

# Custom Mini Programming Language Processor

## Phase 1: Language Design and Lexical Analysis (Scanner)

CMPE 458

Prepared By Group #2

Hendrix Gryspeerdt – 20337154

Monica Saad - 20348798

Ryan Silverberg - 20342023

Simon John – 20348233

February 6<sup>th</sup>, 2025

# Overview of Modifications

In this stage of the project, the team implemented a lexical analyzer. This included adding recognition and organization of tokens and their positions relevant to the input text. This code was used to determine the length of tokens and their location to later be used for meaningful error handling.

The new types of tokens able to be scanned include keywords, identifiers, operators, numbers, string literals, punctuators, and errors. This allows for full handling of the programming language with opportunity for further additions.

The purpose of the identifiers/keywords is to allow for custom variable naming and recognition of control flow like “if”, “else”, “repeat”, and “until”. Keywords are recognized using a constant list of strings, and this is the same for recognizing operators in the code.

Numbers also were implemented and recognized separately as Integers and Floats by looking for a decimal point present in the entire token.

String literals were recognized by their prefix which was is quotation mark, and the implementation makes sure that it is terminated otherwise the token is recognized as an error. Finally, punctuators were added to allow for recognition of comments, proper line termination, and brackets. Multi-line comments were implemented to be recognized as starting with “!” while single line comments are recognized as beginning with “?”.

Throughout all present modifications, error detection was added to each token parsing algorithm to allow for the lexer to recognize if there was a problem within the scanning of the current token and report it to the user.

## List of Changes

- Whitespace & comment removal
  - Single-line comments (?? ...): skipped entirely
  - Multi-line comments (?! ... !?): everything between ?! and !? is ignored, if !? is missing, the lexer throws an unterminated comment error
- Predefined list for keyword recognition:
  - `static const char *keywords[] = {"if", "else", "while", "factorial", "repeat", "until", "int", "string"};`
  - If an identifier matches one of these during tokenization, it is classified as `TOKEN_KEYWORD`
- String literal handling:
  - Maximum length: 256 (including quotes)
  - Errors handled: unterminated string (missing closing “), exceeding maximum length
- Number recognition
  - Handling integers and floating point numbers
  - Invalid number formats: multiple decimals (e.g. 123.45) extra decimals (e.g. 1.2.3), missing decimal digit (e.g. 12.)

- Invalid cases are flagged as ERROR\_INVALID\_NUMBER
- Checking valid operator sequences:
  - Operators are matched using a lookup function isOperatorStr()
  - Valid single operators: =, +, -, \*, /, ~, |, &, ^, !, >, <
  - Valid multi operators: ==, !=, >=, <=, +=, -=, \*=, /=, |=, &=, ^=, <<, >>, &&, ||, ++, --
- Specific error messages:
  - The lexer prints detailed error messages in a compiler-like format:  
 <file>:<line>:<column>: <error message>  
 <code snippet>  
 ^~~ (error highlighting)
  - No error: "No error"
  - Invalid character: "error: invalid character"
  - Invalid number: "error: invalid number"
  - Unterminated string: "error: unterminated string literal"
  - String too long: "error: string literal too long"
  - Unterminated comment: "error: unterminated comment, missing matching '!'?"
  - Default: "Unknown error"
- Correct file extensions:
  - The program checks if the input filename ends in .cisc, if not, it exits with an error and prints the following message: "Incorrect file extension, the correct extension is .cisc"

## New Tokens, Grammar Rules & Semantic Operations

The following is a list of tokens added/modified; examples and regex rules for each token type are displayed in Table 1:

- Removed TOKEN\_NUMBER and replaced with TOKEN\_INTEGER and TOKEN\_FLOAT
- Added TOKEN\_KEYWORD
- Added TOKEN\_IDENTIFIER
- Added TOKEN\_STRING\_LITERAL
- Added TOKEN\_PUNCTUATOR

Table 1: Table of new tokens added, their regex rules, and example matches of the tokens

Token Type	Regex Rule	Example Matches
Keywords	^(if else while factorial repeat until int string)\$	else
Identifiers	^[a-zA-Z_][a-zA-Z0-9_]*\$	my_var, _x
String Literals	^"[~]*"\$	"hello"
Operators	See isOperator() function	+, -, *, /, ==, !=, <, >
Punctuation	;\{\}\(\),	{, }, (, ), ,, (comma ,)
Integer	^[0-9]+\$	42, 12345, 0
Float	^[0-9]+\.[0-9]+\$	10.0, 0.001

# Changes made to Intermediate Representation/ Code Generation Output

In phase 1 of this project, there were some minor changes made to the intermediate representation of the tokens produced by lexical analysis as well as their textual representation.

First, the modifications made to the data structures in the file `tokens.h`. The `Token` struct was modified and this modification is detailed in Figure 2. To accommodate the changes to the `Token` struct, a new struct, `LexemePosition`, was introduced. There were also changes made to the `TokenType` and `ErrorType` enums which can be seen in Figure 3 and Figure 4.

```
35     typedef struct _LexemePosition
36     {
37         int line;
38         int col_start;
39         int col_end;
40     } LexemePosition;
```

Figure 1: New struct introduced to record the position of a token's lexeme in the input.

```
39     typedef struct {
40         TokenType type;
41         char lexeme[100];
42         int line;
43         ErrorType error;
44     } Token;
```

```
43     typedef struct _Token
44     {
45         TokenType type;
46         char lexeme[100];
47         LexemePosition position;
48         ErrorType error;
49     } Token;
```

Figure 2: Comparison of changes to the `Token` struct. (left) Original. (right) Updated. The changes include introducing a struct `LexemePosition` to store the exact position of the token in the input, and modifications to the `TokenType` and `ErrorType` enums.

```
17     typedef enum {
18         TOKEN_EOF,
19         TOKEN_NUMBER,
20         TOKEN_OPERATOR,
21         TOKEN_ERROR
22     } TokenType;
```

```
11     typedef enum
12     {
13         TOKEN_EOF,
14         TOKEN_INTEGER,
15         TOKEN_FLOAT,
16         TOKEN_OPERATOR,
17         TOKEN_KEYWORD,
18         TOKEN_IDENTIFIER,
19         TOKEN_STRING_LITERAL,
20         TOKEN_PUNCTUATOR,
21         TOKEN_ERROR,
22     } TokenType;
```

Figure 3: Comparison of changes to the TokenType enum. (left) Original. (right) Updated. The new types added were to support tokens such as keywords, variable identifiers, string literals, and punctuation.

<pre>27     typedef enum { 28         ERROR_NONE, 29         ERROR_INVALID_CHAR, 30         ERROR_INVALID_NUMBER, 31         ERROR_CONSECUTIVE_OPERATORS 32     } ErrorType;</pre>	<pre>25     typedef enum 26     { 27         ERROR_NONE, 28         ERROR_INVALID_CHAR, 29         ERROR_INVALID_NUMBER, 30         ERROR_UNTERMINATED_STRING, 31         ERROR_STRING_TOO_LONG, 32         ERROR_UNTERMINATED_COMMENT, 33     } ErrorType;</pre>
--	---

Figure 4: Comparison of changes to the ErrorType enum. (left) Original. (right) Updated. The consecutive operators error was removed, two new errors related to string literals and an error for an unterminated comment was added.

Finally, the `print_token` function was updated accordingly to print the new lexeme position information now stored in the `Token` struct. The updated function is shown in Figure 5.

```
92 void print_token(Token token)
91 {
90     printf("Token type=%-10s, lexeme='%s', line=%-2d, column:%d-%d, error_message=\"%s\\n\"",
89         token_type_to_string(token.type), token.lexeme, token.position.line, token.position.
88         col_start, token.position.col_end, error_type_to_error_message(token.error));
}
```

Figure 5: `print_token` function code snippet.

## New Error Signals/Handling Mechanisms

Detection for invalid characters, unterminated strings, strings that are too long, and unterminated comments was added during this phase.

If none of the handlers for the various tokens take care of a character, then the character must be invalid. A token classified as an invalid character is created with the invalid character and the lexical analysis continues.

There are two errors associated with string literals, both of which are related to improper termination. Once a string literal has been identified, if a character that cannot be in a string literal is detected then the string is deemed unterminated. An example of this is if a starting quote and some text is followed by a newline. As the closing quote for the string literal would be expected before a newline is found, when a newline is detected in the described context it can be determined that the string literal is unterminated. This logic is applied to the detection of any nonprintable characters or the end of the file.

If while tokenizing the string literal, the size of the token exceeds the maximum size then then an error must be present. This scenario can occur if the string literal is either too long or unterminated. In this case the lexer continues reading tokens until finding the first instance of a nonprintable character, the end of the file, or the closing quote. If a nonprintable character or the end of file are found before the

closing quote then, by similar logic previously used, the string is unterminated. If a closing quote is found after the string literal has exceeded the maximum size, then the string literal is classified as too long.

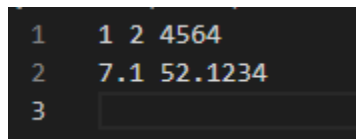
Just like when a string literal terminates properly, after a string literal is classified as too long or unterminated, the lexical analysis proceeds normally.

A multi-line comment must be terminated with '!?'. If while tokenizing a multi-line comment, the end of the file is reached before the end of the multi-line comment is reached then the comment must be unterminated. If a multi-line comment is unterminated then the contents of the file after the start of the comment is classified as part of the comment until the end of the file.

In the future detection for invalid numbers will be added to the lexical analysis phase.

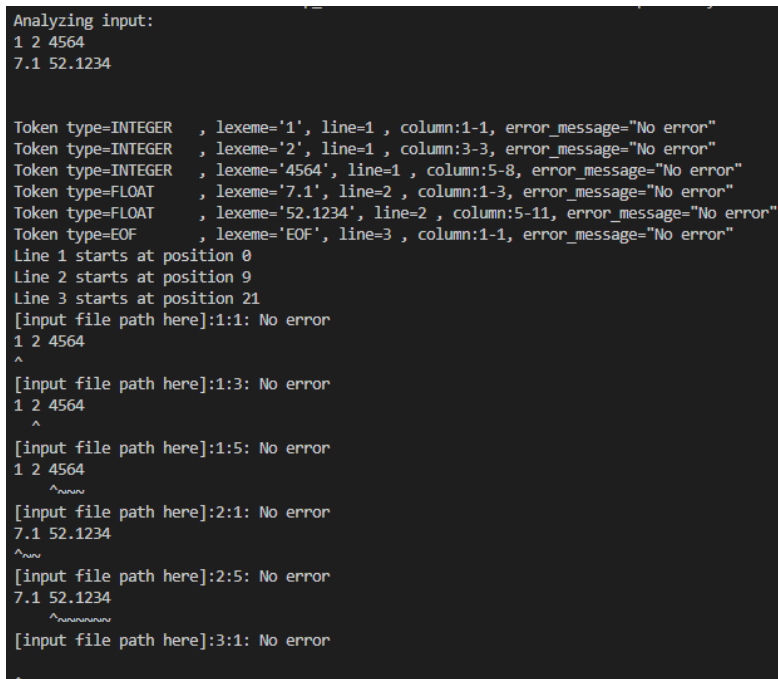
## Integers and Floats

The Figure 6 and Figure 7 below show the contents of the numbers test file, 'numbers.cisc', and the output of running lexer on the test file.



```
1 1 2 4564
2 7.1 52.1234
3
```

Figure 6: test case for numbers



```
Analyzing input:
1 2 4564
7.1 52.1234

Token type=INTEGER , lexeme='1', line=1 , column:1-1, error_message="No error"
Token type=INTEGER , lexeme='2', line=1 , column:3-3, error_message="No error"
Token type=INTEGER , lexeme='4564', line=1 , column:5-8, error_message="No error"
Token type=FLOAT , lexeme='7.1', line=2 , column:1-3, error_message="No error"
Token type=FLOAT , lexeme='52.1234', line=2 , column:5-11, error_message="No error"
Token type=EOF , lexeme='EOF', line=3 , column:1-1, error_message="No error"
Line 1 starts at position 0
Line 2 starts at position 9
Line 3 starts at position 21
[input file path here]:1:1: No error
1 2 4564
^
[input file path here]:1:3: No error
1 2 4564
^
[input file path here]:1:5: No error
1 2 4564
^
[input file path here]:2:1: No error
7.1 52.1234
^
[input file path here]:2:5: No error
7.1 52.1234
^
[input file path here]:3:1: No error
^
```

Figure 7: output of the test case for numbers

The output shows that numbers 1, 2, and 4564 were identified as integers and 7.1 and 52.1234 were identified as floats as expected. Counting the number characters on each line, it can be seen that lines 1,

2, and 3 do indeed start at positions 0, 9, and 21, just like the lexer stated. The lexer is also able to identify the end of the file. Both the starting position of lines and the detection of the end of the file can be seen in this and all subsequent tests.

## Operators

Figure 8 and Figure 9 show the test file, 'operators.cisc', and its output.

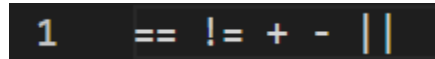


Figure 8: test case for operators

```
Analyzing input:
== != + - ||

Token type=OPERATOR , lexeme=='==', line=1 , column:1-1, error_message="No error"
Token type=OPERATOR , lexeme!='!=', line=1 , column:4-4, error_message="No error"
Token type=OPERATOR , lexeme='+', line=1 , column:7-7, error_message="No error"
Token type=OPERATOR , lexeme='-', line=1 , column:9-9, error_message="No error"
Token type=OPERATOR , lexeme='||', line=1 , column:11-11, error_message="No error"
Token type=EOF      , lexeme='EOF', line=1 , column:13-13, error_message="No error"
Line 1 starts at position 0
[input file path here]:1:1: No error
== != + - ||
^
[input file path here]:1:4: No error
== != + - ||
  ^
[input file path here]:1:7: No error
== != + - ||
    ^
[input file path here]:1:9: No error
== != + - ||
      ^
[input file path here]:1:11: No error
== != + - ||
        ^
[input file path here]:1:13: No error
== != + - ||
          ^
```

Figure 9: output of the test case for operators

All the tokens in the test are correctly identified as operators. As pictured above the lexer can recognize both single and multi-character operators.

## Punctuators and Invalid characters

The below Figure 10 and Figure 11 show the test case for punctuation and invalid characters, 'punctuators.cisc'.

```
1 print(.);
```

Figure 10: test case for punctuators and invalid characters

```
Analyzing input:
print(.);

Token type=IDENTIFIER, lexeme='print', line=1 , column:1-5, error_message="No error"
Token type=PUNCTUATOR, lexeme='(', line=1 , column:6-6, error_message="No error"
Token type=ERROR      , lexeme='.', line=1 , column:7-7, error_message="error: invalid character"
Token type=PUNCTUATOR, lexeme=')', line=1 , column:8-8, error_message="No error"
Token type=PUNCTUATOR, lexeme=';', line=1 , column:9-9, error_message="No error"
Token type=EOF        , lexeme='EOF', line=1 , column:10-10, error_message="No error"
Line 1 starts at position 0
[input file path here]:1:1: No error
print(.);
^~~~~~
[input file path here]:1:6: No error
print(.);
^
[input file path here]:1:7: error: invalid character
print(.);
^
[input file path here]:1:8: No error
print(.);
^
[input file path here]:1:9: No error
print(.);
^
[input file path here]:1:10: No error
print(.);
^
```

Figure 11: output of the test case for punctuators and invalid characters

The lexer correctly identifies the brackets and the semi-colon as punctuators and the period as an invalid character.

## Keywords and Identifiers

```
1 if while
2 _identifier an0th3r_identifier
3
```

Figure 12: test case for keywords and identifiers



```

Analyzing input:
if while
_identifier an0th3r_identifier

Token type=KEYWORD    , lexeme='if', line=1 , column:1-2, error_message="No error"
Token type=KEYWORD    , lexeme='while', line=1 , column:4-8, error_message="No error"
Token type=IDENTIFIER, lexeme='_identifier', line=2 , column:1-11, error_message="No error"
Token type=IDENTIFIER, lexeme='an0th3r_identifier', line=2 , column:13-30, error_message="No error"
Token type=EOF        , lexeme='EOF', line=3 , column:1-1, error_message="No error"
Line 1 starts at position 0
Line 2 starts at position 9
Line 3 starts at position 40
line_start_pos=0, line_length=8
.\phase1-w25\test\keywords&identifiers.cisc:1:1: No error
if while
^~
line_start_pos=0, line_length=8
.\phase1-w25\test\keywords&identifiers.cisc:1:4: No error
if while
^~~~~~
line_start_pos=9, line_length=30
.\phase1-w25\test\keywords&identifiers.cisc:2:1: No error
_identifier an0th3r_identifier
^~~~~~
line_start_pos=9, line_length=30
.\phase1-w25\test\keywords&identifiers.cisc:2:13: No error
_identifier an0th3r_identifier
^~~~~~
line_start_pos=40, line_length=0
.\phase1-w25\test\keywords&identifiers.cisc:3:1: No error
^

```

Figure 13: output of the test case for keywords and identifiers

As seen in Figure 12 and Figure 13 the keywords 'if' and 'while' and identifiers in the 'keywords&identifiers.cisc' test case are all correctly categorized. The line positioning and end of file are also still working properly.

## Strings

```

1  "this is a valid string literal"
2  "this string literal is unterminated because it ends in a new line
3  "this string literal is unterminated as well, but also exceeds the maximum size for a string literal. extra characters
4  "this string literal is too long, not unterminated, the program knows the difference because it can find the closing quote"
5

```

Figure 14: test case for strings

```

Analyzing input:
"this is a valid string literal"
"this string literal is unterminated because it ends in a new line"
"this string literal is unterminated as well, but also exceeds the maximum size for a string literal. extra characters"
"this string literal is too long, not unterminated, because the program can find the closing quote now"

Token type=STRING_LITERAL, lexeme="this is a valid string literal", line=1, column=1-32, error_message="No error"
Token type=ERROR, lexeme="this string literal is unterminated because it ends in a new line", line=2, column=1-66, error_message="error: unterminated string literal"
Token type=ERROR, lexeme="this string literal is unterminated as well, but also exceeds the maximum size for a string literal", line=3, column=1-98, error_message="error: unterminated string literal"
Token type=ERROR, lexeme="this string literal is too long, not unterminated, because the program can find the closing quote", line=4, column=1-98, error_message="error: string literal too long"
Token type=EOF, lexeme="EOF", line=4, column=104-104, error_message="No error"

Line 1 starts at position 0
Line 2 starts at position 33
Line 3 starts at position 100
Line 4 starts at position 219
line_start_pos=0, line_length=32
.phasel-w25(test\strings.cisc:1:1: No error
"this is a valid string literal"
~
line_start_pos=33, line_length=66
.phasel-w25(test\strings.cisc:2:1: error: unterminated string literal
"this string literal is unterminated because it ends in a new line
~
line_start_pos=100, line_length=118
.phasel-w25(test\strings.cisc:3:1: error: unterminated string literal
"this string literal is unterminated as well, but also exceeds the maximum size for a string literal. extra characters
~
line_start_pos=219, line_length=103
.phasel-w25(test\strings.cisc:4:1: error: string literal too long
"this string literal is too long, not unterminated, because the program can find the closing quote now"
~
line_start_pos=219, line_length=103
.phasel-w25(test\strings.cisc:4:104: No error
"this string literal is too long, not unterminated, because the program can find the closing quote now"
~

```

Figure 15: output of the test case for strings

The 'strings.cisc' test case in Figure 14 and Figure 15 first shows a proper string literal then all possible ways the string can return as an error. Firstly, the string can be unterminated. The string can also be too long, after identifying that the string is too long it can be unterminated or just too long. The test case shows these cases in the order in which they were described.

## Comments

The test case for comments, 'comments.cisc', is shown in Figure 16 and Figure 17.

```

1  ?? this is a single line comment
2  ?! this is a
3  multi line
4  comment!?
5  ?!this
6  is the same
7  thing but this time
8  it's unterminated

```

Figure 16: test case for comments

```

Analyzing input:
?? this is a single line comment
?! this is a
multi line
comment!?
?!this
is the same
thing but this time
it's unterminated

Token type=ERROR      , lexeme='', line=5 , column:1-1, error_message="error: unterminated comment, missing matching '!?' "
Token type=EOF        , lexeme='EOF', line=8 , column:18-18, error_message="No error"
Line 1 starts at position 0
Line 2 starts at position 33
Line 3 starts at position 47
Line 4 starts at position 58
Line 5 starts at position 68
Line 6 starts at position 75
Line 7 starts at position 87
Line 8 starts at position 107
line_start_pos=68, line_length=6
.\phase1-w25\test\comments.cisc:5:1: error: unterminated comment, missing matching '!?'
?!this
^
line_start_pos=107, line_length=17
.\phase1-w25\test\comments.cisc:8:18: No error
it's unterminated
^

```

*Figure 17: output of the test case for comments*

Both the single line and multi-line comment are correctly ignored, and the unterminated multi-line comment is detected and reported. Once again, the line positioning and end of file detection is still working properly.