Project 3 Report

Section 1 – Approach

Overview

Project 3 challenged me to integrate and extend the lessons of Project 2, blending scanning, parsing, and semantic actions into a cohesive compiler front-end. My initial assumption—that Project 3 would simply layer a few extra features atop my existing Project 2 code—proved naive. In fact, I underestimated how much of Project 2's structure would need to be revisited and revised.

Approach 1 – Direct Import from Project 2

- I began by copying the grammar rules, semantic actions, and header inclusions from Project 2, following the example layouts from the Project 3 approach PDF.
- This led to immediate and widespread build failures, including type mismatches, unresolved symbols, and token conflicts—especially around %union types and error recovery.
- I realized many of the illustrative rules in the PDF were not meant to be imported wholesale, but were pedagogical starting points meant to inspire a tailored implementation.
- My experience with Project 2—where incremental tweaks often worked—did not translate. Here, direct copy-paste left the parser in an unrecoverable state. Debugging was time-consuming and often inconclusive, as errors would cascade across the grammar and symbol table code.

Approach 2 – Incremental, Ground-Up Implementation

- Recognizing the limits of the import approach, I reread both Project 2 and 3 specifications and started with a minimalist, compiling parser.
- I brought over only the essential, working features from Project 2, carefully adapting them for Project 3's requirements—especially where Project 2's code had hard-coded assumptions (such as parameter lists or variable scoping).
- At each step, I re-implemented scanner tokens, grammar fragments, and semantic actions, using Project 2's working state as a checkpoint but not a template.

- This time, I implemented one feature at a time: first getting single-statement parsing correct, then expanding to block statements, and then integrating error reporting and symbol table management.
- Lessons from Project 2 were invaluable—especially around left recursion, operator precedence, and handling the interaction of %union types across scanner and parser modules.
- As with Project 2, I maintained a read-only "golden" copy at each stable milestone for fast rollback, which proved even more valuable given the higher complexity and interconnectedness of Project 3.

Key Differences from Project 2

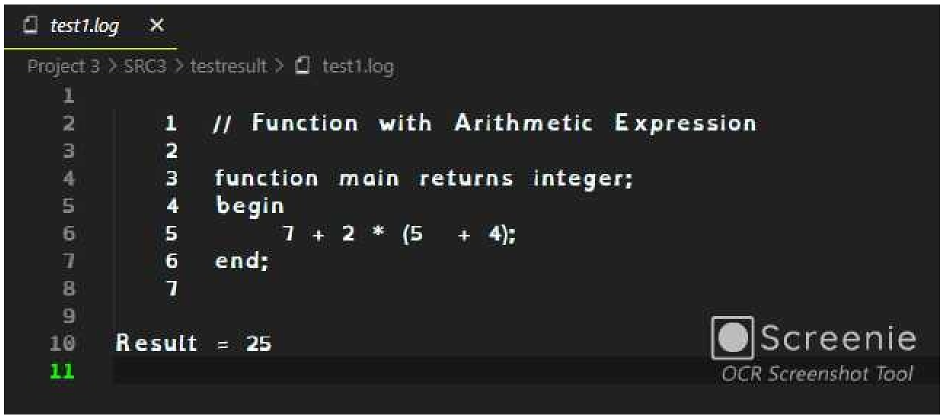
- Project 2 focused mostly on parsing and basic semantic checks, with relatively flat grammar and limited scope for variables and control flow.
- Project 3 demanded robust error recovery, comprehensive symbol table integration, and support for more complex constructs like folds, lists, and type-checked expressions.
- I learned that simply extending Project 2 was not enough; I needed to refactor, modularize, and sometimes reimagine foundational components (especially scanner and semantic actions) to meet the new requirements.

Chosen Strategy

By blending careful reuse of proven Project 2 logic with a ground-up, incremental design for new features, I ensured that each addition to Project 3 could be tested and validated in isolation. This "hybrid" strategy, combined with more disciplined version control and snapshotting, dramatically reduced debugging time and gave me confidence that each stage worked as intended.

Section 2 – Test Cases

<$n#figure:TEST1>



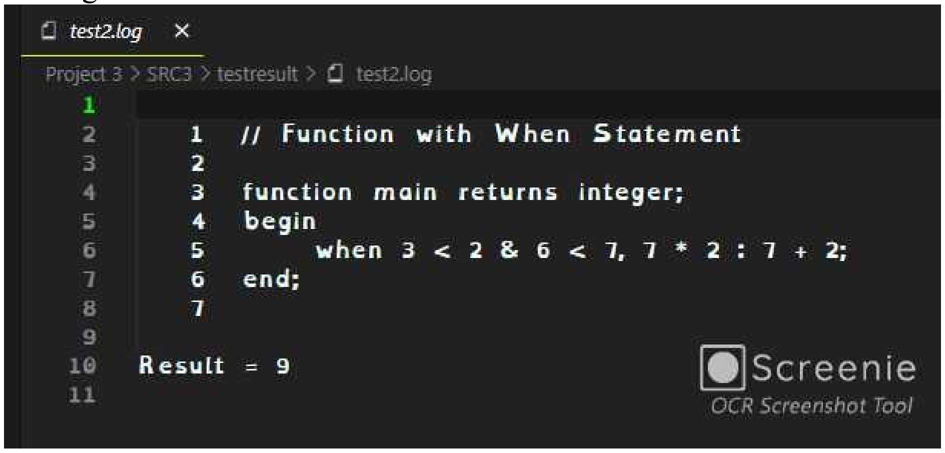
```

   1  // Function with Arithmetic Expression
   2
   3  function main returns integer;
   4  begin
   5       7 + 2 * (5   + 4);
   6  end;
   7

Result = 25

```

<$n#figure:TEST2>



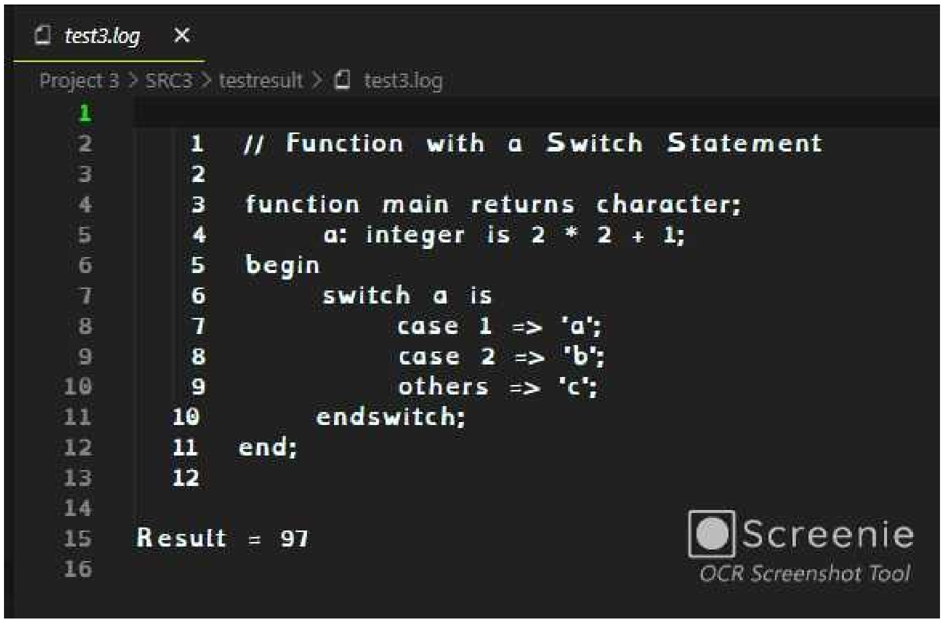
```

   1  // Function with When Statement
   2
   3  function main returns integer;
   4  begin
   5       when 3 < 2 & 6 < 7, 7 * 2 : 7 + 2;
   6  end;
   7

Result = 9

```

<$n#figure:TEST3>

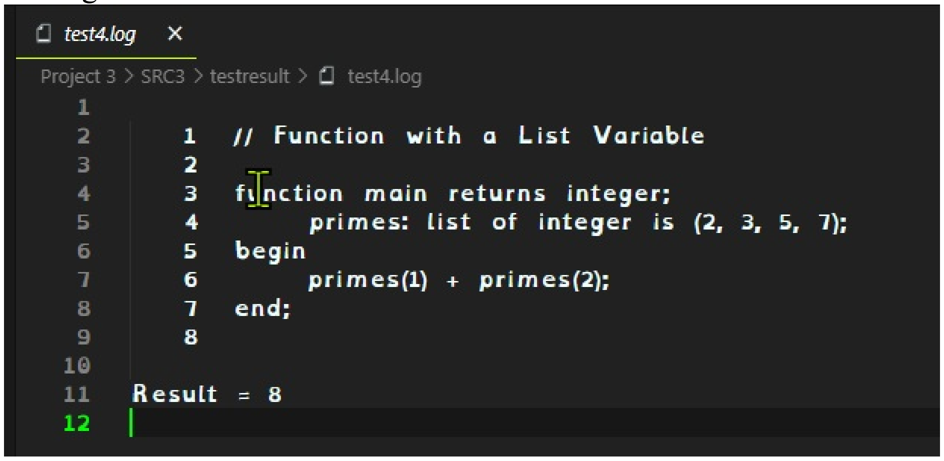


```
test3.log
Project 3 > SRC3 > testresult > test3.log

   1 // Function with a Switch Statement
   2
   3 function main returns character;
   4      a: integer is 2 * 2 + 1;
   5 begin
   6      switch a is
   7           case 1 => 'a';
   8           case 2 => 'b';
   9           others => 'c';
  10      endswitch;
  11 end;
  12

Result = 97
```

<$n#figure:TEST4>

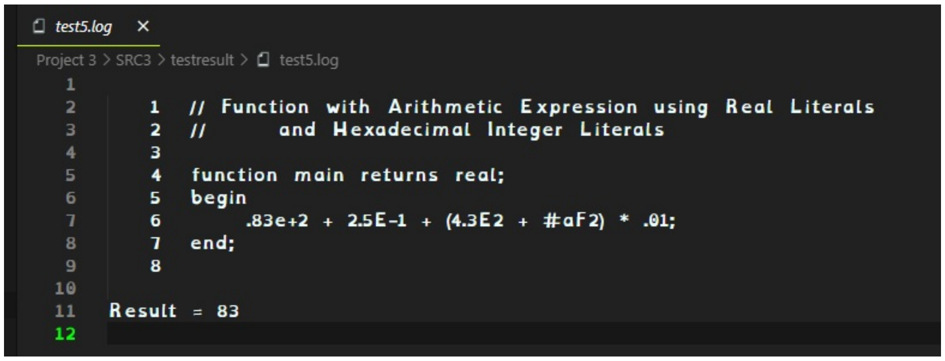


```
test4.log
Project 3 > SRC3 > testresult > test4.log

   1 // Function with a List Variable
   2
   3 function main returns integer;
   4      primes: list of integer is (2, 3, 5, 7);
   5 begin
   6      primes(1) + primes(2);
   7 end;
   8

Result = 8
```

<$n#figure:TEST5>

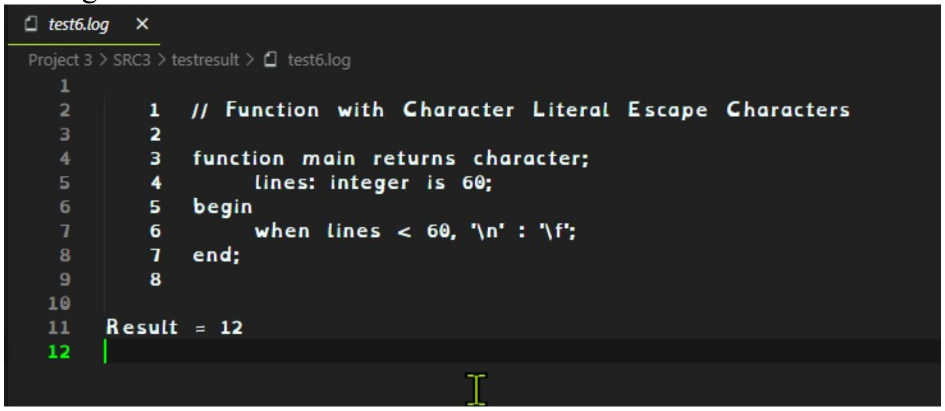


```
test5.log
Project 3 > SRC3 > testresult > test5.log

   1  // Function with Arithmetic Expression using Real Literals
   2  //       and Hexadecimal Integer Literals
   3
   4  function main returns real;
   5  begin
   6       .83e+2 + 2.5E-1 + (4.3E2 + #aF2) * .01;
   7  end;
   8

Result = 83
```

<$n#figure:TEST6>

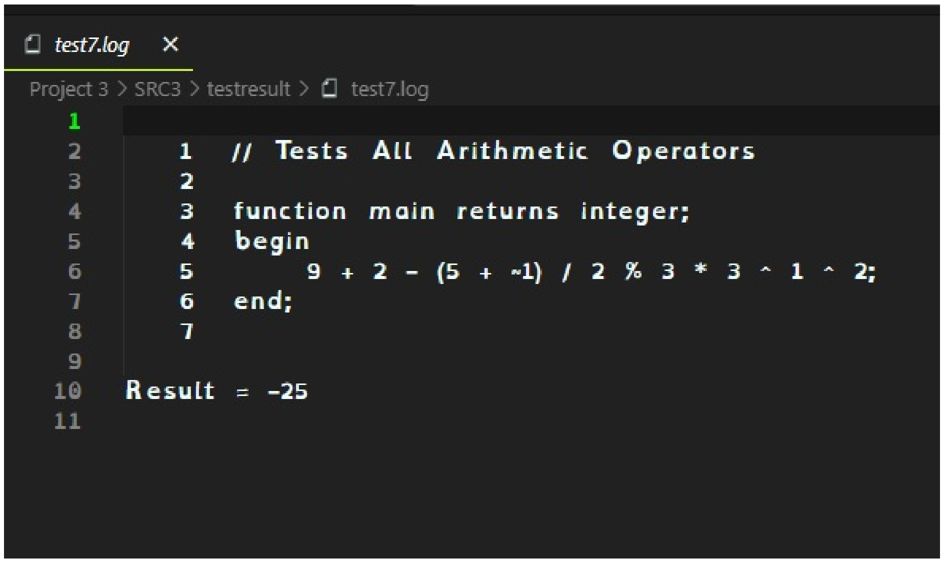


```
test6.log
Project 3 > SRC3 > testresult > test6.log

   1  // Function with Character Literal Escape Characters
   2
   3  function main returns character;
   4       lines: integer is 60;
   5  begin
   6       when lines < 60, '\n' : '\f';
   7  end;
   8

Result = 12
```

<$n#figure:TEST7>



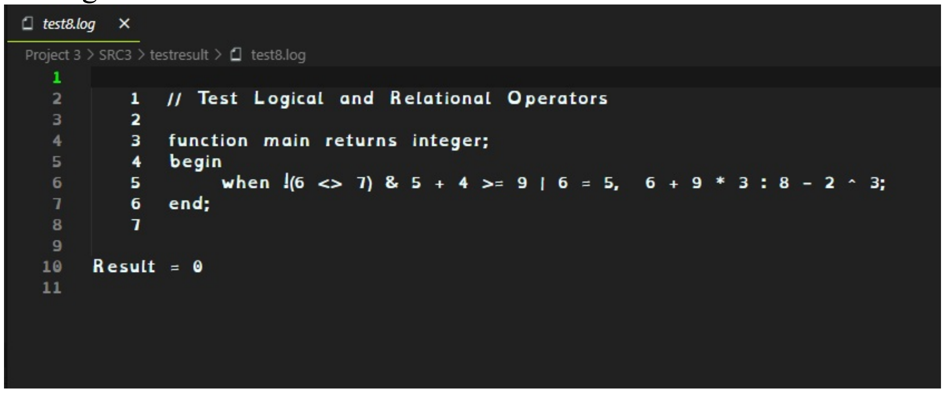
test7.log

Project 3 > SRC3 > testresult > test7.log

```
    1   // Tests All Arithmetic Operators
    2
    3   function main returns integer;
    4   begin
    5       9 + 2 - (5 + ~1) / 2 % 3 * 3 ^ 1 ^ 2;
    6   end;
    7

Result = -25
```

<$n#figure:TEST8>



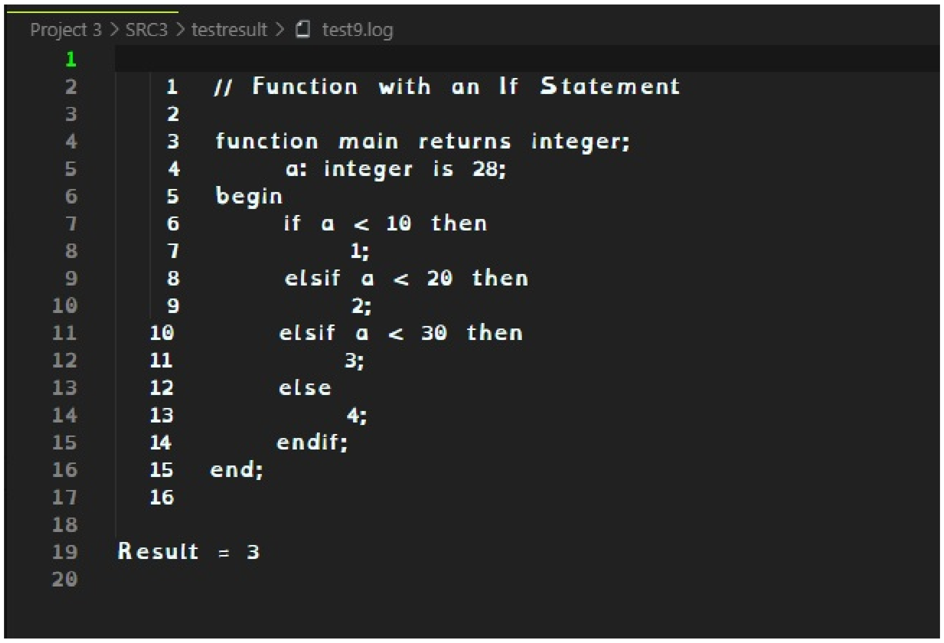
test8.log

Project 3 > SRC3 > testresult > test8.log

```
    1   // Test Logical and Relational Operators
    2
    3   function main returns integer;
    4   begin
    5       when !(6 <> 7) & 5 + 4 >= 9 | 6 = 5,  6 + 9 * 3 : 8 - 2 ^ 3;
    6   end;
    7

Result = 0
```

<$n#figure:TEST9>

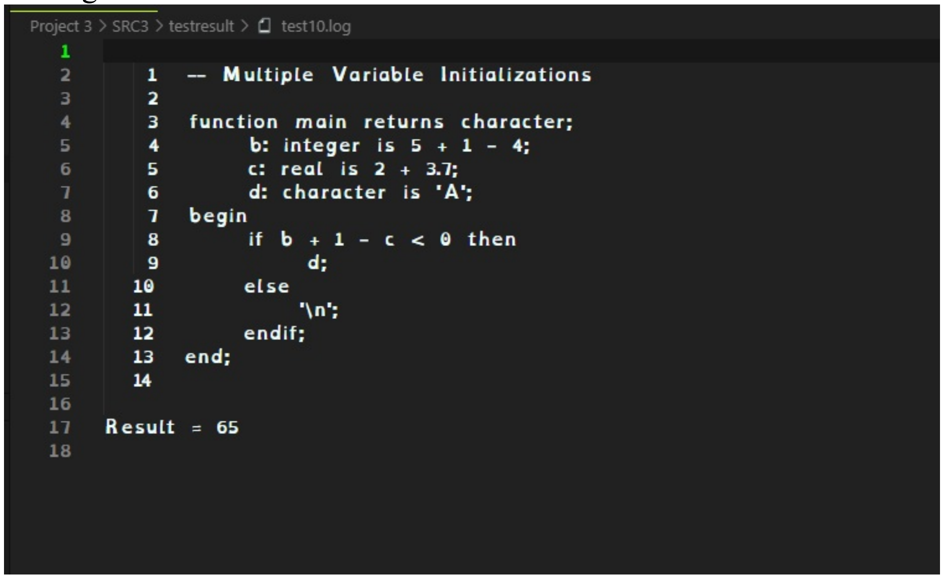


Project 3 > SRC3 > testresult > test9.log

```
1   // Function with an If Statement
2
3   function main returns integer;
4        a: integer is 28;
5   begin
6        if a < 10 then
7             1;
8        elsif a < 20 then
9             2;
10       elsif a < 30 then
11            3;
12       else
13            4;
14       endif;
15  end;
16

Result = 3
```

<$n#figure:TEST10>

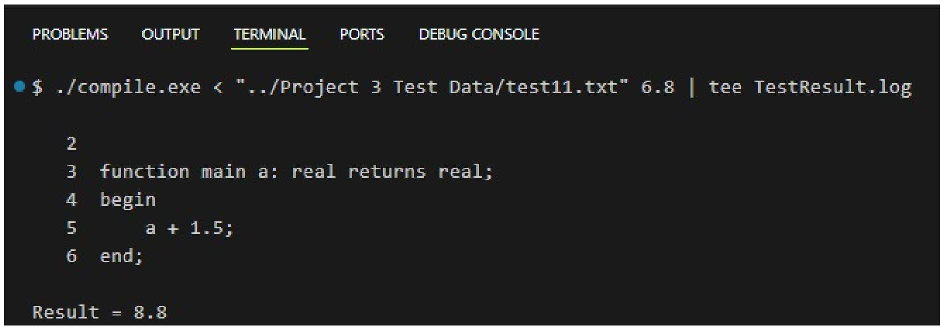


Project 3 > SRC3 > testresult > test10.log

```
1   -- Multiple Variable Initializations
2
3   function main returns character;
4        b: integer is 5 + 1 - 4;
5        c: real is 2 + 3.7;
6        d: character is 'A';
7   begin
8        if b + 1 - c < 0 then
9             d;
10       else
11            '\n';
12       endif;
13  end;
14

Result = 65
```

<$n#figure:TEST11>

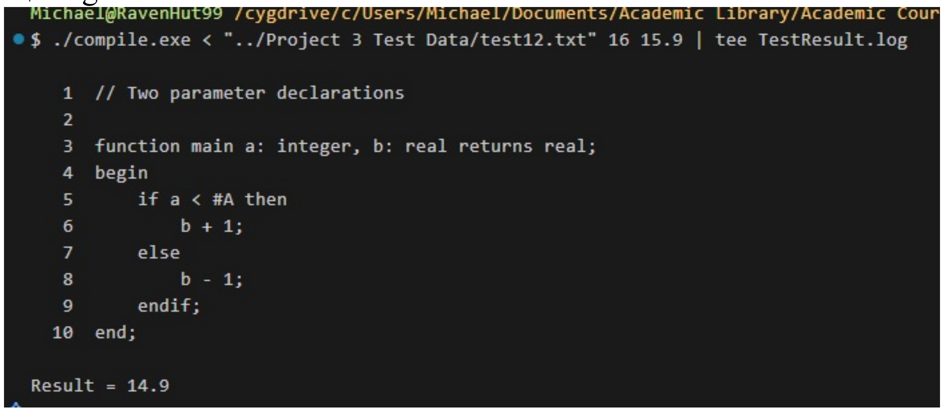

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
$ ./compile.exe < "../Project 3 Test Data/test11.txt" 6.8 | tee TestResult.log


function main a: real returns real;
begin
    a + 1.5;
end;

Result = 8.8
```

<$n#figure:TEST12>



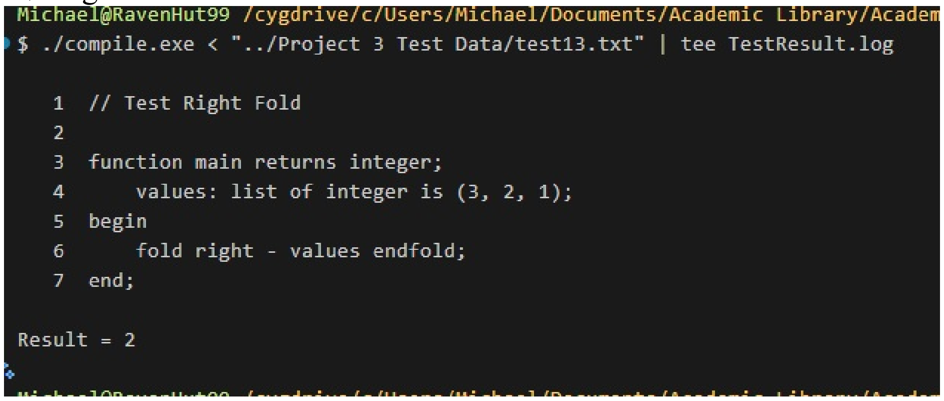
```
Michael@RavenHut99 /cygdrive/c/Users/Michael/Documents/Academic Library/Academic Cour
$ ./compile.exe < "../Project 3 Test Data/test12.txt" 16 15.9 | tee TestResult.log

// Two parameter declarations

function main a: integer, b: real returns real;
begin
    if a < #A then
        b + 1;
    else
        b - 1;
    endif;
end;

Result = 14.9
```

<$n#figure:TEST13>



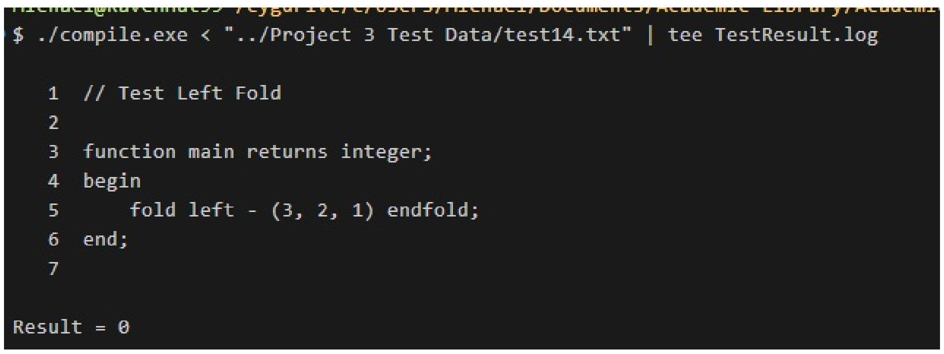
```
Michael@RavenHut99 /cygdrive/c/Users/Michael/Documents/Academic Library/Academ
$ ./compile.exe < "../Project 3 Test Data/test13.txt" | tee TestResult.log

// Test Right Fold

function main returns integer;
    values: list of integer is (3, 2, 1);
begin
    fold right - values endfold;
end;

Result = 2
```

<$n#figure:TEST14>



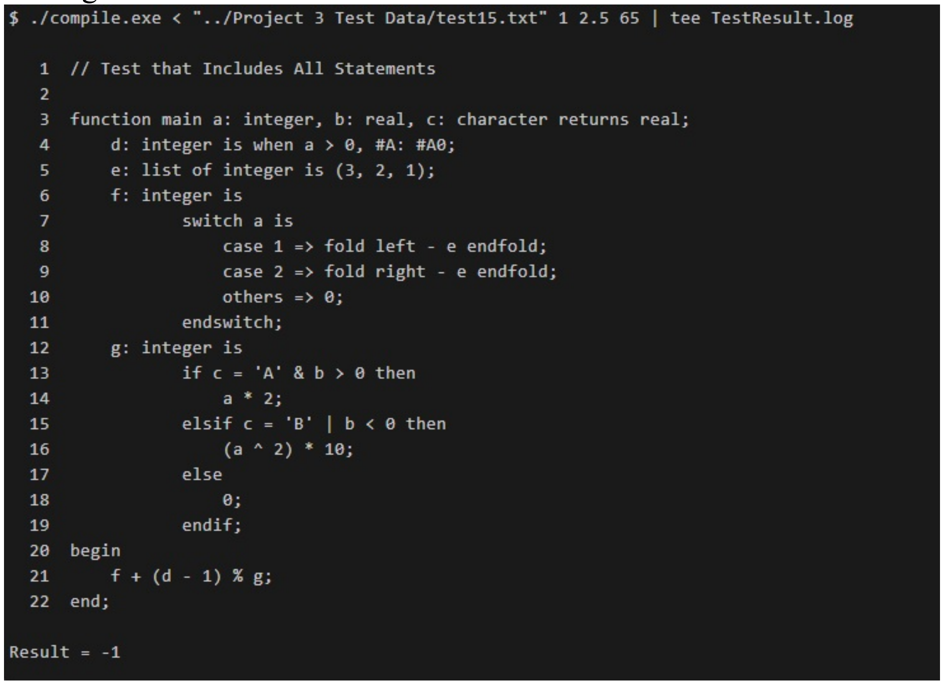
```
$ ./compile.exe < "../Project 3 Test Data/test14.txt" | tee TestResult.log

// Test Left Fold

function main returns integer;
begin
    fold left - (3, 2, 1) endfold;
end;


Result = 0
```

<$n#figure:TEST15>



```
$ ./compile.exe < "../Project 3 Test Data/test15.txt" 1 2.5 65 | tee TestResult.log

// Test that Includes All Statements

function main a: integer, b: real, c: character returns real;
    d: integer is when a > 0, #A: #A0;
    e: list of integer is (3, 2, 1);
    f: integer is
           switch a is
               case 1 => fold left - e endfold;
               case 2 => fold right - e endfold;
               others => 0;
           endswitch;
    g: integer is
           if c = 'A' & b > 0 then
               a * 2;
           elsif c = 'B' | b < 0 then
               (a ^ 2) * 10;
           else
               0;
           endif;
begin
    f + (d - 1) % g;
end;

Result = -1
```

Section 3 – Lessons Learned

1. **Refactoring is not optional.**

   Project 3 forced me to refactor major parts of my Project 2 code, not just extend it. Assumptions that worked for the simpler project often broke down as requirements expanded.

2. **Modularization pays off.**

   The time spent breaking out scanner logic, error reporting, and symbol table management into dedicated modules (an idea I started in Project 2) made debugging and enhancements much easier in Project 3.

3. **Test-driven development works—even for compilers.**
   Creating and running small, focused test cases—starting with those from Project 2—helped surface issues early and often. This was especially important as I added features like folds and complex type handling.
4. **Header and %union consistency is crucial.**
   Many difficult-to-diagnose errors came from mismatched type declarations across modules. Project 2's struggles here prepared me for the even stricter type requirements in Project 3.
5. **Incremental, snapshot-based progress is essential.**
   Having a reliable "known-good" version to fall back on made it possible to experiment boldly with new features and refactoring, knowing I could always restore a working baseline.
6. **Documentation and self-review help avoid repeat mistakes.**
   Recording lessons learned and reviewing my Project 2 missteps (especially around ambiguous grammar and token definitions) prevented me from repeating those errors as the grammar for Project 3 grew more complex.

In summary, Project 3 deepened my understanding of how real-world compilers evolve: theory is a roadmap, but robust, modular, and testable code is what gets you to the finish line. The transition from Project 2 to Project 3 was not just an extension, but an evolution requiring real change in approach, habits, and engineering discipline.