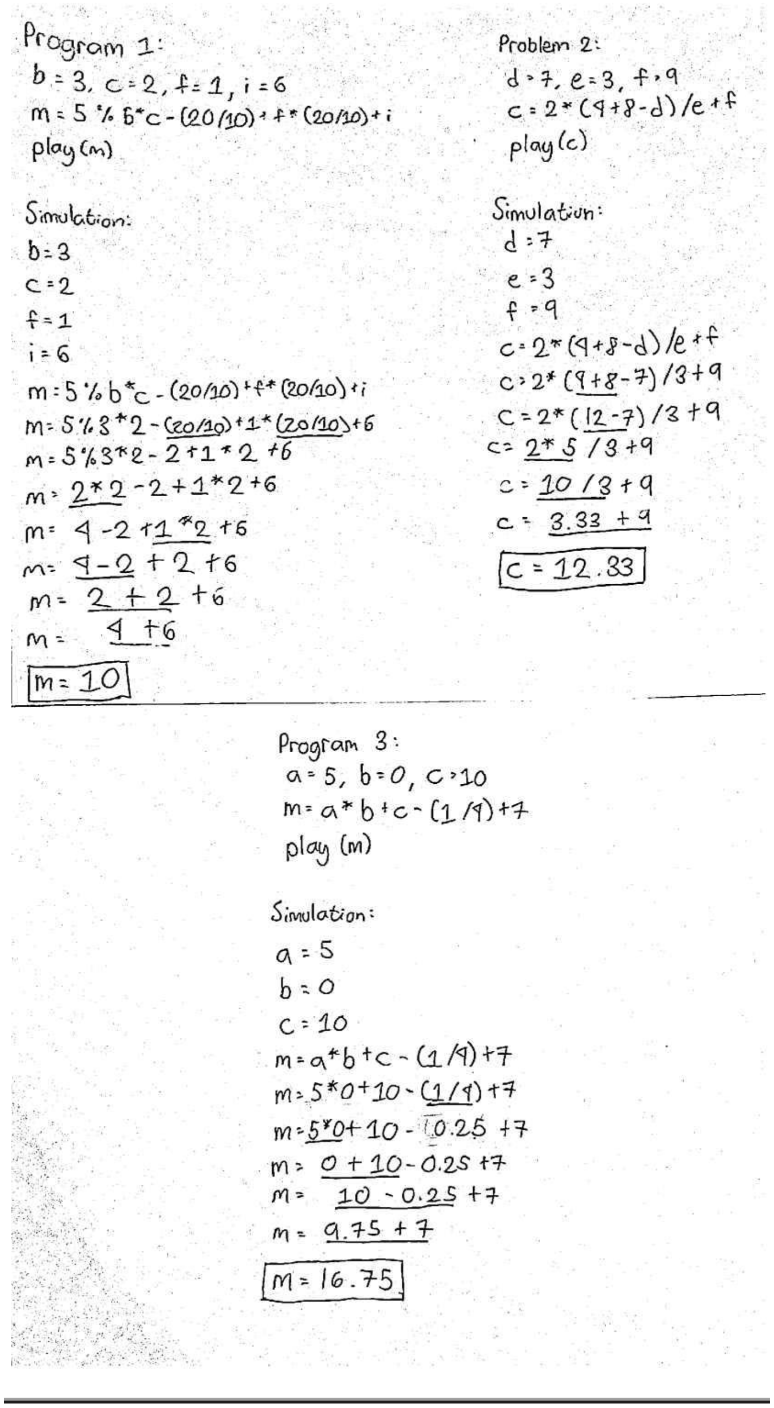
Ma. Corazon C. Macaraig
BSCS 3 – 2
2021-10699
DBTK

1. Simulate the expressions using your code (hand written).



Program 1:
b = 3, c = 2, f = 1, i = 6
m = 5 % b*c - (20/10) + f*(20/10) + i
play(m)

Simulation:
b = 3
c = 2
f = 1
i = 6
m = 5 % b*c - (20/10) + f*(20/10) + i
m = 5 % 3*2 - (20/10) + 1*(20/10) + 6
m = 5 % 3*2 - 2 + 1*2 + 6
m = 2*2 - 2 + 1*2 + 6
m = 4 - 2 + 1*2 + 6
m = 4 - 2 + 2 + 6
m = 2 + 2 + 6
m = 4 + 6
$\boxed{m = 10}$

Problem 2:
d = 7, e = 3, f = 9
c = 2*(9+8-d)/e + f
play(c)

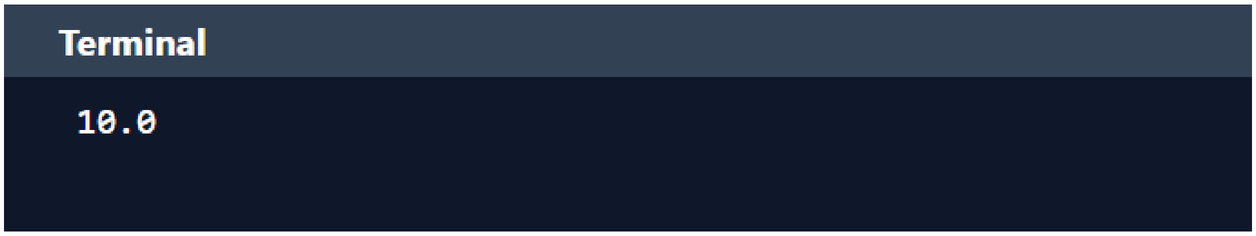
Simulation:
d = 7
e = 3
f = 9
c = 2*(9+8-d)/e + f
c = 2*(9+8-7)/3 + 9
c = 2*(12-7)/3 + 9
c = 2*5/3 + 9
c = 10/3 + 9
c = 3.33 + 9
$\boxed{c = 12.33}$

Program 3:
a = 5, b = 0, c = 10
m = a*b + c - (1/4) + 7
play (m)

Simulation:
a = 5
b = 0
c = 10
m = a*b + c - (1/4) + 7
m = 5*0 + 10 - (1/4) + 7
m = 5*0 + 10 - 0.25 + 7
m = 0 + 10 - 0.25 + 7
m = 10 - 0.25 + 7
m = 9.75 + 7
$\boxed{m = 16.75}$

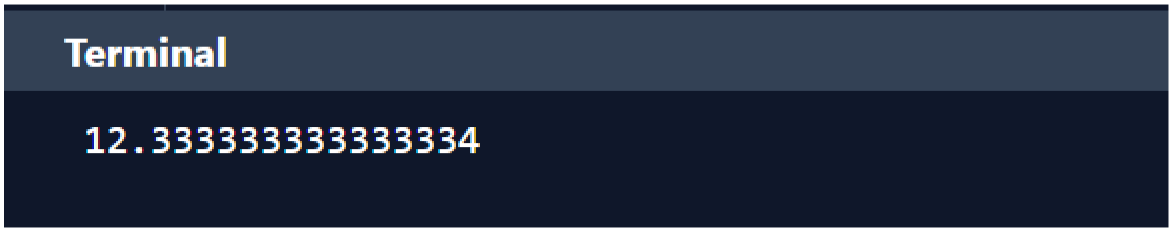
2. Run your compiler and display the output to validate your simulation.

Program 1:



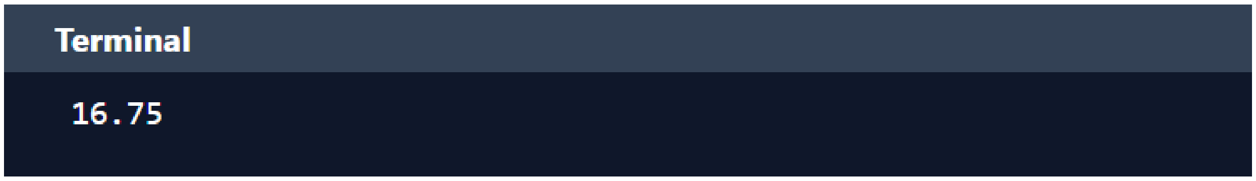
```
Terminal
10.0
```

Program 2:



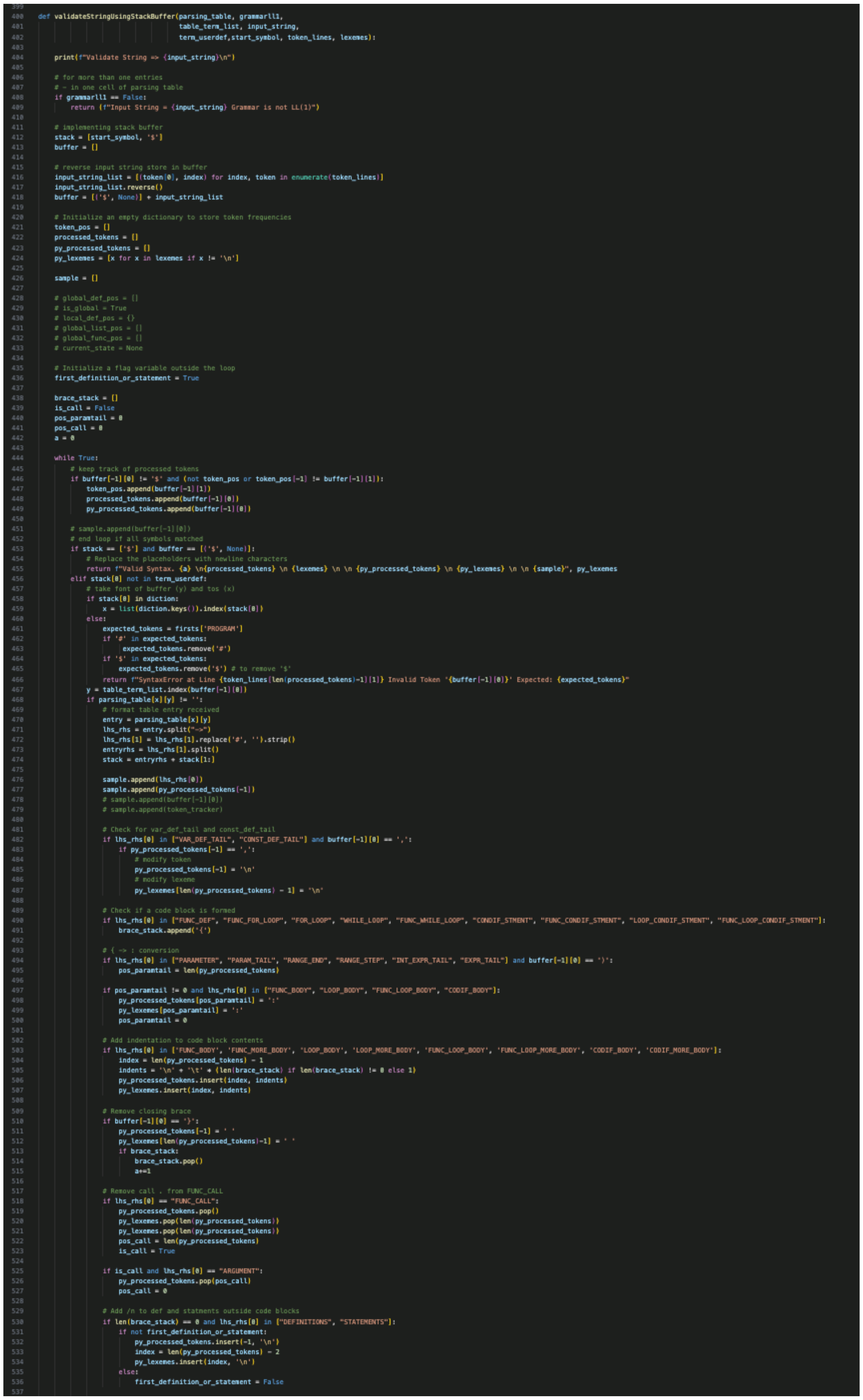
```
Terminal
12.333333333333334
```

Program 3:



```
Terminal
16.75
```

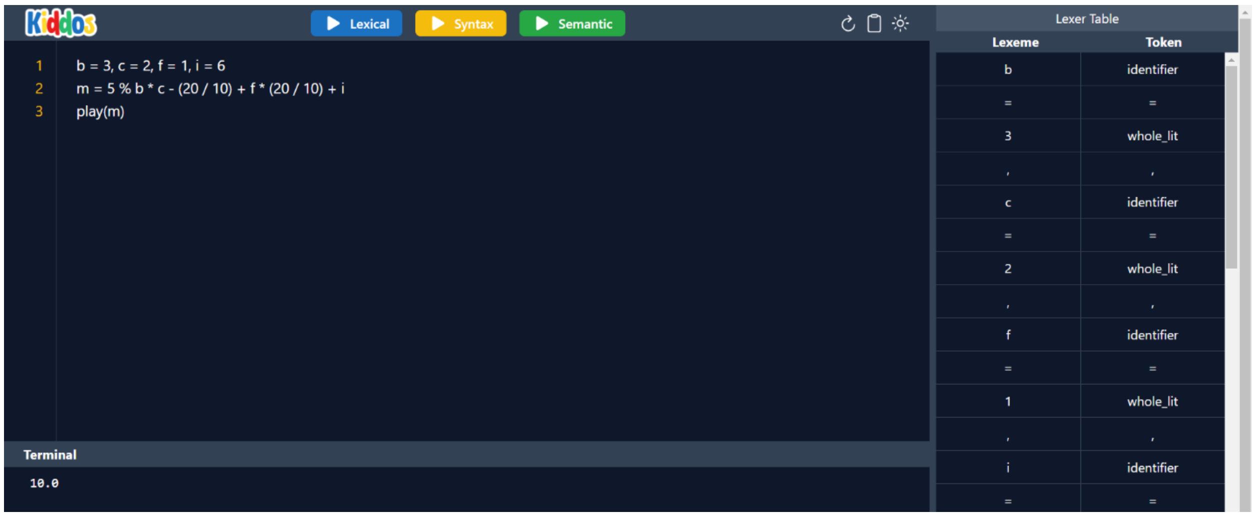
3. Submit the screenshot of your code.



```python

def validateStringUsingStackBuffer(parsing_table, grammarll1,
                                   table_term_list, input_string,
                                   term_userdef,start_symbol, token_lines, lexemes):

    print(f"Validate String => {input_string}\n")

    # for more than one entries
    # - in one cell of parsing table
    if grammarll1 == False:
        return (f"Input String = {input_string} Grammar is not LL(1)")

    # implementing stack buffer
    stack = [start_symbol, '$']
    buffer = []

    # reverse input string store in buffer
    input_string_list = [(token[0], index) for index, token in enumerate(token_lines)]
    input_string_list.reverse()
    buffer = [('$', None)] + input_string_list

    # Initialize an empty dictionary to store token frequencies
    token_pos = []
    processed_tokens = []
    py_processed_tokens = []
    py_lexemes = [x for x in lexemes if x != '\n']

    sample = []

    # global_def_pos = []
    # is_global = True
    # local_def_pos = {}
    # global_list_pos = []
    # global_func_pos = []
    # current_state = None

    # Initialize a flag variable outside the loop
    first_definition_or_statement = True

    brace_stack = []
    is_call = False
    pos_paramtail = 0
    pos_call = 0
    a = 0

    while True:
        # keep track of processed tokens
        if buffer[-1][0] != '$' and (not token_pos or token_pos[-1] != buffer[-1][1]):
            token_pos.append(buffer[-1][1])
            processed_tokens.append(buffer[-1][0])
            py_processed_tokens.append(buffer[-1][0])

        # sample.append(buffer[-1][0])
        # end loop if all symbols matched
        if stack == ['$'] and buffer == [('$', None)]:
            # Replace the placeholders with newline characters
            return f"Valid Syntax. {a} \n{processed_tokens} \n {lexemes} \n \n {py_processed_tokens} \n {py_lexemes} \n \n {sample}", py_lexemes
        elif stack[0] not in term_userdef:
            # take font of buffer (y) and tos (x)
            if stack[0] in diction:
                x = list(diction.keys()).index(stack[0])
            else:
                expected_tokens = firsts['PROGRAM']
                if '#' in expected_tokens:
                     expected_tokens.remove('#')
                if '$' in expected_tokens:
                    expected_tokens.remove('$') # to remove '$'
                return f"SyntaxError at Line {token_lines[len(processed_tokens)-1][1]} Invalid Token '{buffer[-1][0]}' Expected: {expected_tokens}"
            y = table_term_list.index(buffer[-1][0])
            if parsing_table[x][y] != '':
                # format table entry received
                entry = parsing_table[x][y]
                lhs_rhs = entry.split("->")
                lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
                entryrhs = lhs_rhs[1].split()
                stack = entryrhs + stack[1:]

                sample.append(lhs_rhs[0])
                sample.append(py_processed_tokens[-1])
                # sample.append(buffer[-1][0])
                # sample.append(token_tracker)

                # Check for var_def_tail and const_def_tail
                if lhs_rhs[0] in ["VAR_DEF_TAIL", "CONST_DEF_TAIL"] and buffer[-1][0] == ',':
                    if py_processed_tokens[-1] == ',':
                        # modify token
                        py_processed_tokens[-1] = '\n'
                        # modify lexeme
                        py_lexemes[len(py_processed_tokens) - 1] = '\n'

                # Check if a code block is formed
                if lhs_rhs[0] in ["FUNC_DEF", "FUNC_FOR_LOOP", "FOR_LOOP", "WHILE_LOOP", "FUNC_WHILE_LOOP", "CONDIF_STMENT", "FUNC_CONDIF_STMENT", "LOOP_CONDIF_STMENT", "FUNC_LOOP_CONDIF_STMENT"]:
                    brace_stack.append('{')

                # { -> : conversion
                if lhs_rhs[0] in ["PARAMETER", "PARAM_TAIL", "RANGE_END", "RANGE_STEP", "INT_EXPR_TAIL", "EXPR_TAIL"] and buffer[-1][0] == ')':
                    pos_paramtail = len(py_processed_tokens)

                if pos_paramtail != 0 and lhs_rhs[0] in ["FUNC_BODY", "LOOP_BODY", "FUNC_LOOP_BODY", "CODIF_BODY"]:
                    py_processed_tokens[pos_paramtail] = ':'
                    py_lexemes[pos_paramtail] = ':'
                    pos_paramtail = 0

                # Add indentation to code block contents
                if lhs_rhs[0] in ['FUNC_BODY', 'FUNC_MORE_BODY', 'LOOP_BODY', 'LOOP_MORE_BODY', 'FUNC_LOOP_BODY', 'FUNC_LOOP_MORE_BODY', 'CODIF_BODY', 'CODIF_MORE_BODY']:
                    index = len(py_processed_tokens) - 1
                    indents = '\n' + '\t' * (len(brace_stack) if len(brace_stack) != 0 else 1)
                    py_processed_tokens.insert(index, indents)
                    py_lexemes.insert(index, indents)

                # Remove closing brace
                if buffer[-1][0] == '}':
                    py_processed_tokens[-1] = ' '
                    py_lexemes[len(py_processed_tokens)-1] = ' '
                    if brace_stack:
                        brace_stack.pop()
                        a+=1

                # Remove call . from FUNC_CALL
                if lhs_rhs[0] == "FUNC_CALL":
                    py_processed_tokens.pop()
                    py_lexemes.pop(len(py_processed_tokens))
                    py_lexemes.pop(len(py_processed_tokens))
                    pos_call = len(py_processed_tokens)
                    is_call = True

                if is_call and lhs_rhs[0] == "ARGUMENT":
                    py_processed_tokens.pop(pos_call)
                    pos_call = 0

                # Add /n to def and statments outside code blocks
                if len(brace_stack) == 0 and lhs_rhs[0] in ["DEFINITIONS", "STATEMENTS"]:
                    if not first_definition_or_statement:
                        py_processed_tokens.insert(-1, '\n')
                        index = len(py_processed_tokens) - 2
                        py_lexemes.insert(index, '\n')
                    else:
                        first_definition_or_statement = False

```

4. Submit the screenshot of your output after running your compiler.

Program 1:



```
b = 3, c = 2, f = 1, i = 6
m = 5 % b * c - (20 / 10) + f * (20 / 10) + i
play(m)
```

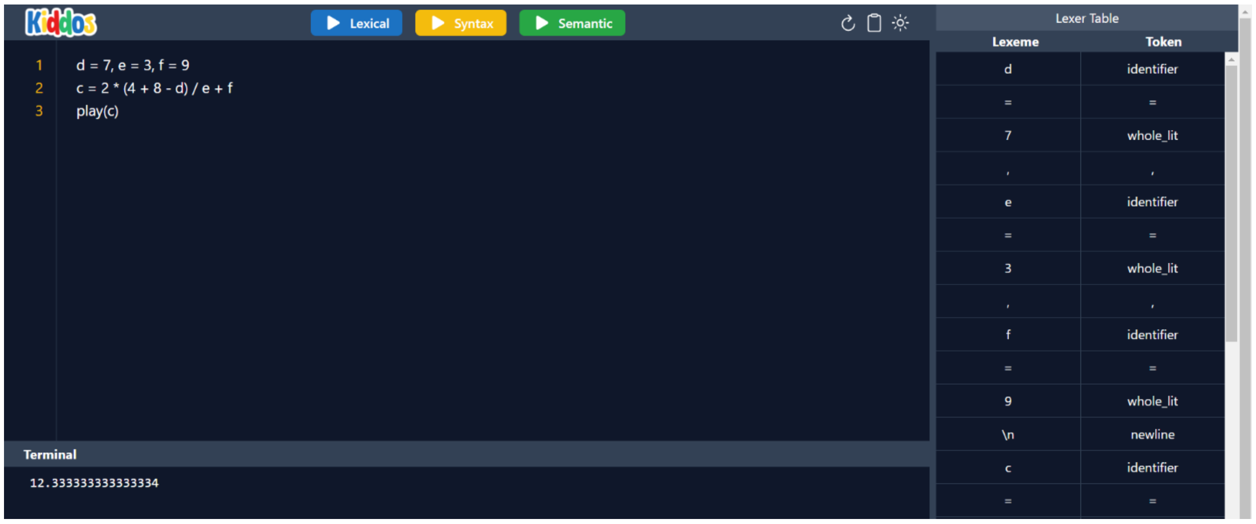
Terminal

```
10.0
```

Lexer Table

Lexeme	Token
b	identifier
=	=
3	whole_lit
,	,
c	identifier
=	=
2	whole_lit
,	,
f	identifier
=	=
1	whole_lit
,	,
i	identifier
=	=

Program 2:



```
d = 7, e = 3, f = 9
c = 2 * (4 + 8 - d) / e + f
play(c)
```

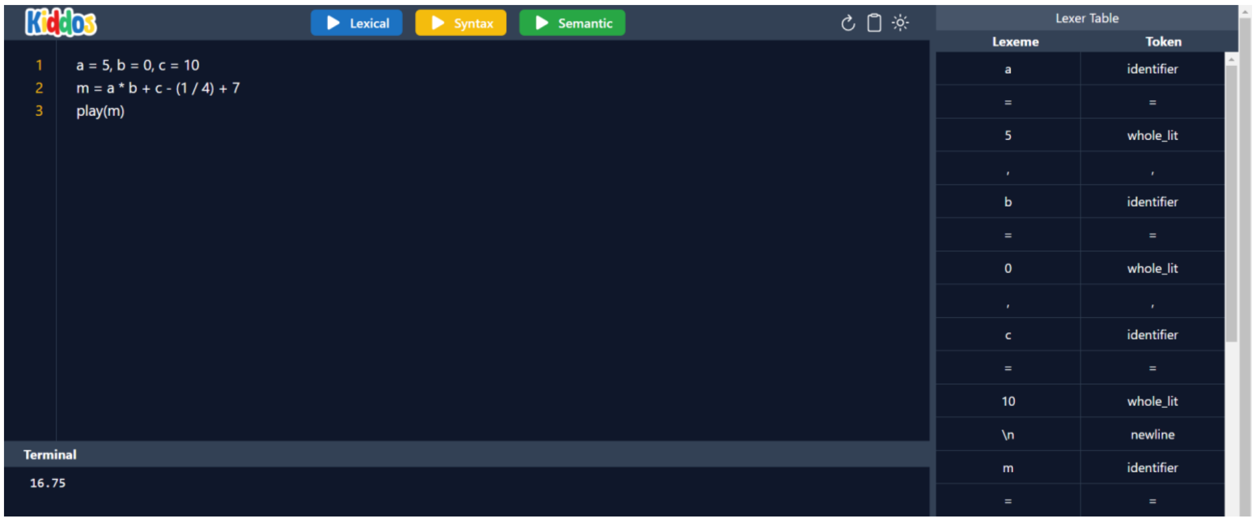
Terminal

```
12.333333333333334
```

Lexer Table

Lexeme	Token
d	identifier
=	=
7	whole_lit
,	,
e	identifier
=	=
3	whole_lit
,	,
f	identifier
=	=
9	whole_lit
\n	newline
c	identifier
=	=

Program 3:



```
a = 5, b = 0, c = 10
m = a * b + c - (1 / 4) + 7
play(m)
```

```
16.75
```

Lexeme	Token
a	identifier
=	=
5	whole_lit
,	,
b	identifier
=	=
0	whole_lit
,	,
c	identifier
=	=
10	whole_lit
\n	newline
m	identifier
=	=