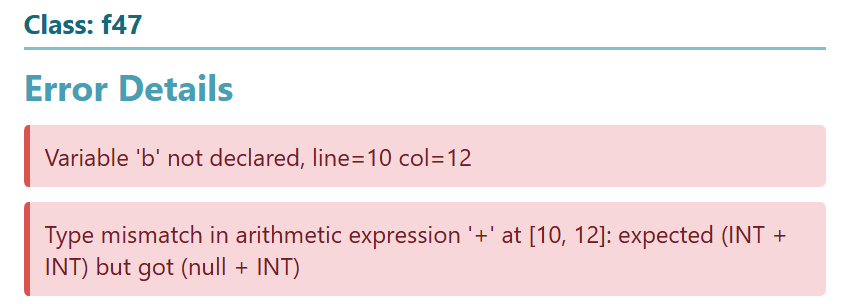
## For-loop expressions must be of specific types



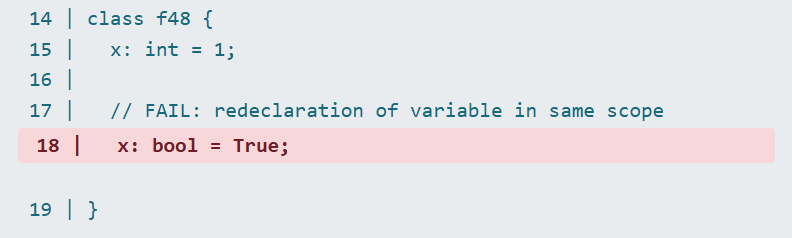


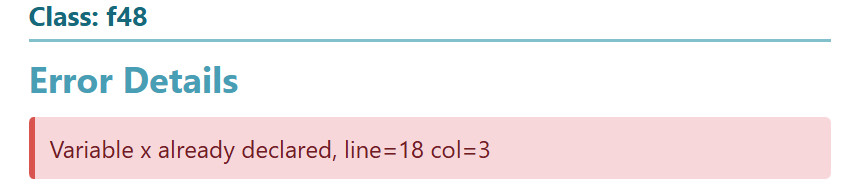
## Variables must be declared before use





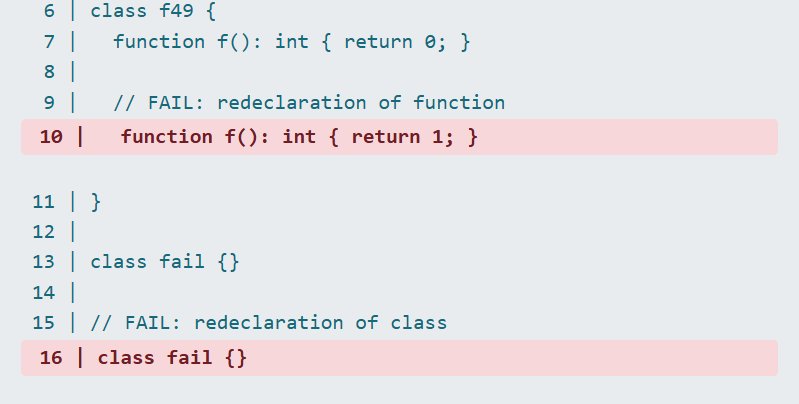
## Variables cannot be declared more than once in the same scope

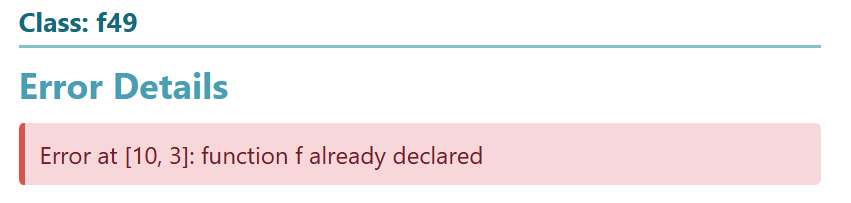




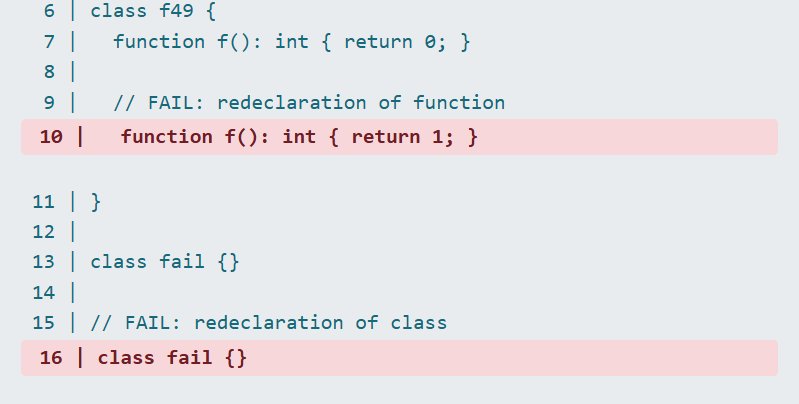
## Functions and Classes cannot be redefined

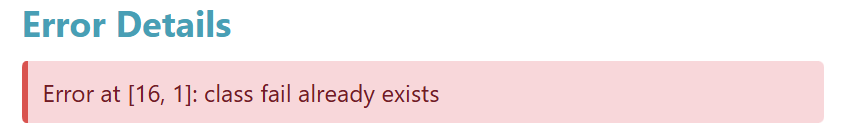
### Function redeclaration error





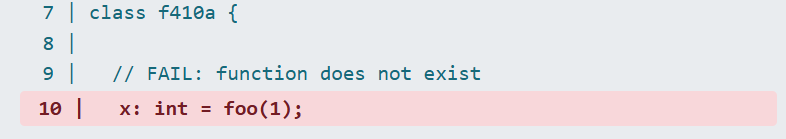
### Class redeclaration error

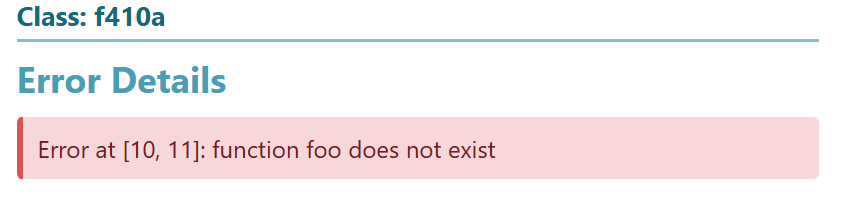




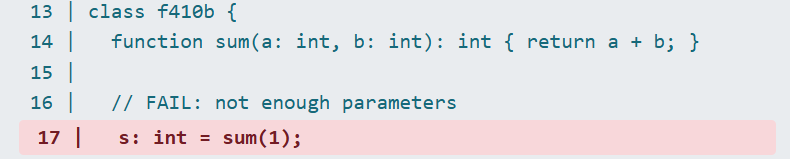
## Function calls must match their declarations

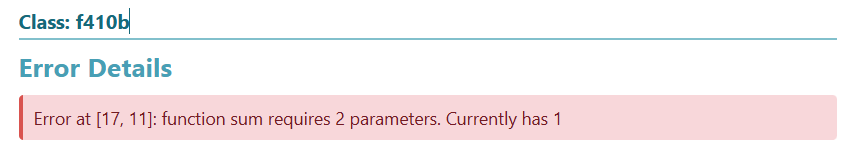
### Functions must be declared to be invoked



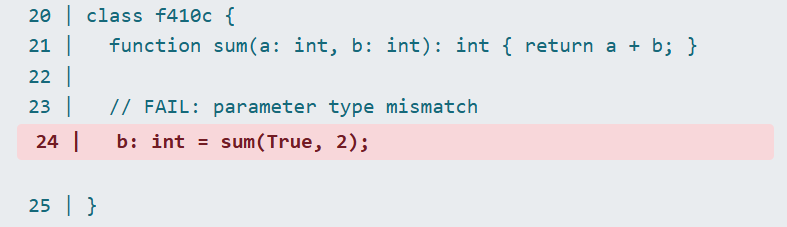


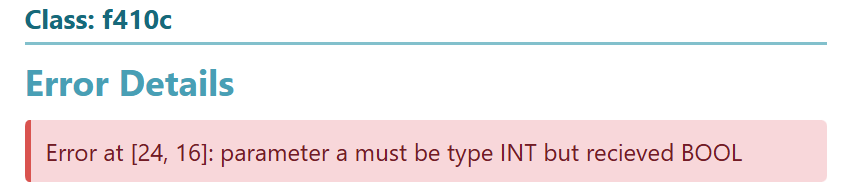
### Function calls must include all parameters from declaration





### Function parameters must match declration parameter types





# Summary of Submitted Examples

The examples are contained in **src/tests/finalSubmission/section5**.

Many of the input files cover multiple items. In general all files include:

* Variable Declaration
* Variable Assignment
* Class Declaration(s)
* Type-checking
* Expression evaluation

## Highlights of Example Input 1

* bank\_account.txt
  + Variable Comparison Operators, and logic

## Highlights of Example Input 2

* calculator.txt
  + Decision Nodes: If statements, conditional expressions, and branching

## Highlights of Example Input 3

* employee.txt
  + Boolean logic, truth values like True/False, logical operations

## Highlights of Example Input 4

* fleet.txt
  + Variable scoping, function declaration, if statements, conditional execution, variable modification

## Highlights of Example Input 5

* gaming.txt
  + Scoping of local vs class variable

## Highlights of Example Input 6

* inventory.txt
  + Inheritance, parent-child class relationship

## Highlights of Example Input 7

* library.txt
  + while-loop, boolean loop and exit condition

## Highlights of Example Input 8

* student.txt
  + Variable operations, function call, function returning

## Highlights of Example Input 9

* task\_manager.txt
  + for-loop, incrementing loop condition verification

## Highlights of Example Input 10

* weather.txt
  + Character list, list assignments, list manipulation

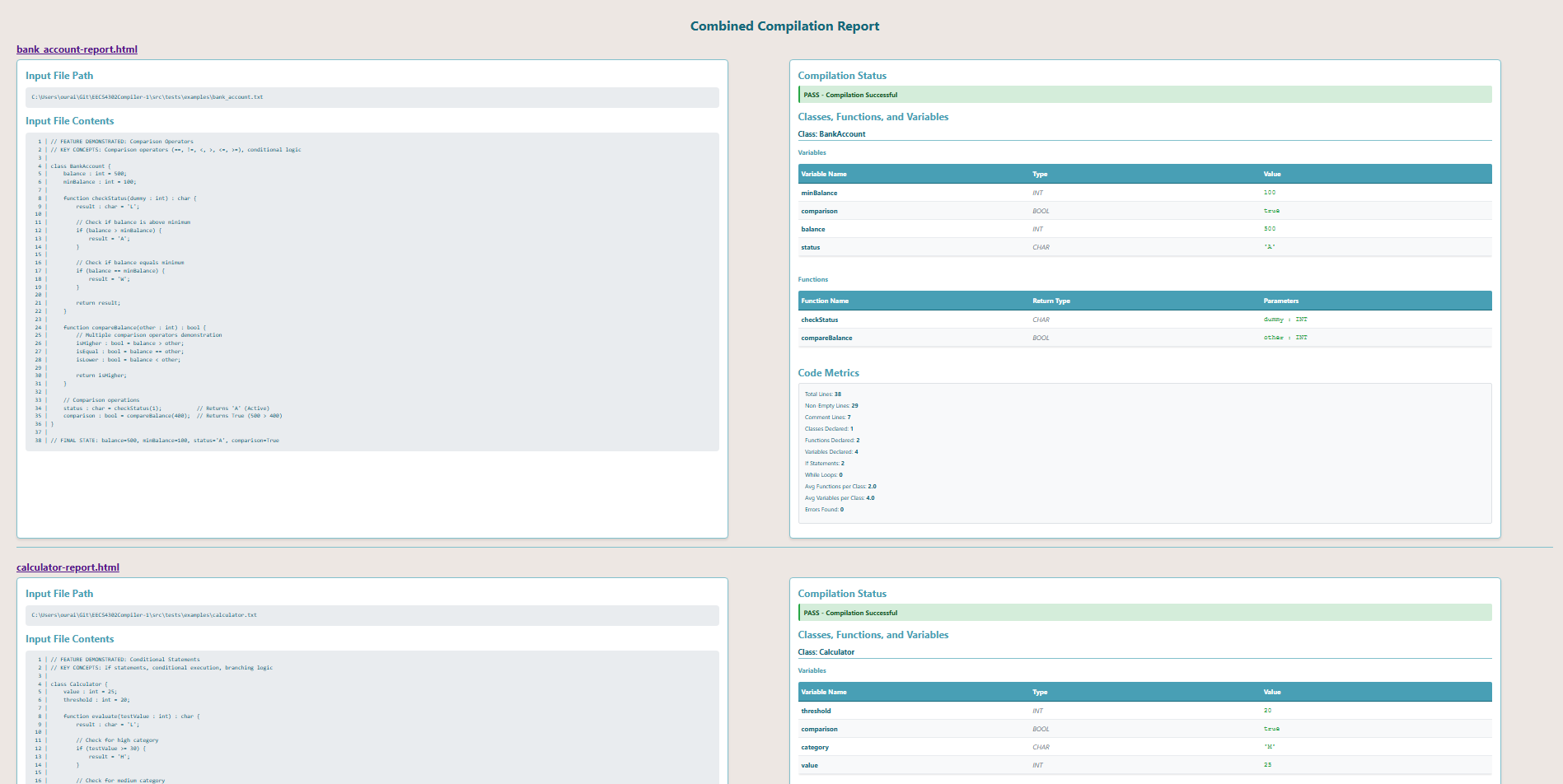
# Miscellaneous Features

## Formatted HTML Report

* Clean, easy to read report with clear results
* Original input file, file path, variables, functions, and classes are listed
* Code Metrics

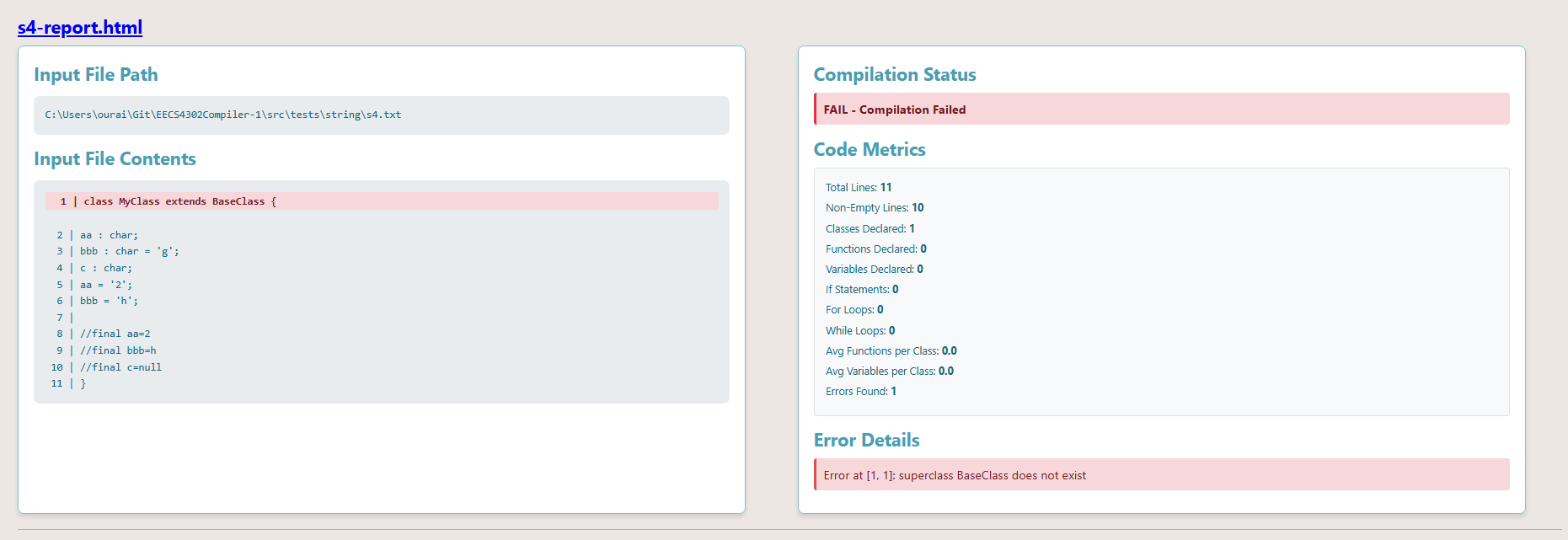


## Multiple Input File Support

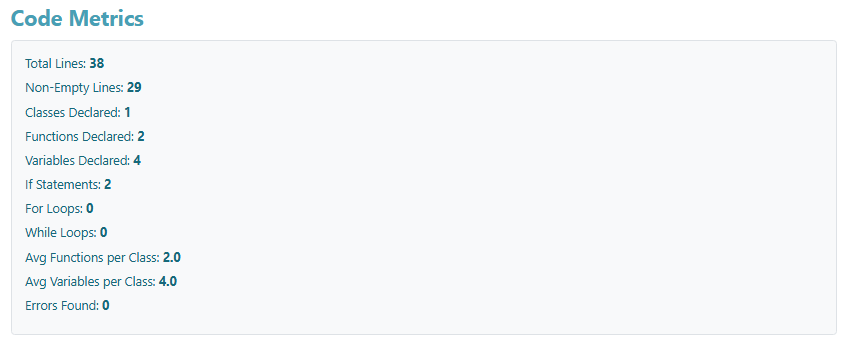
* Able to call a directory containing multiple files, and compile all these files at once
* The files are processed sequentially, and all reports are generated at the end
* A combined report is also generated, which shows all the files on one report, with hyperlinks to each individual report if needed

## Error Highlighting

* Semantic errors are highlighted in red, so they are easier to identify in the input text



## Code Metrics

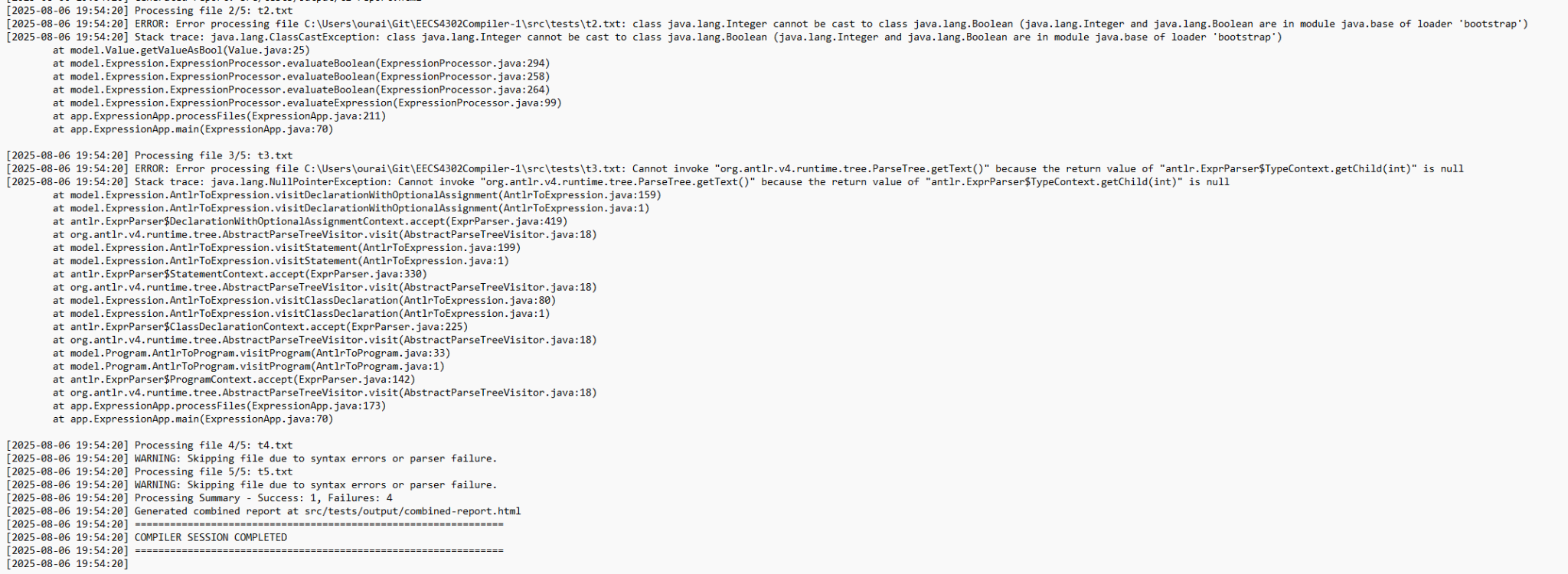
* The following metrics are tracked on a per file basis, this is calculated at report generation time
* Total Lines
* Non-Empty Lines
* Comment Lines
* Classes Declared
* Functions Declared
* Variables Declared
* If Statements
* For Loops
* While Loops
* Avg Functions per Class
* Avg Variables per Class
* Errors Found
* 

## Log Retention

* For more detailed analysis, we have created a history.log file within the tests/output folder
* This is a rolling file, which appends the console outputs and includes details about each file similar to the report, but does not get overridden, and retains a record of old executions.



* This is the way the user can debug syntax errors, as they do not show up in the html report.



# Limitations

### Not Supported:

* **Method overloading, overriding:** For simplicity's sake, every function name must be unique.
* **Nested functions:** Functions cannot be declared inside other functions or within loops.
* **List operations:** Only list declarations and direct assignments are supported; dynamic operations like adding or removing elements are not implemented.
* **Character expressions:** Equality and relational operations involving char types are not supported. These expressions currently only work with int and bool types.
* **No anonymous/lambda functions:** All functions must be named and declared at the class level.
* **No string operations or methods:** Strings are allowed ("...") but the compiler does not currently support operations like concatenation or indexing.
* **No operator precendence enforcement in grammar**

### Just Out of Scope (due to time constraints):

These features were not implemented but could be supported in future iterations:

* Object creation and dot notation (e.g., new ClassName(), object.field)
* Private variables and access modifiers
* else if / else branches
* do-while loops
* print statements