Project 4: Static Semantic Analysis in Compiler Design

1. Introduction to the Project

Project 4 focused on implementing a static semantic analyzer as part of a multi-phase compiler project for a custom programming language. The core objective was to extend the compiler to enforce type and scope rules at compile time, ensuring that programs not only followed correct syntax but also adhered to the semantic rules of the language. This phase built on earlier projects that established parsing and interpretation capabilities, requiring a deeper integration of symbol tables, type checking, and error reporting. The project served as both a practical and conceptual exploration of how real-world compilers safeguard program correctness before execution.

2. Approach Taken

My approach was structured around **incremental development and testing**, leveraging both prior work and the newly provided skeleton for static semantic analysis. At the outset, I thoroughly reviewed the project requirements, phase breakdown, and provided resources, and made a step-by-step to-do list to guide progress.

Key steps in my methodology included:

- **Baseline Integration:**
  I started with a working baseline from the previous project (Project 2), ensuring that I could merge new requirements without breaking existing functionality. This allowed for safe, incremental changes.

- **Incremental Semantic Checks:**
  Rather than implementing all checks at once, I integrated type and scope checks phase by phase. Each semantic rule—such as type matching in assignments, duplicate declarations, and operator type restrictions—was first added, tested, and validated using provided and custom test cases.

- **Use of Helper Functions:**

To promote code reuse and clarity, I encapsulated complex checks (such as those for control-flow branches and operator requirements) in dedicated helper functions. This modular approach reduced duplication and simplified debugging.

- **Symbol Table Management:**
  I carefully transitioned from runtime value tracking to compile-time type tracking by maintaining separate symbol tables for type and value information. This separation helped clarify the compiler's responsibilities in each phase.

- **Test-Driven Debugging:**
  After each change, I ran the corresponding semantic test files, closely analyzing compiler output to identify both successful detections and any regressions. This immediate feedback loop was crucial in catching subtle issues, such as improper handling of list types or failure to recognize valid identifiers.

Throughout, I worked independently, but made a point to document significant changes and decision points to aid in both troubleshooting and potential future collaboration.

3. Lessons Learned and Suggested Improvements

Lessons Learned:

- **Importance of Incremental Development:**
  Attempting to implement multiple semantic rules at once led to hard-to-debug errors and build issues. Working incrementally—with compilation and testing after each step—helped isolate problems and maintain project momentum.

- **Role of the Lexer:**
  I discovered the impact of lexical analysis on the entire project: a scanner rule that failed to recognize underscores in identifiers caused a cascade of lexical and syntax errors, reinforcing that even small oversights in early compiler phases have major downstream effects.

- **Effective Error Handling:**
  Implementing meaningful error messages and careful handling of

special enum values (like `MISMATCH` and `NONE`) proved critical for both user experience and debugging. Cascading errors were reduced by guarding against redundant or misleading messages.

- **Value of Code Modularity:**
  Encapsulating semantic checks in helper functions not only made my grammar actions more readable but also made it easier to update logic in response to changing requirements or discovered bugs.

Suggested Improvements:

- **Early and Rigorous Testing of Lexer Patterns:**
  More extensive identifier testing early in the project would have prevented several avoidable errors and saved debugging time.

- **Use of Automated Build/Test Scripts:**
  Integrating automated scripts for compiling and running semantic test suites would have streamlined regression testing and improved confidence in each code change.

- **Clearer Documentation of Enum Semantics:**
  While working with custom types and error markers (e.g., `MISMATCH`), I realized that in-code comments and consistent enum usage guidelines would help prevent semantic confusion, especially as the codebase grows.

- **Scoping and Namespace Management:**
  As symbol tables became more complex, introducing better scope management (possibly using stack-based symbol tables) would set the stage for future phases, such as handling local variables or function parameters more elegantly.

Conclusion

Project 4 provided deep insight into the practical challenges of enforcing static semantic rules in compiler design. By applying incremental development, focusing on test-driven progress, and continually refining my approach in response to observed issues, I not only met the project objectives but also built a stronger foundation for future enhancements. The

process highlighted the importance of robust lexical analysis, modular code, and proactive error management in building reliable software tools.

Test Plan

Figure 1. Semantic1.txt

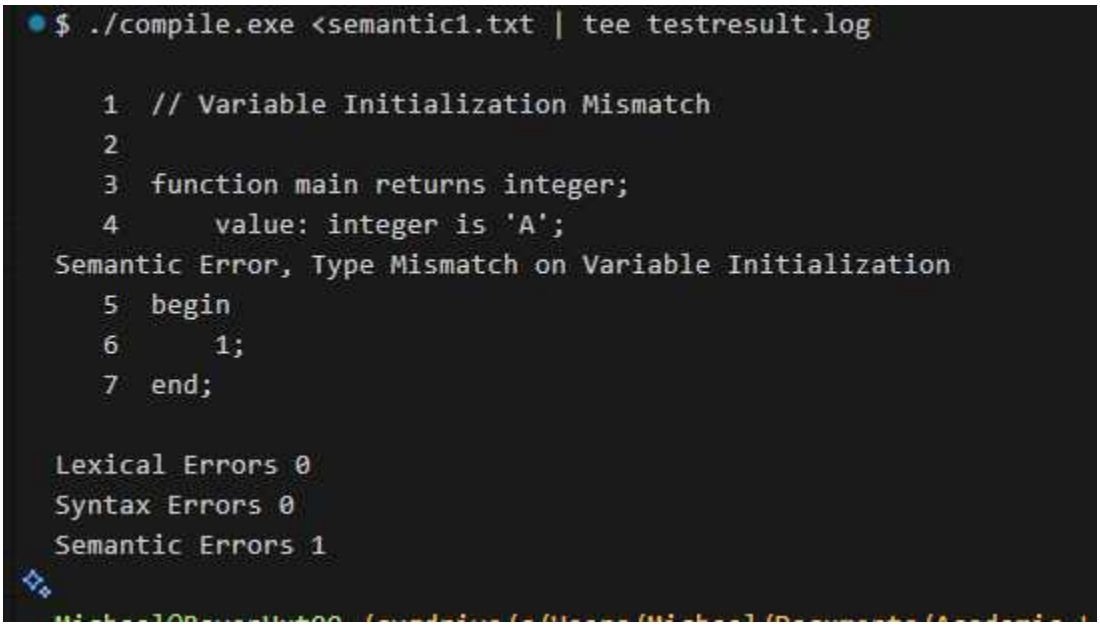

Figure 2. Semantic2.txt

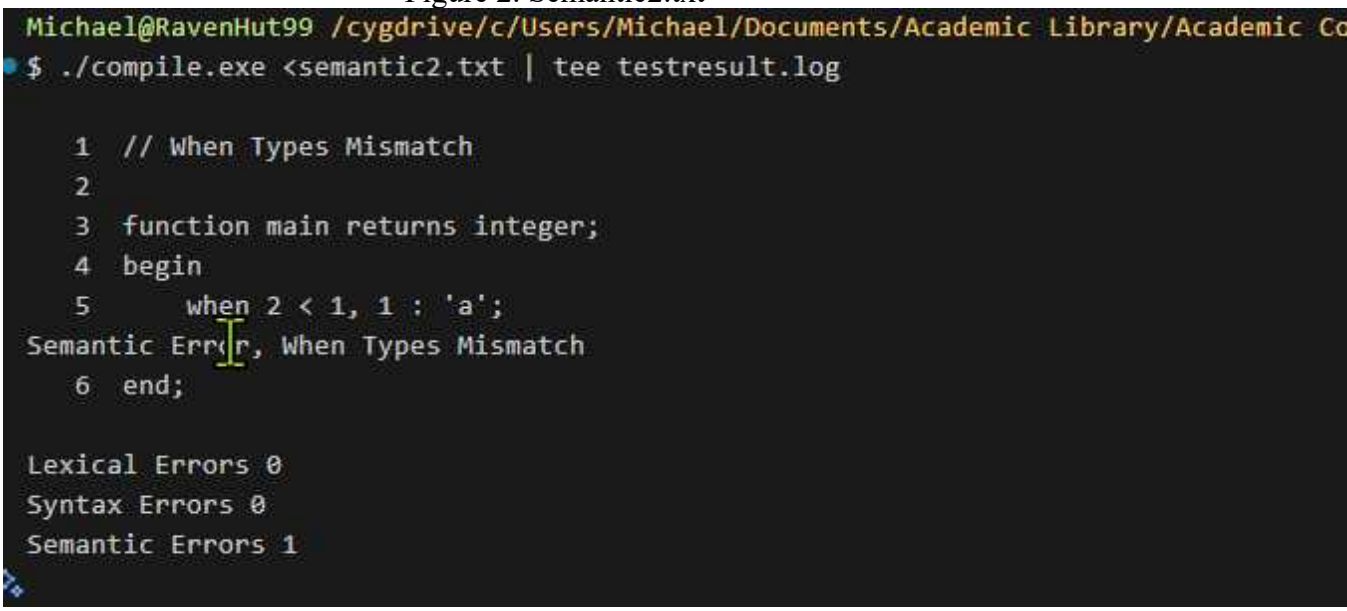

Figure 3. Semantic3.txt

```
Michael@RavenHut99 /cygdrive/c/Users/Michael/Documents/Acade
$ ./compile.exe <semantic3.txt | tee testresult.log

   1  // Non Integer Switch Expression
   2
   3  function main returns integer;
   4      b: character is 'A';
   5  begin
   6      switch b is
   7          case 1 => 2;
   8          case 2 => 4;
   9          others => 6;
  10      endswitch;
Semantic Error, Switch Expression Not Integer
  11  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 4. Semantic4.txt

```
Michael@RavenHut99 /cygdrive/c/Users/Michael/Documents/Academic Libra
$ ./compile.exe <semantic4.txt | tee testresult.log

   1  // Case Types Mismatch
   2
   3  function main returns integer;
   4      b: character is 'b';
   5  begin
   6      switch 1 is
   7          case 1 => 2;
   8          case 2 => b;
Semantic Error, Case Types Mismatch
   9          others => 6;
  10      endswitch;
  11  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 5. Semantic5.txt

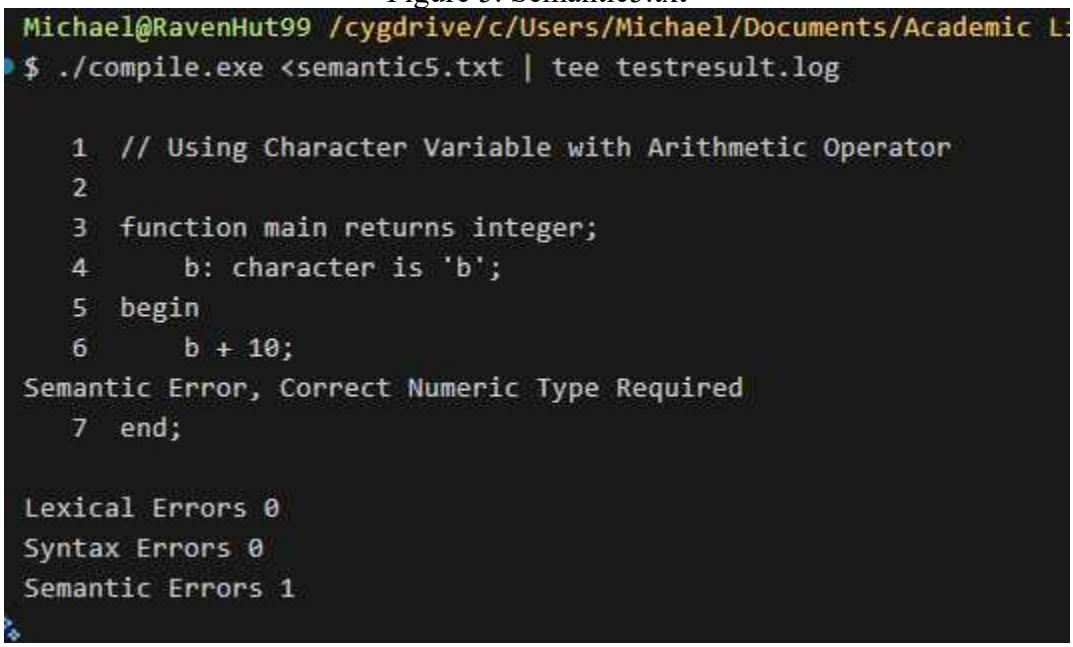

Figure 6. Semantic6.txt

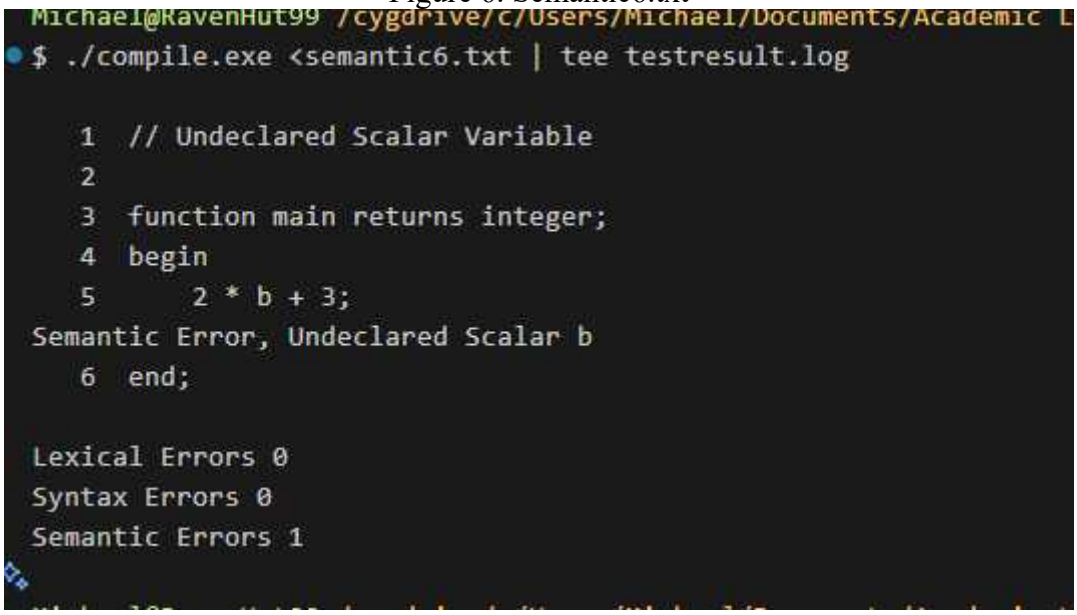

Figure 7. Semantic7.txt

```
$ ./compile.exe <semantic7.txt | tee testresult.log

    1  // Undeclared List Variable
    2
    3  function main returns integer;
    4  begin
    5      primes(1) + 1;
Semantic Error, Undeclared List primes
    6  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 8. Semantic8.txt

```
$ ./compile.exe <semantic8.txt | tee testresult.log

    1  // List with Elements of Different Types
    2
    3  function main returns integer;
    4      aList: list of integer is (1, 2, 3.5);
Semantic Error, Element Types Mismatch
    5  begin
    6      aList(1);
    7  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 9. Semantic9.txt

```
$ ./compile.exe <semantic9.txt | tee testresult.log

   1  // List Type Does Not Match Element Types
   2
   3  function main returns character;
   4      aList: list of character is (1, 2, 3);
Semantic Error, Element Types Mismatch
   5  begin
   6      aList(1);
   7  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 10. Semantic10.txt

```
$ ./compile.exe <semantic10.txt | tee testresult.log

   1  // List Subscript is not Integer
   2
   3  function main returns integer;
   4      aList: list of integer is (1, 2, 3);
   5  begin
   6      aList(1.5);
Semantic Error, Index Types Mismatch
   7  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 11. Semantic11.txt

```
$ ./compile.exe <semantic11.txt | tee testresult.log

     1  -- Mixing Numeric and Character Types with Relational Operator
     2
     3  function main returns integer;
     4  begin
     5      if 'b' < 'c' then
     6          1;
     7      elsif 'b' < 1 then
     8          2;
     9      else
  Semantic Error, If-Else Type Mismatch
    10          3;
    11      endif;
    12  end;

  Lexical Errors 0
  Syntax Errors 0
  Semantic Errors 1
```

Figure 12. Semantic12.txt

```
$ ./compile.exe <semantic12.txt | tee testresult.log

   1  // Using Character Literal with Exponentiation Operator
   2  //     and Negation Operator
   3
   4  function main returns integer;
   5      c: character is ~'c';
Semantic Error, Negation Operator Requires Numeric Types
   6  begin
   7      5 ^ 'P';
Semantic Error, Exponentiation Operator Requires Numeric Types
   8  end;
   9

Lexical Errors 0
Syntax Errors 0
Semantic Errors 2
```

Figure 13. Semantic13.txt

```
$ ./compile.exe <semantic13.txt | tee testresult.log

   1  // Mixing Real Literals with the Remainder Operator
   2
   3  function main returns integer;
   4  begin
   5      4 % 4.8;
Semantic Error, Remainder Operator Requires Integer Operands
   6  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 14. Semantic14.txt

```
./compile.exe <semantic14.txt | tee testresult.log

 1  -- If Elsif Else Mismatch
 2
 3  function main returns integer;
 4  begin
 5      if 9 < 10 then
 6          1;
 7      elsif 8 = 1 then
 8          2;
 9      else
emantic Error, If-Else Type Mismatch
10          3.7;
11      endif;
12  end;

exical Errors 0
yntax Errors 0
emantic Errors 1
```

Figure 15 Semantic15.txt

```
$ ./compile.exe <semantic16.txt | tee testresult.log

   1  -- Narrowing Variable Initialization
   2
   3  function main returns integer;
   4      b: integer is 5 * 2.5;
Semantic Error, Type Mismatch on Variable Initialization
   5  begin
   6      b + 1;
   7  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 16. Semantic16.txt

```
$ ./compile.exe <semantic16.txt | tee testresult.log

   1  -- Narrowing Variable Initialization
   2
   3  function main returns integer;
   4      b: integer is 5 * 2.5;
Semantic Error, Type Mismatch on Variable Initialization
   5  begin
   6      b + 1;
   7  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 17. Semantic17.txt

```
$ ./compile.exe <semantic17.txt | tee testresult.log

   1  -- Narrowing Function Return
   2
   3  function main returns integer;
   4      b: integer is 6 * 2;
   5  begin
   6      if 8 < 0 then
   7          b + 3.0;
   8      else
   9          b * 4.6;
  10      endif;
  11  end;
Semantic Error, Type Mismatch on Variable Initialization

Lexical Errors 0
Syntax Errors 0
Semantic Errors 1
```

Figure 18. Semantic18.txt

```
$ ./compile.exe <semantic18.txt | tee testresult.log

   1  -- Duplicate Scalar and List Variables
   2
   3  function main returns integer;
   4      scalar: integer is 4 * 2;
   5      scalar: character is 'b';
Semantic Error, Duplicate Scalar scalar
   6      a_list: list of integer is (4, 2);
   7      a_list: list of real is (2.3, 4.4);
Semantic Error, Duplicate List a_list
   8  begin
   9      1;
  10  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 2
```

Figure 19. Semantic19.txt

```
    1  // Multiple Semantic Errors
    2
    3  function main returns integer;
    4      value: integer is 4.5;
Semantic Error, Type Mismatch on Variable Initializati
    5      numbers: list of real is (1, 2, 3);
Semantic Error, Element Types Mismatch
    6      one: integer is '1';
Semantic Error, Type Mismatch on Variable Initializati
    7  begin
    8      if value > 0 then
    9          fold left + ('a', 'b') endfold;
Semantic Error, Fold Requires A Numeric List
   10      elsif name = 'N' then
Semantic Error, Undeclared Scalar name
   11          fold right * (1, 2.5) endfold;
Semantic Error, Element Types Mismatch
Semantic Error, Fold Requires A Numeric List
   12      else
Semantic Error, If-Else Type Mismatch
   13          when value < 10, 1 : 1.5;
Semantic Error, When Types Mismatch
   14      endif;
   15  end;

Lexical Errors 0
Syntax Errors 0
Semantic Errors 9
```

Figure 20. Valid1

```
$ ./compile.exe <valid1.txt | tee testresult.log

    1  -- Program with a Real Variable
    2
    3  function main returns real;
    4      a: real is 4.5;
    5  begin
    6      a;
    7  end;

Compiled Successfully
```

Figure 21. Valid2

```
$ ./compile.exe <valid2.txt | tee testresult.log

    1  -- Program with a Hexadecimal Literals
    2
    3  function main returns integer;
    4      a: integer is #A;
    5  begin
    6      a + #a;
    7  end;

Compiled Successfully
```

Figure 22. Valid3

```
$ ./compile.exe <valid3.txt | tee testresult.log

    1  -- Program with a Real Variable
    2
    3  function main returns real;
    4      a: real is 4 + 4.5;
    5  begin
    6      a;
    7  end;

Compiled Successfully
```