PROJECT 1 REPORT

CMSC 430 – COMPILER THEORY AND DESIGN

[Robert D. Carswell]

[23 January 2024]

Contents

Executive Summary

The project focuses on enhancing the existing lexical analyzer and compilation listing generator code, explicitly targeting the scanner.l file. The modifications include adding new language features, including reserved words such as else, elsif, endfold, endif, fold, if, left, real, right, and then. Each reserved word becomes a distinct token represented in all uppercase. Two logical operators, | (OROP) and ! (NOTOP), as well as five relational operators (=, <>, >, >=, and <=) represented by the single token RELOP, are introduced. The subtraction operator (-) becomes a new lexeme for the ADDOP token, while the division operator (/) is added for the MULOP token. Additionally, new tokens include REMOP (remainder operator, %), EXPOP (exponentiation operator, ^), and NEGOP (unary minus operator, ~). A new type of comment, starting with -- and ending with the end of the line, is introduced. Identifier definitions are modified to allow underscores, with restrictions on consecutive underscores. Two new types of literals are added: hexadecimal integers (starting with # followed by digits or A-F) and real literals (allowing digits, decimal point, and an optional exponent). Character literals are modified to permit five additional escape characters: '\b', '\t', '\n', '\b', and '\f'. Additionally, the tokens.h header file is updated to include definitions for each new token. These major changes greatly increase the lexical analyzer's abilities, making them more flexible to support a wider range of language structures and improving the accuracy of the compilation listing generator.

Testing

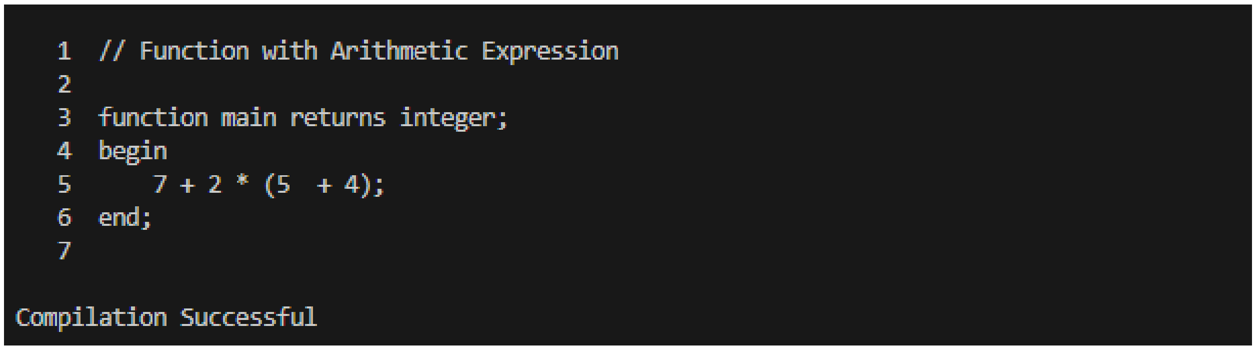
Test Case Table

Test Cases	Description	Expected Output	Actual Output	Pass/Fail
Test Case 1	Read test file 1	File contents with : Msg (Bottom of file): Compilation Successful	File contents with : Msg (Bottom of file): Compilation Successful	Pass
Test Case 2	Read test file 2	File contents with: Msg (Under line 5): Lexical Error, Invalid Character $ Msg (Bottom of file): Lexical Errors: 1 Syntax Errors: 0 Semantic Errors: 0	File contents with: Msg (Under line 5): Lexical Error, Invalid Character $ Msg (Bottom of file): Lexical Errors: 1 Syntax Errors: 0 Semantic Errors: 0	Pass
Test Case 3	Read test file 3	File contents with : Msg (Bottom of file): Compilation Successful	File contents with : Msg (Bottom of file): Compilation Successful	Pass
Test Case 3	Read test file 4	File contents with : Msg (Bottom of file): Compilation Successful	File contents with : Msg (Bottom of file): Compilation Successful	Pass
Test Case 3	Read test file 5	File contents with : Msg (Bottom of file): Compilation Successful	File contents with : Msg (Bottom of file): Compilation Successful	Pass
Test Case 3	Read test file 6	File contents with : Msg (Bottom of file): Compilation Successful	File contents with : Msg (Bottom of file): Compilation Successful	Pass
Test Case 3	Read test file 7	File contents with: Msg (Under line 5): Lexical Error, Invalid Character $ Msg (Under line 5): Lexical Error, Invalid Character ?	File contents with: Msg (Under line 5): Lexical Error, Invalid Character $ Msg (Under line 5): Lexical Error, Invalid Character ?	Pass

Test Cases	Description	Expected Output	Actual Output	Pass/Fail
		Msg (Bottom of file): Lexical Errors: 2 Syntax Errors: 0 Semantic Errors: 0	Msg (Bottom of file): Lexical Errors: 2 Syntax Errors: 0 Semantic Errors: 0	
Test Case 3	Read test file 6	File contents with: Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 13): Lexical Error, Invalid Character _ Msg (Under line 14): Lexical Error, Invalid Character _ Msg (Bottom of file): Lexical Errors: 5 Syntax Errors: 0 Semantic Errors: 0	File contents with: Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 12): Lexical Error, Invalid Character _ Msg (Under line 13): Lexical Error, Invalid Character _ Msg (Under line 14): Lexical Error, Invalid Character _ Msg (Bottom of file): Lexical Errors: 5 Syntax Errors: 0 Semantic Errors: 0	Pass

Test Case Screenshots

Test Case 1

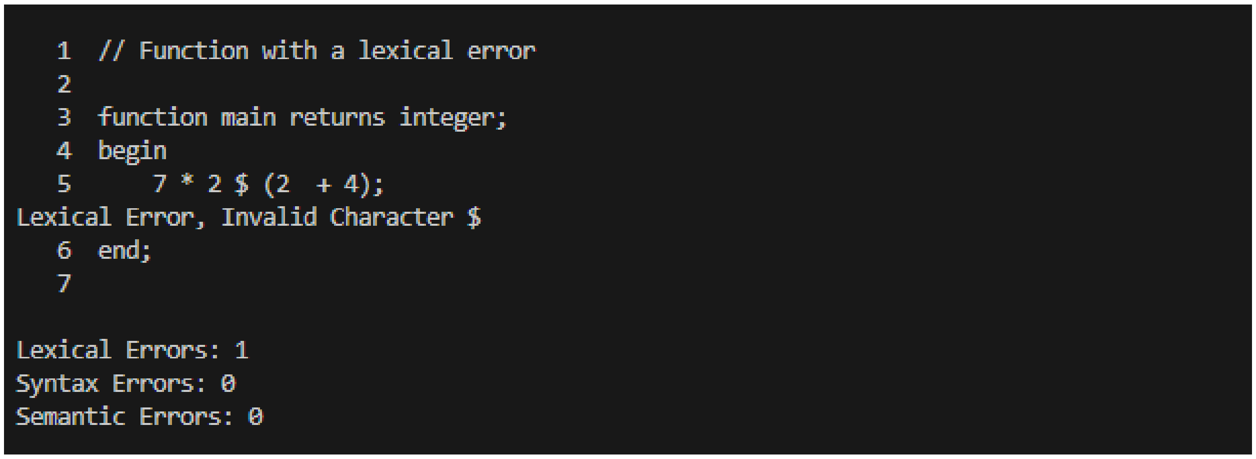


```
// Function with Arithmetic Expression

function main returns integer;
begin
    7 + 2 * (5  + 4);
end;


Compilation Successful
```

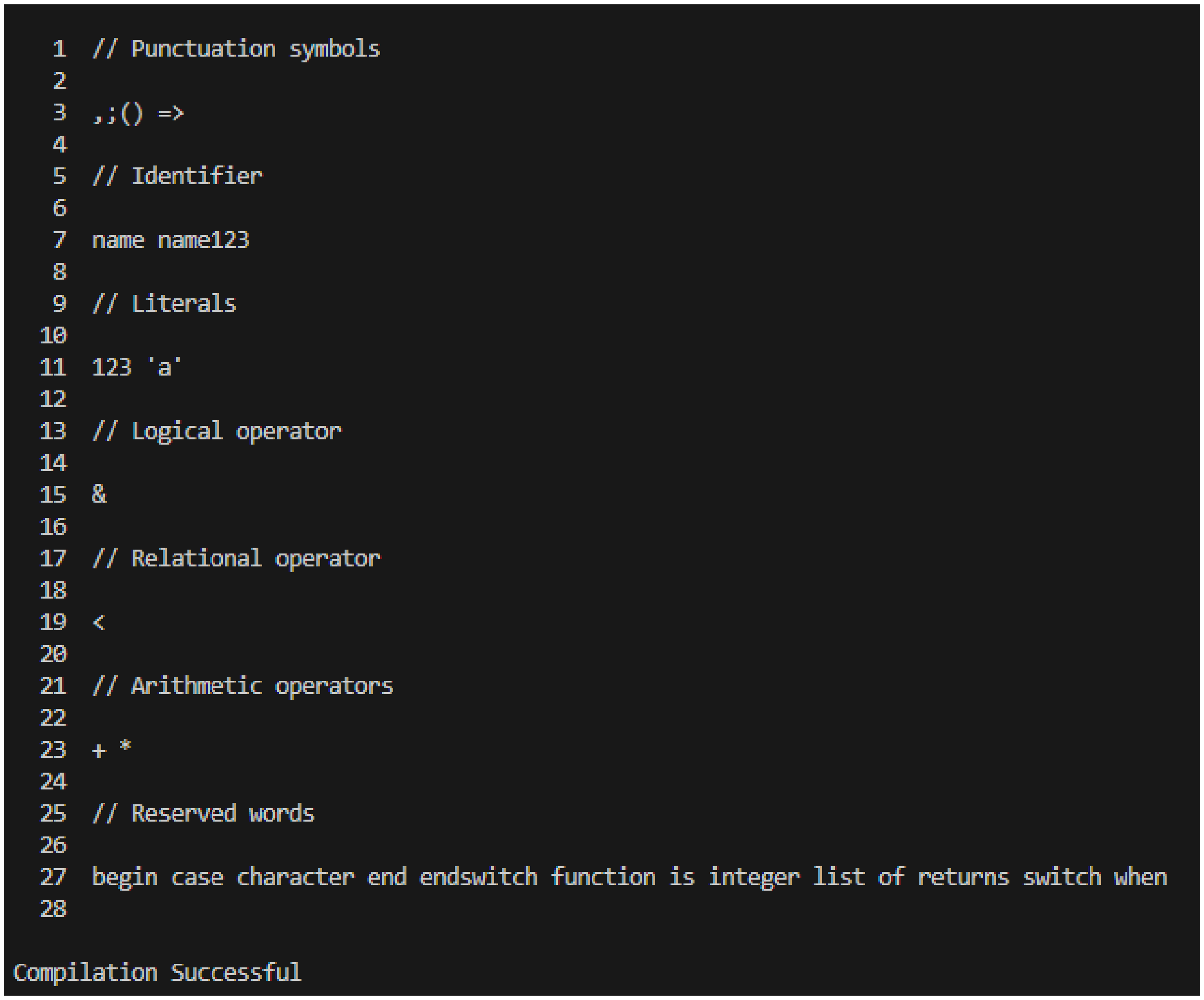
Test Case 2



```
   1  // Function with a lexical error
   2
   3  function main returns integer;
   4  begin
   5      7 * 2 $ (2  + 4);
Lexical Error, Invalid Character $
   6  end;
   7

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

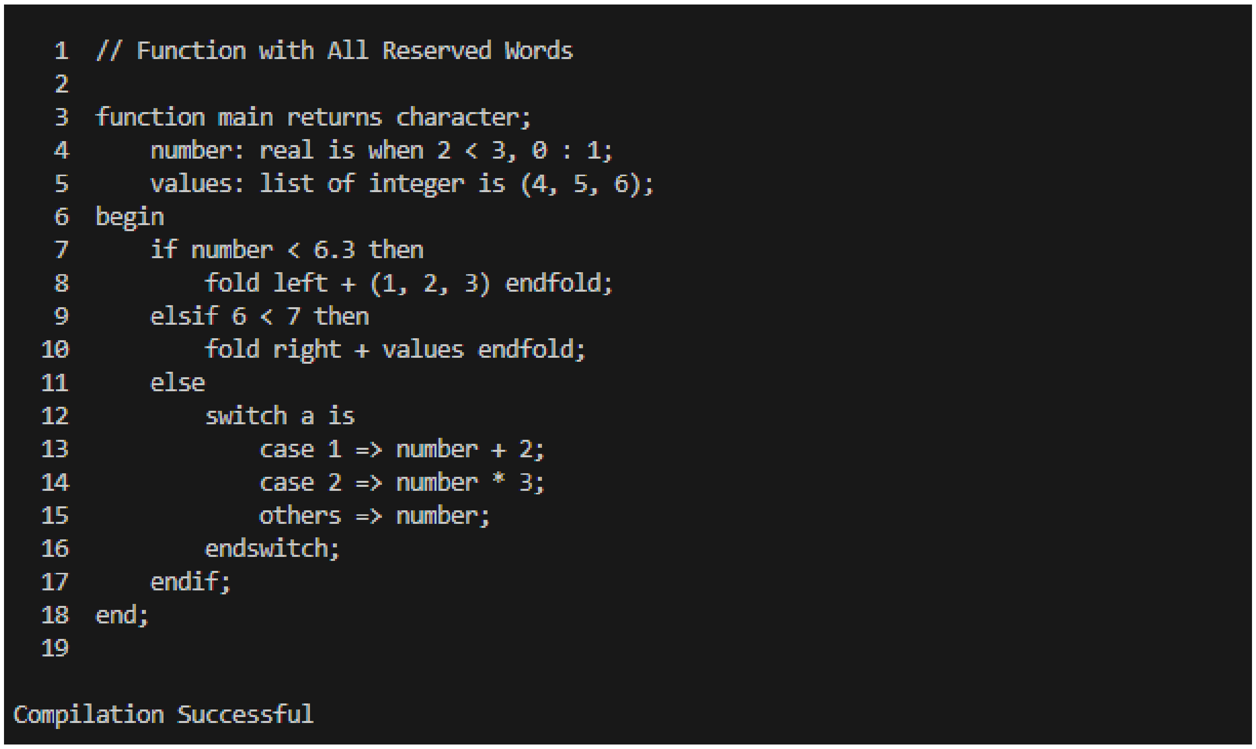
Test Case 3



```
   1  // Punctuation symbols
   2
   3  ,;() =>
   4
   5  // Identifier
   6
   7  name name123
   8
   9  // Literals
  10
  11  123 'a'
  12
  13  // Logical operator
  14
  15  &
  16
  17  // Relational operator
  18
  19  <
  20
  21  // Arithmetic operators
  22
  23  + *
  24
  25  // Reserved words
  26
  27  begin case character end endswitch function is integer list of returns switch when
  28

Compilation Successful
```

Test Case 4



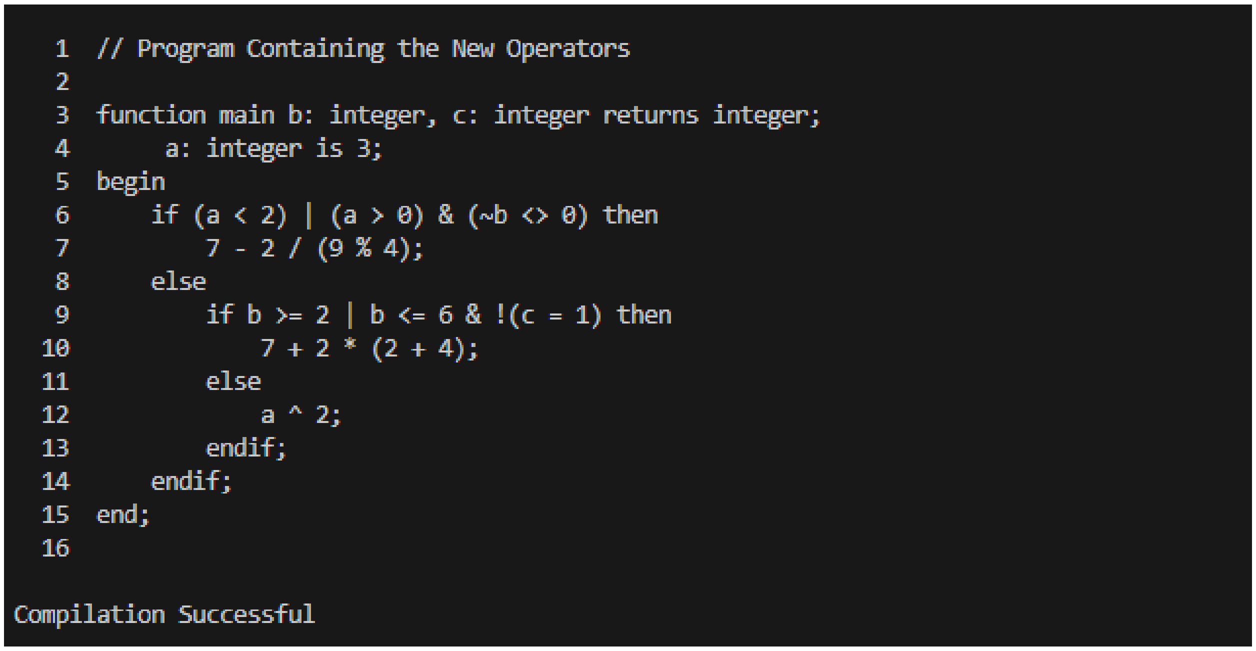
```
// Function with All Reserved Words

function main returns character;
    number: real is when 2 < 3, 0 : 1;
    values: list of integer is (4, 5, 6);
begin
    if number < 6.3 then
        fold left + (1, 2, 3) endfold;
    elsif 6 < 7 then
        fold right + values endfold;
    else
        switch a is
            case 1 => number + 2;
            case 2 => number * 3;
            others => number;
        endswitch;
    endif;
end;

```

```
Compilation Successful
```

Test Case 5



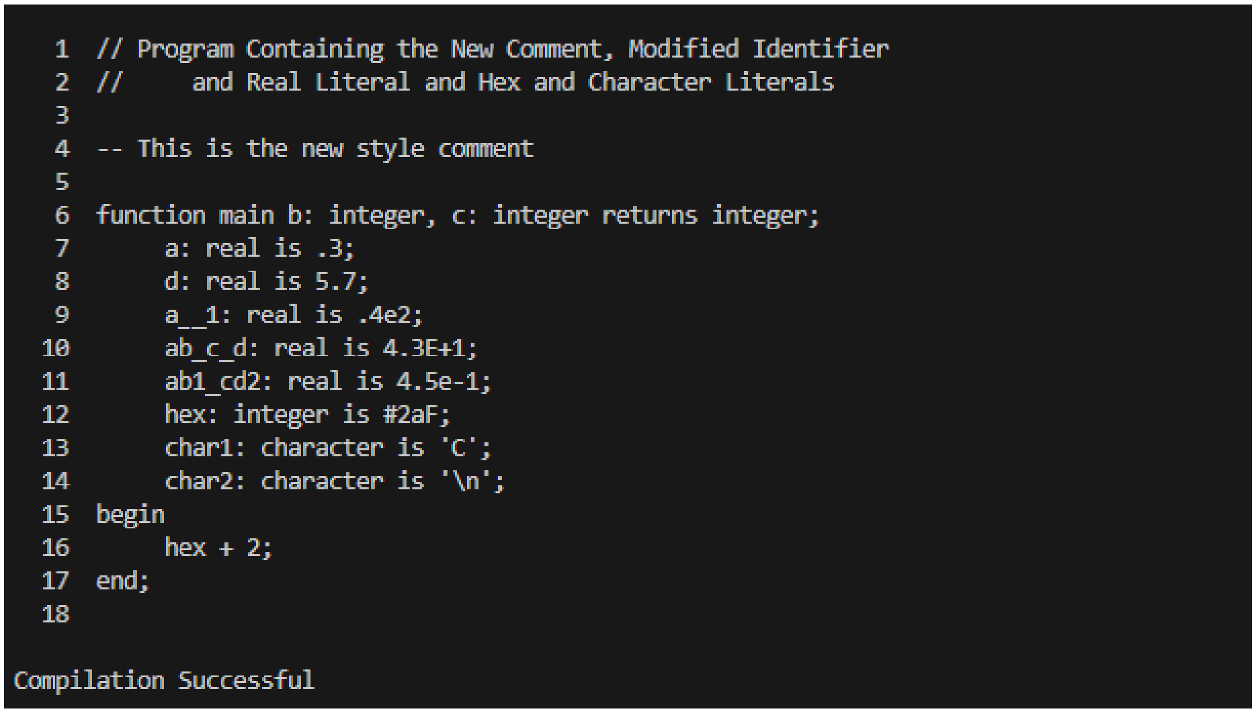
```
// Program Containing the New Operators

function main b: integer, c: integer returns integer;
     a: integer is 3;
begin
    if (a < 2) | (a > 0) & (~b <> 0) then
        7 - 2 / (9 % 4);
    else
        if b >= 2 | b <= 6 & !(c = 1) then
            7 + 2 * (2 + 4);
        else
            a ^ 2;
        endif;
    endif;
end;

```

```
Compilation Successful
```

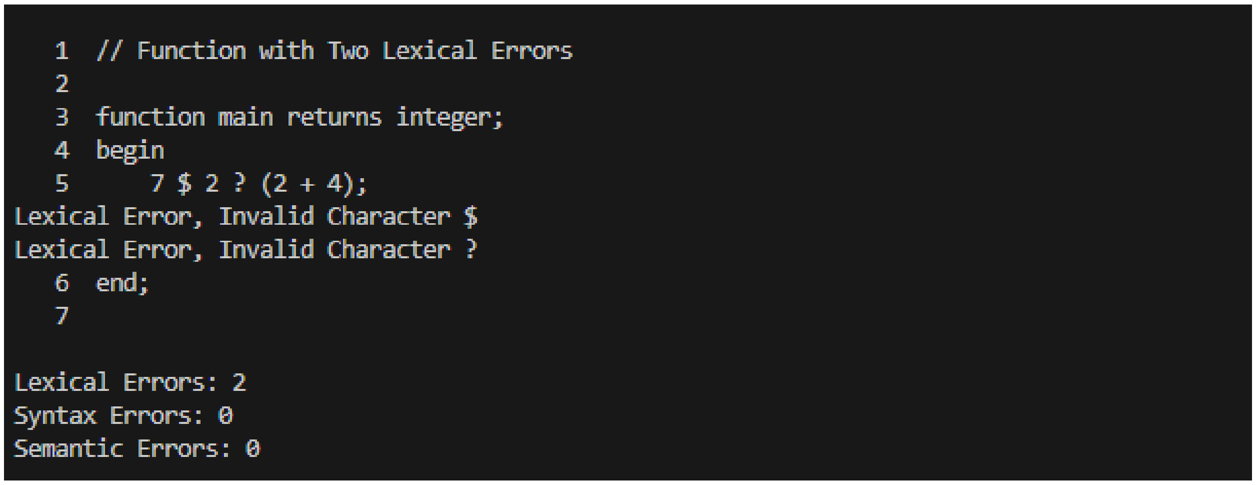
Test Case 6



```
   1  // Program Containing the New Comment, Modified Identifier
   2  //     and Real Literal and Hex and Character Literals
   3
   4  -- This is the new style comment
   5
   6  function main b: integer, c: integer returns integer;
   7       a: real is .3;
   8       d: real is 5.7;
   9       a__1: real is .4e2;
  10       ab_c_d: real is 4.3E+1;
  11       ab1_cd2: real is 4.5e-1;
  12       hex: integer is #2aF;
  13       char1: character is 'C';
  14       char2: character is '\n';
  15  begin
  16       hex + 2;
  17  end;
  18

Compilation Successful
```

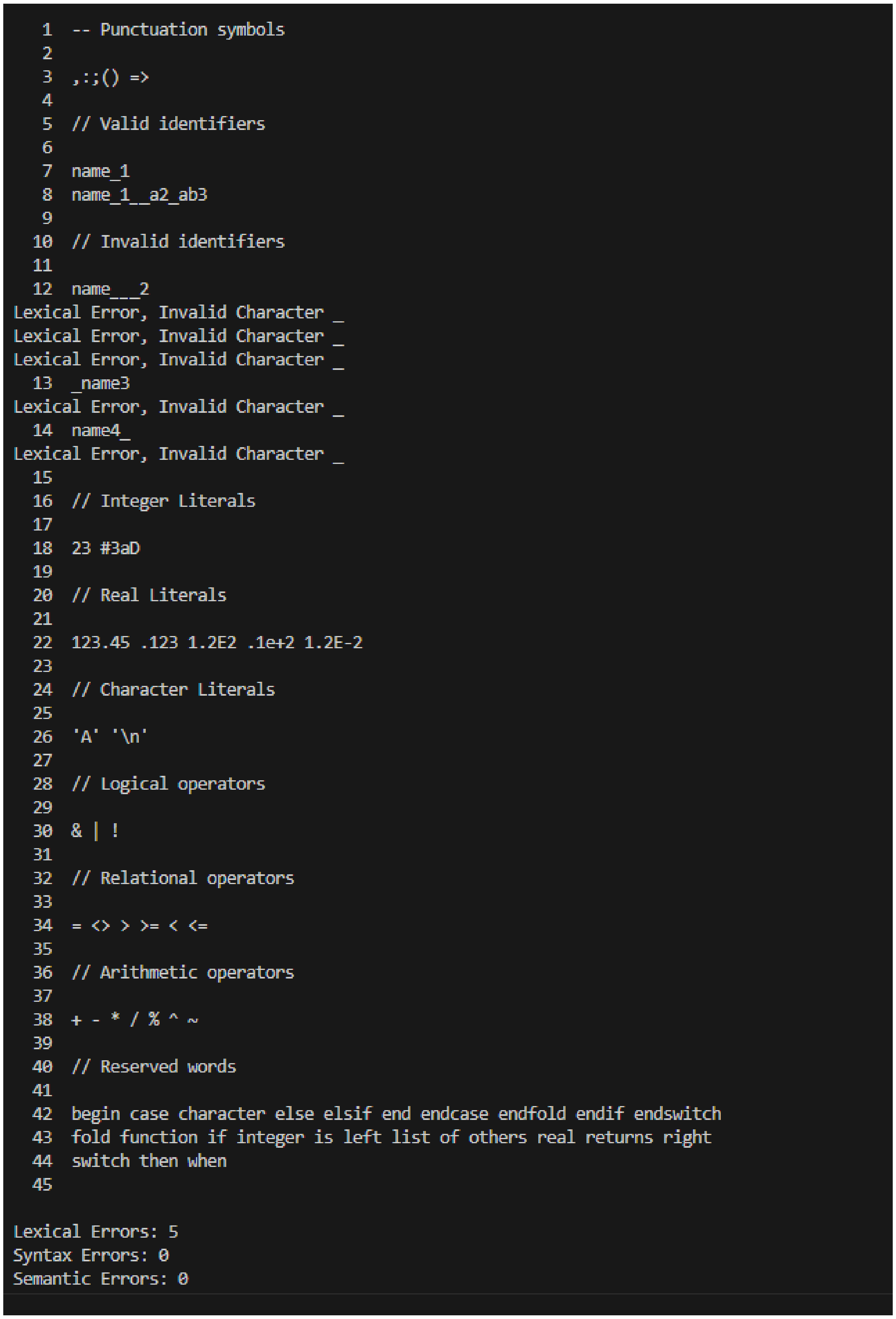
Test Case 7



```
   1  // Function with Two Lexical Errors
   2
   3  function main returns integer;
   4  begin
   5      7 $ 2 ? (2 + 4);
Lexical Error, Invalid Character $
Lexical Error, Invalid Character ?
   6  end;
   7

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

Test Case 8



```
   1  -- Punctuation symbols
   2
   3  ,:;() =>
   4
   5  // Valid identifiers
   6
   7  name_1
   8  name_1__a2_ab3
   9
  10  // Invalid identifiers
  11
  12  name___2
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
Lexical Error, Invalid Character _
  13  _name3
Lexical Error, Invalid Character _
  14  name4_
Lexical Error, Invalid Character _
  15
  16  // Integer Literals
  17
  18  23 #3aD
  19
  20  // Real Literals
  21
  22  123.45 .123 1.2E2 .1e+2 1.2E-2
  23
  24  // Character Literals
  25
  26  'A' '\n'
  27
  28  // Logical operators
  29
  30  & | !
  31
  32  // Relational operators
  33
  34  = <> > >= < <=
  35
  36  // Arithmetic operators
  37
  38  + - * / % ^ ~
  39
  40  // Reserved words
  41
  42  begin case character else elsif end endcase endfold endif endswitch
  43  fold function if integer is left list of others real returns right
  44  switch then when
  45

Lexical Errors: 5
Syntax Errors: 0
Semantic Errors: 0
```

Approach

In building upon the foundational knowledge acquired during week one, which primarily covered the general structure of the program and the interaction between its components, I focused on implementing changes according to requirements 1–8 as outlined in the instructions. For requirements 9–13, a shift in perspective was essential, recognizing the potential unintended outcomes of regular expressions with even minor alterations. Adopting a literal approach using the expression ('.'|'\\b'|'\\t'|'\\n'|'\\b'|'\\f'), I ensured that the program only accepted the specific escape characters requested. This meticulous approach underscores the necessity for clear rules and standards in software development, especially when handling literals. The lack of well-established guidelines risks users not achieving their intended outcomes. Therefore, continuous review and validation throughout the software development lifecycle become imperative to align the work with project objectives, emphasizing a commitment to both consistency and quality.

Lessons Learned

The literal approach vividly illustrates the consequences of lacking firm rules and standards, as users may not obtain their desired outcomes, particularly in requirements 9–13. Regular expressions were employed to formalize acceptance parameters, acknowledging that misunderstanding the requirements could yield unintended results. Requirement 13 mandated the modification of character literals with an existing '.' to accept five additional escape characters using a '\f' format. A seemingly straightforward solution of escaping the backslash ('\\.') led to complexities, as characters previously accepted, like 'a,' would no longer be valid, potentially causing errors. Such straightforward misunderstandings highlight the potential difficulties in the complex environment where compilers operate, constrained by rules governing their behavior to produce expected results consistently.However, the approach document provided was more than helpful and kept me within the scope of the requirements.