Indian Institute of Technology Gandhinagar

Lexer for ***C*--**

Compilers Assignment 2

- Group 6
- Reuben Devanesan, 19110059
- Kamal Vaishnav, 20110089
- Preetam Chhimpa, 20110145
- Rishab Jain, 20110164
- GitHub Repository: https://github.com/IITGN-CS327-2024/our-own-compiler-com-piler-t6

Contents

Introduction

CMinusMinus (C--) is a programming language designed to offer a simplified syntax and semantics aimed at educational purposes and ease of understanding for beginners. It uses the ***.cmm*** file extension and is case-sensitive, distinguishing between uppercase and lowercase letters in variable names, keywords, and other identifiers. The language is structured to start program execution from the first line of the script, similar to Python, making it straightforward for new programmers to grasp the flow of control in programs.

Token Classes Explanation and Listings

1) Keywords

- Keywords in C-- are reserved words that have a special meaning in the language.
- They are used to define the syntax and structure of the program, such as declaring variables, specifying data types, controlling flow with conditional statements, and more.
- Keywords cannot be used as identifiers for variables or functions.
- All C-- keywords are lowercase. Following are the list of all keywords:

```
keywords = { 'pendown', 'penup', 'integer', 'character', 'floatpoint', 'doublepoint', 'textwave', 'flag', 'true', 'false', 'given', 'elsegiven', 'otherwise', 'iterate', 'through', 'fn', 'exitwith', 'strive', 'capture', 'immute', 'interrupt', 'resume', 'dim', 'sort','while', 'add', 'remove', 'exchange', 'getidx', 'asc', 'dsc','tail', 'head', ‘range’, ‘throwexception’ }
```

2) Identifiers

- Identifiers are names used to identify variables, functions, and other entities in a program. They must start with a letter (a-z, A-Z) or an underscore (_), followed by letters, digits (0-9), underscores, or dollar signs ($).
- Following is the regex for recognizing identifiers of C--:

```
r'\b[a-zA-Z_][a-zA-Z_0-9]*\b'
```

3) Literals

- Literals represent fixed values assigned directly in the code. They can be integers, floating-point numbers, strings, or boolean values.
  - Integer Literals: Represent whole numbers.
  - Float Literals: Represent numbers with fractional parts.
  - String Literals: Represent sequences of characters enclosed in double-quotes.
  - Boolean Literals: Represent true or false values.

4) Operators

- Operators are symbols that perform operations on operands. In C--, operators are divided into unary and binary types.

a) Unary Operators: Unary operators operate on a single operand.

```
unary_operators = { '++', '--' }
```

b) Binary Operators: Binary operators take two operands and perform a variety of arithmetic, relational, or logical operations.

```
binary_operators = { ‘+', '-', '*', '/', '%', '^', '==', '!=', '>', '<',
'>=', '<=', '=', '+=', '-=', '*=', '/=', '%=', '&&', '$$', '!' }
```

c) Whitespace Characters: It includes spaces, tabs, and newlines. Used to separate tokens in the source code for readability but do not have a meaning by themselves.

```
whitespace = { ' ', '\n', '\t' }
```

d) Parentheses and Delimiters: Parentheses and delimiters are used to organize code, indicate the beginning and end of blocks, separate parameters in function calls, etc.

- Parentheses are specifically used for grouping and function calls, while delimiters serve a broader range of uses, including terminating statements and separating elements in lists.
- Delimiters are used to separate tokens and include characters like semicolons, commas, and others that are not specifically used for grouping.

```
parenthesis = { '(', ')', '[', ']', '{', '}' }
delimiters = { ';', ',', '.',':' }
```

By categorizing tokens as described and using the associated regular expressions, the lexer for C-- can accurately parse and tokenize source code, effectively preparing it for the subsequent stages of compilation.

Regular Expressions for Token Identification

Regular expressions (Regex) are used to define patterns for identifying the various tokens in C-- code. The following are the key rules and the corresponding Regex patterns used in the lexer for C--:

- **Maximal Munch Rule:** This rule dictates that when choosing among multiple matching patterns, the lexer should select the one that consumes the longest portion of the input string. This approach maximizes the amount of input processed in each step and helps in reducing ambiguities.
- **Priority Ordering Rule:** In cases where two or more patterns match an equal length of the input string, the lexer resolves the ambiguity by prioritizing the pattern that appears first in the set of lexical specifications. This ordering principle ensures a deterministic and predictable tokenization process.
- **No Rule Match Handling:** For any substring of the input that does not match any defined pattern or rule, the lexer classifies it as an error. This is managed by introducing an ERROR token, which encapsulates strings not recognized by the lexical specifications of C--. This rule is crucial for robust error detection and reporting, enhancing the language's fault tolerance and developer feedback mechanisms.

By implementing these rules, the lexer for C-- efficiently transforms source code into a series of tokens, laying the groundwork for the subsequent phases of compilation with clarity and precision.

Regex for Tokens

1) Comments: Ignored by the lexer, identified by # for single-line and triple quotes """ """ for multi-line comments.

a) **Single-line Comment**

```
r'#.*'
```

b) **Multi-line Comment**

```
r'"""[\s\S]*?"""'
```

2) Keywords: Defined by specific words that are reserved for language syntax.

```
r'\b(pendown|penup|integer|character|floatpoint|doublepoint|textwave|flag|True|
False|given|elsegiven|otherwise|iterate|through|fn|exitwith|strive|capture|immu
te|interrupt|resume|dim|sort|range|throwexeption)\b'
```

3) Operators: Include arithmetic, comparison, assignment, and logical operators.

a) **Unary Operators**

```
r'(\+\+|--)'
```

b) **Binary Operators**

```
r'(\+|\-|\*|\/|%|\^|==|!=|>|<|>=|<=|=|\+=|\-=|\*=|\/=|%=|&&|\$\$|!)'
```

4) Literals: Include integer, floating-point numbers, strings, and booleans.

a) **Integer**

```
r'-?\b\d+\b'
```

b) **Float**

```
r'-?\b\d+\.\d*\b|-?\b\d+\.\b'
```

c) **String**

```
r'"(?:\\.|[^"\\])*"'
```

5) Identifiers: Names for variables and functions.

```
r'\b[a-zA-Z_][a-zA-Z_0-9]*\b'
```

Test Cases and Results

To validate the lexer, seven test cases were crafted involving all tokens. These test cases include variable declarations, arithmetic operations, conditional statements, loops, and function definitions. The lexer's output correctly identified and classified each token, demonstrating the lexer's ability to parse and tokenize C-- source code effectively.

Test Case 1

```
# This is a single line comment

"""
This is a multi-line comment
spanning multiple lines.
"""

myInteger (integer) = 10;
pendown("Hello, World!");

# Testing character and boolean literals
myChar (character) = 'c';
myBool (flag) = true;
myfloat (floatpoint) = 1.2;
mydouble (doublepoint) = 1.2345;
```

Test Case 1 Output

```
<identifier, "myInteger">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "10">
<delimiters, ";">
<keywords, "pendown">
<parentheses, "(">
<string, "Hello, World!">
<parentheses, ")">
<delimiters, ";">
<identifier, "myChar">
<parentheses, "(">
<keywords, "character">
<parentheses, ")">
<binary_operators, "=">
```

```
<char_literal, "'c'">
<delimiters, ";">
<identifier, "myBool">
<parentheses, "(">
<keywords, "flag">
<parentheses, ")">
<binary_operators, "=">
<keywords, "true">
<delimiters, ";">
<identifier, "myfloat">
<parentheses, "(">
<keywords, "floatpoint">
<parentheses, ")">
<binary_operators, "=">
<float_literal, "1.2">
<delimiters, ";">
<identifier, "mydouble">
<parentheses, "(">
<keywords, "doublepoint">
<parentheses, ")">
<binary_operators, "=">
<float_literal, "1.2345">
<delimiters, ";">
```

Test Case 2

```
myNum1 (integer) = 5;
myNum2 (integer) = 10;
sum (integer) = myNum1 + myNum2;
diff (integer) = myNum2 - myNum1;
product (integer) = myNum1 * myNum2;
quotient (floatpoint) = myNum2 / myNum1;

myIncrement (integer) = 0;
myIncrement++;
myDecrement (integer) = 10;
myDecrement--;
```

Test Case 2 Output

```
<identifier, "myNum1">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "5">
<delimiters, ";">
<identifier, "myNum2">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "10">
<delimiters, ";">
<identifier, "sum">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<identifier, "myNum1">
<binary_operators, "+">
<identifier, "myNum2">
<delimiters, ";">
<identifier, "diff">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<identifier, "myNum2">
```

```
<binary_operators, "-">
<identifier, "myNum1">
<delimiters, ";">
<identifier, "product">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<identifier, "myNum1">
<identifier, "myNum2">
<delimiters, ";">
<identifier, "quotient">
<parentheses, "(">
<keywords, "floatpoint">
<parentheses, ")">
<binary_operators, "=">
<identifier, "myNum2">
<identifier, "myNum1">
<delimiters, ";">
<identifier, "myIncrement">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "0">
<delimiters, ";">
<identifier, "myIncrement">
<unary_operators, "++">
<delimiters, ";">
<identifier, "myDecrement">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "10">
<delimiters, ";">
<identifier, "myDecrement">
<unary_operators, "--">
<delimiters, ";">
```

Test Case 3

```
x (integer) = 20;
y (integer) = 30;

given (x < y && y > 10) {
    pendown("x is less than y and y is greater than 10");
}
elsegiven (x == y $$ x != 10) {
    pendown("x is equal to y or x is not 10");
}
otherwise {
    pendown("None of the conditions are true");
}
```

Test Case 3 Output

```
<identifier, "x">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "20">
<delimiters, ";">
<identifier, "y">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "30">
<delimiters, ";">
<keywords, "given">
<parentheses, "(">
<identifier, "x">
<binary_operators, "<">
<identifier, "y">
<binary_operators, "&&">
<identifier, "y">
<binary_operators, ">">
<integer_literal, "10">
<parentheses, ")">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<string, "x is less than y and y is greater than 10">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
<keywords, "elsegiven">
<parentheses, "(">
<identifier, "x">
<binary_operators, "==">
<identifier, "y">
<binary_operators, "$$">
<identifier, "x">
<binary_operators, "!=">
<integer_literal, "10">
<parentheses, ")">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<string, "x is equal to y or x is not 10">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
<keywords, "otherwise">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<string, "None of the conditions are true">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
```

Test Case 4

```
fn sum(a (integer), b(integer)) {
    result (integer) = a + b;
    exitwith result;
}

iterate (i (integer) = 0; through range(0.1 ... 5.2, 0.1);) {
    pendown(i);
}
```

Test Case 4 Output

```
<keywords, "fn">
<identifier, "sum">
<parentheses, "(">
<identifier, "a">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<delimiters, ",">
<identifier, "b">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<parentheses, ")">
<parentheses, "{">
<identifier, "result">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<identifier, "a">
<binary_operators, "+">
<identifier, "b">
<delimiters, ";">
<keywords, "exitwith">
<identifier, "result">
<delimiters, ";">
<parentheses, "}">
<keywords, "iterate">
<parentheses, "(">
<identifier, "i">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "0">
<delimiters, ";">
<keywords, "through">
<keywords, "range">
<parentheses, "(">
<float_literal, "0.1">
<range_delimiter, "...">
<float_literal, "5.2">
<delimiters, ",">
<float_literal, "0.1">
<parentheses, ")">
<delimiters, ";">
<parentheses, ")">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<identifier, "i">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
```

Test Case 5

```
str1 (textwave) = "Hello";
str2 (textwave) = "World";
combinedStr (textwave) = str1 + ", " + str2 + "!";

pendown(combinedStr); # Should print "Hello, World!"

# Multiline comment with code inside (should be ignored)
"""
iterate (i (integer) = 0; through range(0...5, 1);) {
    pendown(i);
}
"""


myFloat (floatpoint) = 3.14;
myDouble (doublepoint) = 3.1415926535;
```

Test Case 5 Output

```
<identifier, "str1">
<parentheses, "(">
<keywords, "textwave">
<parentheses, ")">
<binary_operators, "=">
<string, "Hello">
<delimiters, ";">
<identifier, "str2">
<parentheses, "(">
<keywords, "textwave">
<parentheses, ")">
<binary_operators, "=">
<string, "World">
<delimiters, ";">
<identifier, "combinedStr">
<parentheses, "(">
<keywords, "textwave">
<parentheses, ")">
<binary_operators, "=">
<identifier, "str1">
<binary_operators, "+">
<string, ", ">
<binary_operators, "+">
<identifier, "str2">
<binary_operators, "+">
```

```
<string, "!">
<delimiters, ";">
<keywords, "pendown">
<parentheses, "(">
<identifier, "combinedStr">
<parentheses, ")">
<delimiters, ";">
<identifier, "myFloat">
<parentheses, "(">
<keywords, "floatpoint">
<parentheses, ")">
<binary_operators, "=">
<float_literal, "3.14">
<delimiters, ";">
<identifier, "myDouble">
<parentheses, "(">
<keywords, "doublepoint">
<parentheses, ")">
<binary_operators, "=">
<float_literal, "3.1415926535">
<delimiters, ";">
```

Test Case 6

```
strive() {
    myValue (integer) = 10;
    given (myValue > 5) {
        throwexception("Value exceeds threshold");
    }
}
capture() {
    pendown("Error has occurred: Value exceeds threshold");
}

strive() {
    mySafeValue (integer) = 4;
    pendown("This operation is safe");
}
capture() {
    pendown("An unexpected error has occurred in a safe operation");
}
```

Test Case 6 Output

```
<keywords, "strive">
<parentheses, "(">
<parentheses, ")">
<parentheses, "{">
<identifier, "myValue">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "10">
<delimiters, ";">
<keywords, "given">
<parentheses, "(">
<identifier, "myValue">
<binary_operators, ">">
<integer_literal, "5">
<parentheses, ")">
<parentheses, "{">
<keywords, "throwexception">
<parentheses, "(">
<string, "Value exceeds threshold">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
<parentheses, "}">
<keywords, "capture">
<parentheses, "(">
<parentheses, ")">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<string, "Error has occurred: Value exceeds threshold">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
<keywords, "strive">
<parentheses, "(">
<parentheses, ")">
<parentheses, "{">
<identifier, "mySafeValue">
<parentheses, "(">
<keywords, "integer">
<parentheses, ")">
<binary_operators, "=">
<integer_literal, "4">
<delimiters, ";">
<keywords, "pendown">
<parentheses, "(">
<string, "This operation is safe">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
<keywords, "capture">
<parentheses, "(">
<parentheses, ")">
<parentheses, "{">
<keywords, "pendown">
<parentheses, "(">
<string, "An unexpected error has occurred in a safe operation">
<parentheses, ")">
<delimiters, ";">
<parentheses, "}">
```

Test Case 7

```
pendown penup integer character floatpoint doublepoint textwave flag true false
given elsegiven otherwise iterate through fn exitwith strive capture immute
interrupt resume dim sort while add remove exchange getidx asc dsc tail head range
throwexception
+ - * / % ^ == != > < >= <= = += -= *= /= %= && $$ !
++ --
( ) [ ] { }
; , . :
...
..
```

Test Case 7 Output

```
<keywords, "pendown">
<keywords, "penup">
<keywords, "integer">
<keywords, "character">
<keywords, "floatpoint">
<keywords, "doublepoint">
<keywords, "textwave">
<keywords, "flag">
<keywords, "true">
<keywords, "false">
<keywords, "given">
<keywords, "elsegiven">
<keywords, "otherwise">
<keywords, "iterate">
<keywords, "through">
<keywords, "fn">
<keywords, "exitwith">
<keywords, "strive">
<keywords, "capture">
<keywords, "immute">
<keywords, "interrupt">
<keywords, "resume">
<keywords, "dim">
<keywords, "sort">
<keywords, "while">
<keywords, "add">
<keywords, "remove">
<keywords, "exchange">
<keywords, "getidx">
<keywords, "asc">
```

```
<keywords, "dsc">
<keywords, "tail">
<keywords, "head">
<keywords, "range">
<keywords, "throwexception">
<binary_operators, "+">
<binary_operators, "-">
<binary_operators, "%">
<binary_operators, "^">
<binary_operators, "==">
<binary_operators, "!=">
<binary_operators, ">">
<binary_operators, "<">
<binary_operators, ">">
<binary_operators, "=">
<binary_operators, "<">
<binary_operators, "=">
<binary_operators, "=">
<binary_operators, "+">
<binary_operators, "=">
<binary_operators, "-">
<binary_operators, "=">
<binary_operators, "=">
<binary_operators, "/=">
<binary_operators, "%">
<binary_operators, "=">
<binary_operators, "&&">
<binary_operators, "$$">
<binary_operators, "!">
<unary_operators, "++">
<unary_operators, "--">
<parentheses, "(">
<parentheses, ")">
<parentheses, "[">
<parentheses, "]">
<parentheses, "{">
<parentheses, "}">
<delimiters, ";">
<delimiters, ",">
<delimiters, ".">
<delimiters, ":">
<range_delimiter, "...">
<delimiters, ".">
<delimiters, ".">
```

Conclusion

The lexer implementation for **CMinusMinus (C--)** demonstrates a robust foundation for parsing and tokenizing source code written in the C-- programming language. Through the use of regular expressions for token identification, the lexer efficiently categorizes and prepares the source code for further processing stages in the compiler. Our assignment showcases the flexibility and power of regular expressions in compiler design, enabling precise lexical analysis essential for language processing. The test cases and their results affirm the lexer's capability to handle a variety of language constructs.