**Project 2 Documentation**

Vincent Testagrossa

University of Maryland Global Campus

CMSC 430: Compiler Theory and Design

Dr. George R. Mikulski

November 5th, 2023

Contents

[Discussion 3](#_Toc150086915)

[Testing 5](#_Toc150086916)

[Function with arithmetic expression 5](#_Toc150086917)

[Function with an Integer Variable 6](#_Toc150086918)

[Function with a Boolean Variable 7](#_Toc150086919)

[Function with a Reduction 8](#_Toc150086920)

[Function with a Real Variable, and Boolean and Real Literals 9](#_Toc150086921)

[Simple Conditional Expression 10](#_Toc150086922)

[Simple Case Selection 11](#_Toc150086923)

[Multiple integer variable initialization 12](#_Toc150086924)

[Single parameter declaration 13](#_Toc150086925)

[Two parameter declarations 14](#_Toc150086926)

[Arithmetic operators 15](#_Toc150086927)

[Relational and logical operators 16](#_Toc150086928)

[Comprehensive test with nested if 17](#_Toc150086929)

[Comprehensive test with nested case 18](#_Toc150086930)

[More relational and logical operators 19](#_Toc150086931)

[Missing operator in expression 20](#_Toc150086932)

[Error in function header, missing colon 21](#_Toc150086933)

[Error in variable declaration 22](#_Toc150086934)

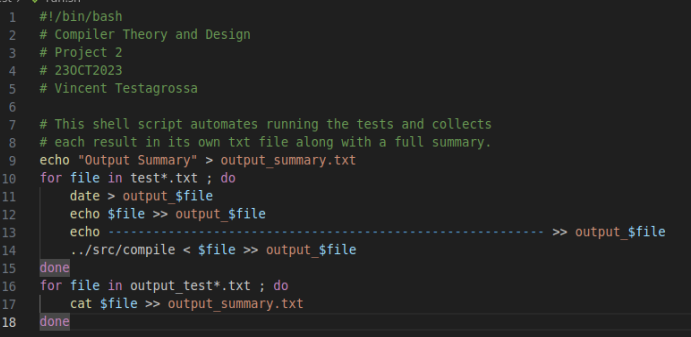
[Multiple Errors, error in when clause and missing others clause 23](#_Toc150086935)

[Multiple Errors 24](#_Toc150086936)

[Reflection 26](#_Toc150086937)

# Discussion

The first step in my approach to the project was to compile the source using the makefile and run ./compile against the test files (that is, after creating the git repository and making the initial commit). I updated the shell script from the previous project to aid in automating the testing process:



I then proceeded with moving over the lexical analyzer from project 1 (while making a backup of the one in the skeleton project, just in case), updated the tokens in the parser, ran the make file, then ran the shell script to test the functions. Tests 1-4 parsed correctly with updated tokens in the parser, so I moved onto including variables, real, real literals, and Boolean literals.

The listing file from project 1 was then copied over to project 2 so the error messages would work correctly, and the error output would be attached to the bottom of the logs. Adding IF and CASE statements to the grammar was relatively straightforward once I added the additional optional\_case production’s left-hand recursion. There was some confusion on using the semicolons at this point, so some adjustments were made once I recognized that I was including erroneous tokens in some of the productions.

The next step was to include optional variables in the function declaration which required a new ‘optional\_variable’ production with the right-hand productions being either a left-recursive production, or nothing. The optional parameters were made similarly, with the only major difference being the inclusion of a comma nonterminal to separate each parameter.

The longest part of the process was getting the logical operators to follow the correct precedence and play nicely with each other. Productions were made for each in the order:

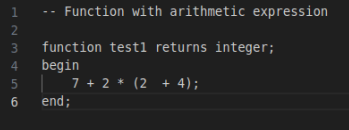
1. Or\_expression
2. And\_expression
3. Not\_expression

This was to allow the bottom-up parsing to follow the correct precedence rules. I did include an additional expression production that simply linked to or\_expression to avoid replacing every mention of expression in other productions. The issues that made this part take so much more time are discussed in the ‘Reflection’ section.

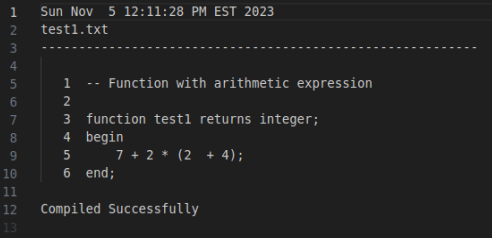
# Testing

## Function with arithmetic expression

Input:

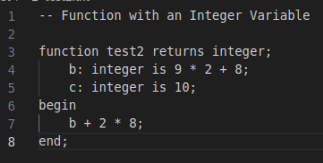


Output:

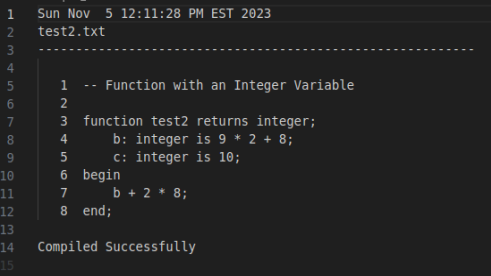


## Function with an Integer Variable

Input:

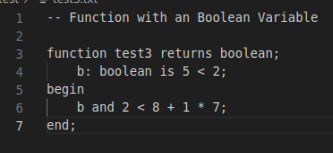


Output:

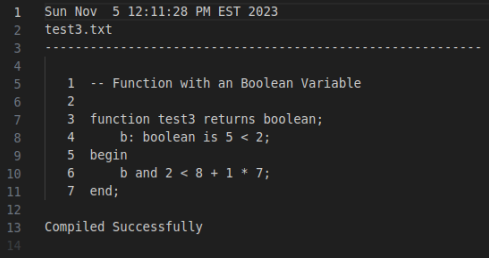


## Function with a Boolean Variable

Input:

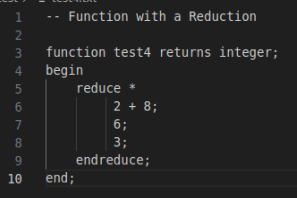


Output:

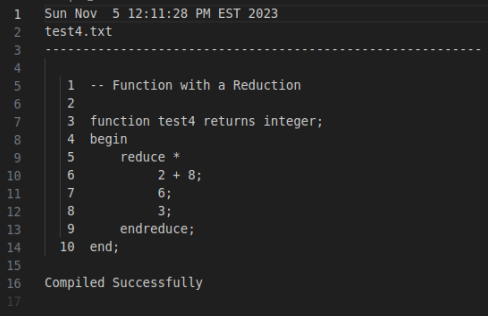


## Function with a Reduction

Input:

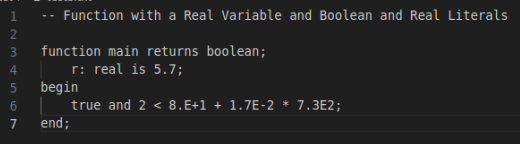


Output:

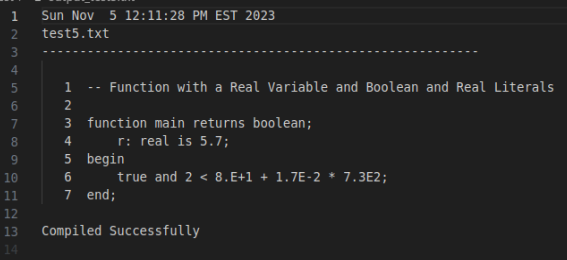


## Function with a Real Variable, and Boolean and Real Literals

Input:

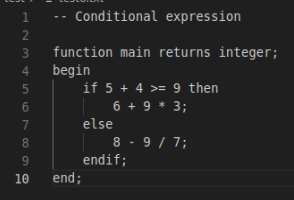


Output:

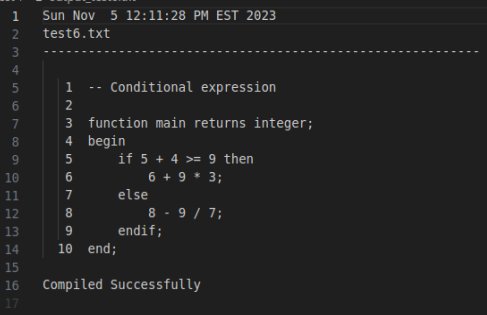


## Simple Conditional Expression

Input:

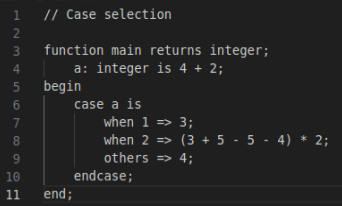


Output:

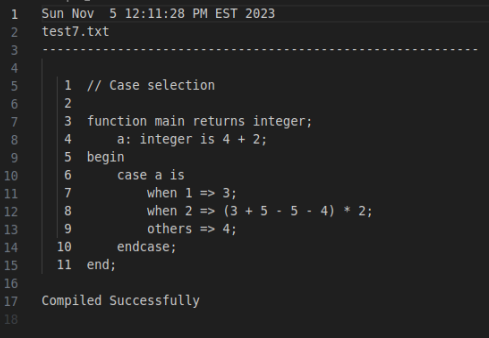


## Simple Case Selection

Input:

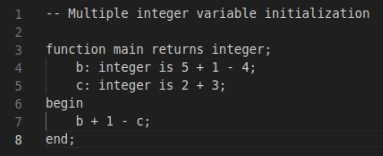


Output:

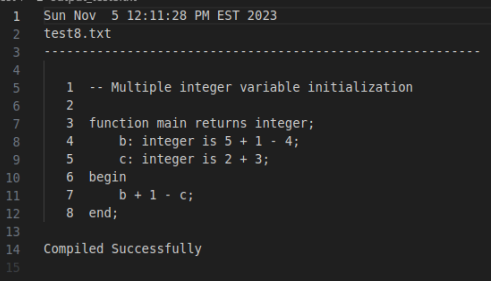


## Multiple integer variable initialization

Input:

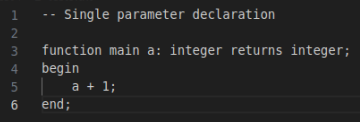


Output:

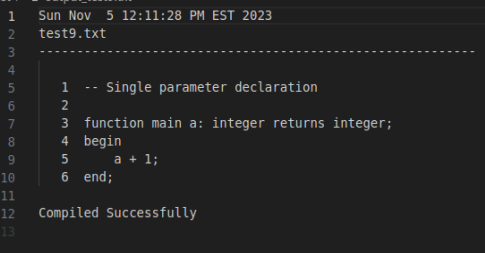


## Single parameter declaration

Input:

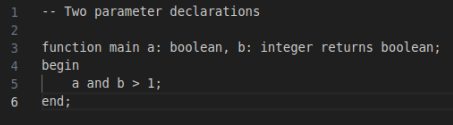


Output:

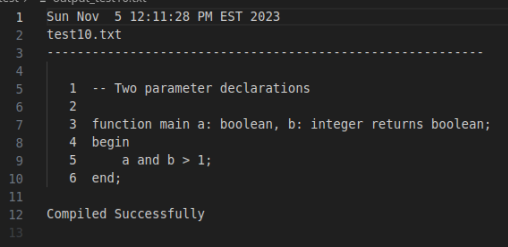


## Two parameter declarations

Input:

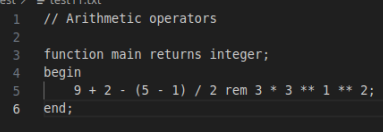


Output:

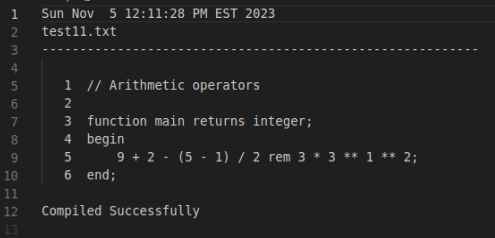


## Arithmetic operators

Input:

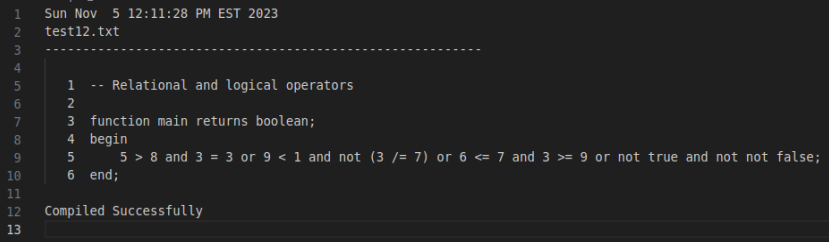


Output:

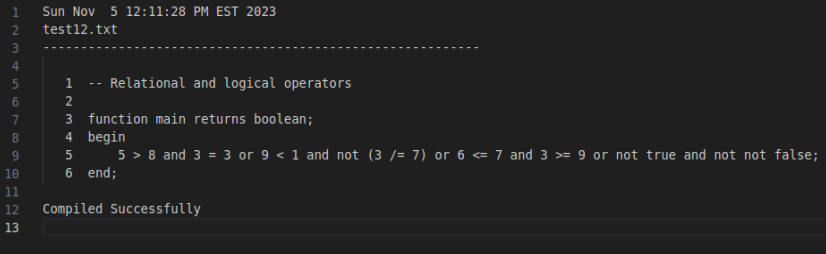


## Relational and logical operators

Input:

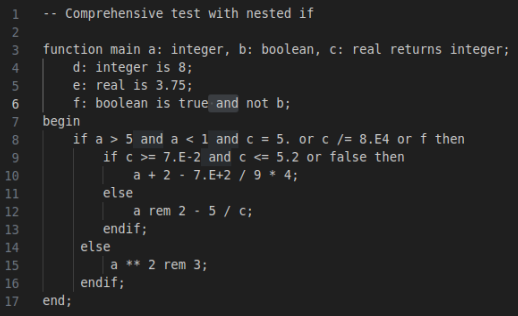


Output:

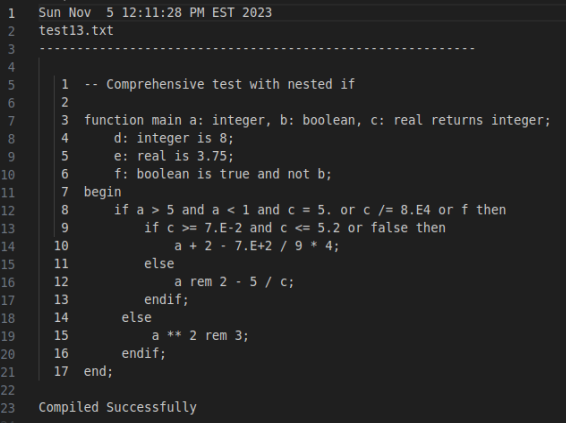


## Comprehensive test with nested if

Input:

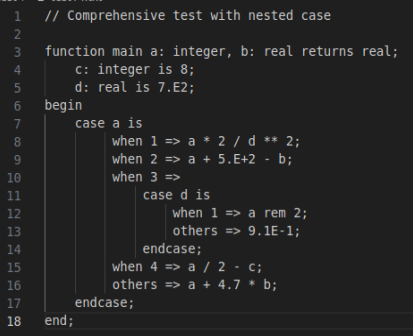


Output:

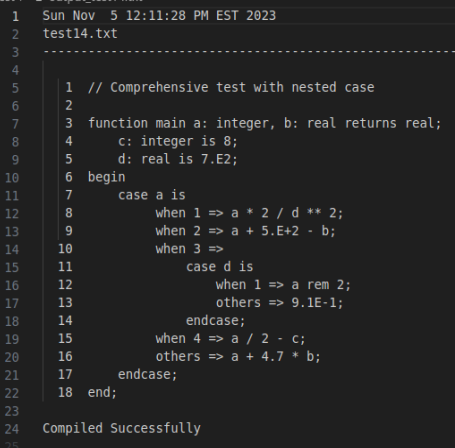


## Comprehensive test with nested case

Input:

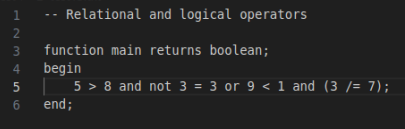


Output:

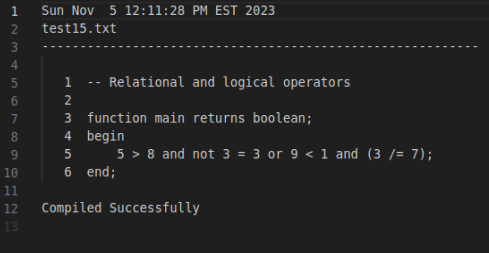


## More relational and logical operators

Input:

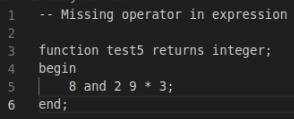


Output:

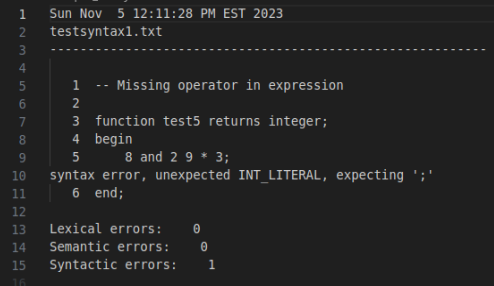


## Missing operator in expression

Input:

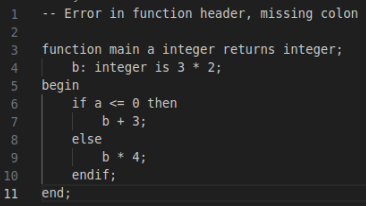


Output:

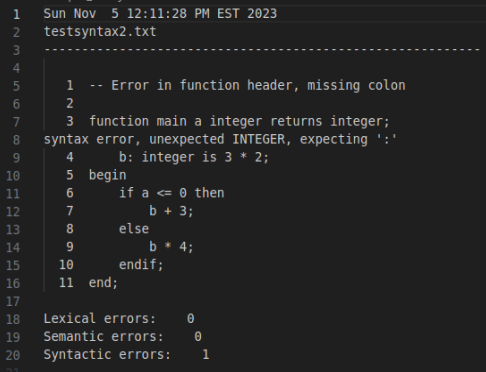


## Error in function header, missing colon

Input:

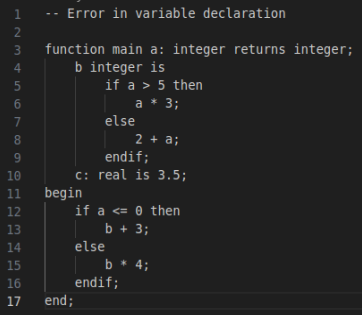


Output:

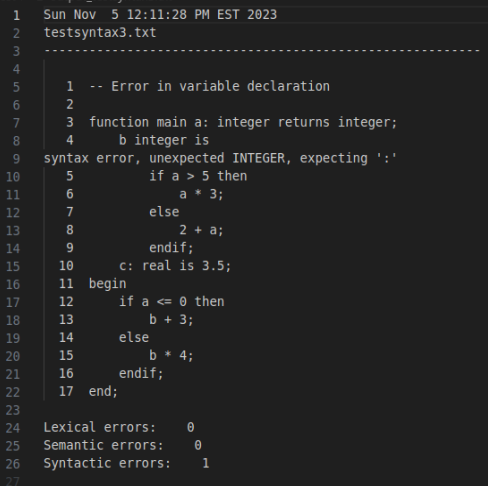


## Error in variable declaration

Input:

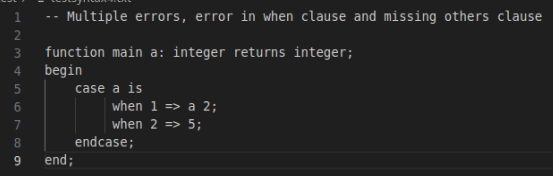


Output:

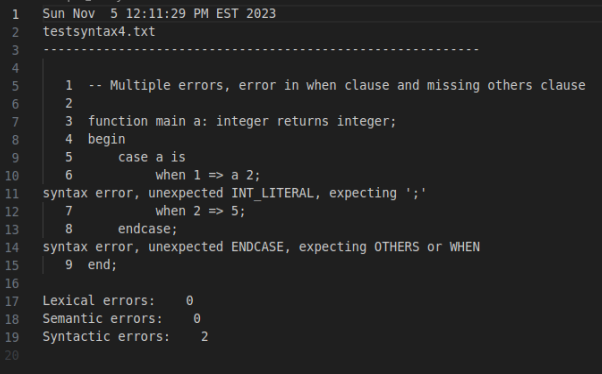


## Multiple Errors, error in when clause and missing others clause

Input:

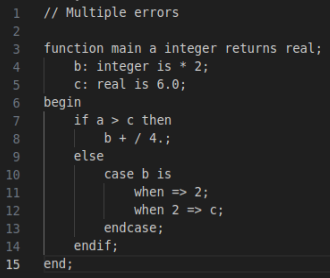


Output:

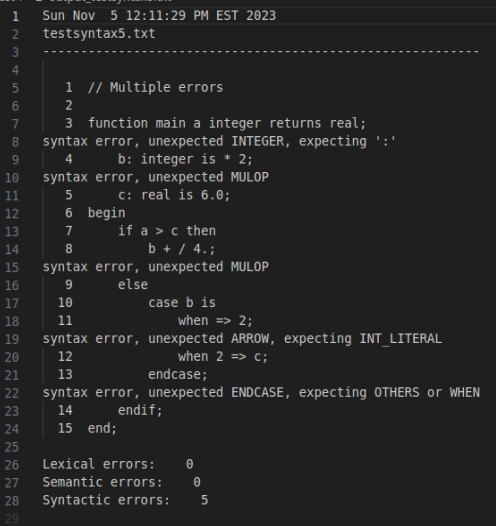


## Multiple Errors

Input:



Output:



# Reflection

This project taught me a lot more about parsing than I already knew. Having created top-down parsers for multiple projects before, learning how bottom-parsing works and the larger subset of languages they can be used to parse was very interesting. It took a little while before I realized that Yacc would generate the tokens, so I wasted some time adding the tokens to the header file a few times before I recognized what was happening. Part of the reason is that I prefer to learn by doing, so I only really skimmed the material before getting started on the project, but that’s part of my learning process.

Getting the productions for the arithmetic operations didn’t take very long, but I ran into some issues with the logical operators. I made each production separate and had the order correct to match the precedence rules, but my not\_expression production was causing problems because I wasn’t making it “fully” right recursive. I had ‘NOTOP not\_expression relation’ as the first right-hand production and whenever there was ‘and not’, the parse tree was incorrect and causing syntax errors elsewhere. I fixed this by removing the ‘relation’ nonterminal from that right-hand production and everything started compiling correctly.

One thing I noticed about adding more right-hand productions for errors was that it didn’t seem entirely necessary for the test cases that were provided, other than within the function header. I added them anyway, just in case I was missing something, but I couldn’t figure out a way to test that they would be needed to recover from errors.