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 6th, 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.

A screen shot of a computer code

Description automatically generated

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

|  |  |
| --- | --- |
| A computer code with text  Description automatically generated with medium confidence | A screen shot of a computer code  Description automatically generated |

Figure : 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.

|  |  |
| --- | --- |
|  |  |

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

|  |  |
| --- | --- |
|  |  |

Figure : 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.

A computer code on a black background

Description automatically generated

Figure : 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 Figures 6 and 7below show the contents of the numbers test file, ‘numbers.cisc’, and the output of running lexer on the test file.

A screenshot of a computer

Description automatically generated

Figure 6: test case for numbers

A screenshot of a computer

Description automatically generated

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

Figures 8 and 9 show the test file, ‘operators.cisc’, and its output.



Figure 8: test case for operators

A screenshot of a computer error

Description automatically generated

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 Figures 10and 11show the test case for punctuation and invalid characters, ‘punctuators.cisc’.



Figure 10: test case for punctuators and invalid characters

A computer screen shot of a program

Description automatically generated

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

A black background with white text

Description automatically generated

Figure 12: test case for keywords and identifiers

A screenshot of a computer program

Description automatically generated

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

As seen in Figures 12 and 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

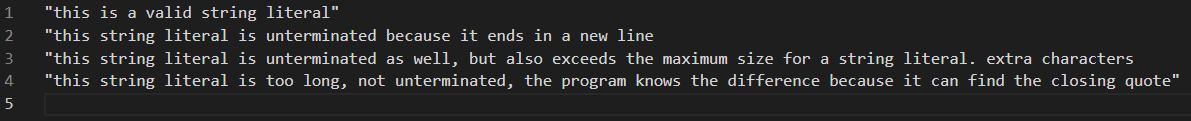


Figure 14: test case for strings

A black screen with many small colored lines

Description automatically generated with medium confidence

Figure 15: output of the test case for strings

The ‘strings.cisc’ test case in Figures 14 and 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 16and 17.

A screenshot of a computer program

Description automatically generated

Figure 16: test case for comments

A computer screen shot of a black screen

Description automatically generated

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.