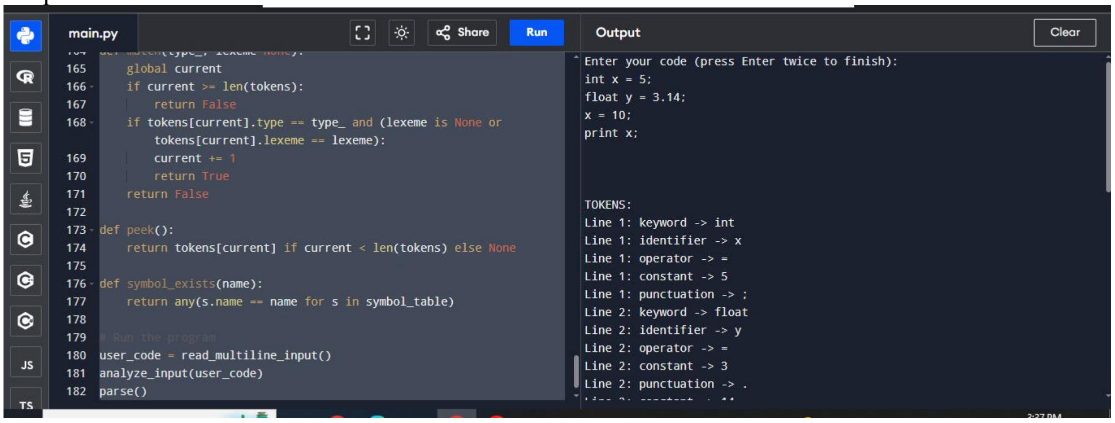
```
import re

symbol_table = []
tokens = []
current = 0

class Symbol:
    def __init__(self, index, name, type_, value, line):
        self.index = index
        self.name = name
        self.type = type_
        self.value = value
        self.line = line

class Token:
    def __init__(self, type_, lexeme, line):
        self.type = type_
        self.lexeme = lexeme
        self.line = line

def read_multiline_input():
    print("Enter your code (press Enter twice to finish):")
    lines = []
    while True:
        line = input()
        if line == "":
            break
        lines.append(line)
    return "\n".join(lines)

def analyze_input(user_input):
    global tokens, symbol_table
    keywords = ["int", "float", "print"]
    var_regex = re.compile(r"^[A-Za-z_][A-Za-z0-9_]*$")
    const_regex = re.compile(r"^[0-9]+(\.[0-9]+)?$")
    op_regex = re.compile(r"^[+\-*/=]$")
    punc_regex = re.compile(r"^[.,;:{}()\[\]]$")

    lines = user_input.split("\n")
    line_num = 0
    symbol_index = 1

    for raw_line in lines:
        if not raw_line.strip():
            continue
        line_num += 1
        line_tokens = tokenize(raw_line)

        for i, token in enumerate(line_tokens):
            if token in keywords:
                tokens.append(Token("keyword", token, line_num))
            elif const_regex.match(token):
                tokens.append(Token("constant", token, line_num))
            elif op_regex.match(token):
                tokens.append(Token("operator", token, line_num))
            elif punc_regex.match(token):
                tokens.append(Token("punctuation", token, line_num))
            elif var_regex.match(token):
                if not symbol_exists(token):
                    type_ = line_tokens[i - 1] if i > 0 and line_tokens[i - 1] in keywords else "unknown"
                    value = line_tokens[i + 2] if (i + 2 < len(line_tokens) and line_tokens[i + 1] == "=") else ""
                    symbol_table.append(Symbol(symbol_index, token, type_, value, line_num))
                    symbol_index += 1
                tokens.append(Token("identifier", token, line_num))
            else:
                tokens.append(Token("unknown", token, line_num))

    print("\nTOKENS:")
    for t in tokens:
        print(f"Line {t.line}: {t.type} -> {t.lexeme}")

    print("\nSYMBOL TABLE:")
    print("Index | Name | Type | Value | Line")
    for s in symbol_table:
        print(f"{s.index:5} | {s.name:5} | {s.type:5} | {s.value:5} | {s.line}")

def tokenize(line):
    tokens = []
    current = ""
    for c in line:
        if c.isspace():
            if current:
                tokens.append(current)
                current = ""
        elif c in "+-*/=.,;:{}()[]":
            if current:
                tokens.append(current)
                current = ""
            tokens.append(c)
        else:
            current += c
    if current:
        tokens.append(current)
    return tokens

def parse():
    global current
    print("\n---------------- PARSING ----------------")
    while current < len(tokens):
        if not statement():
            t = peek()
            print(f"[Syntax Error] Line {t.line if t else '?'}: Unexpected token '{t.lexeme if t else 'EOF'}'")
            current += 1

def statement():
    return declaration() or assignment() or print_stmt()

def declaration():
    global current
    start = current
    if match("keyword", "int") or match("keyword", "float"):
        type_ = tokens[start].lexeme
        if match("identifier"):
            name = tokens[current - 1].lexeme
            value = ""
            if match("operator", "="):
                if not expression():
                    return False
                value = tokens[current - 1].lexeme
            if match("punctuation", ";"):
                print(f"Matched Declaration: {type_} {name} {'= ' + value if value else ''};")
                return True
            else:
                print(f"[Syntax Error] Line {tokens[current - 1].line}: Missing ';'")
        else:
            print(f"[Syntax Error] Line {tokens[current].line}: Expected identifier after type.")
    current = start
    return False

def assignment():
    global current
    start = current
    if match("identifier"):
        name = tokens[current - 1].lexeme
        if match("operator", "="):
            if not expression():
                return False
            value = tokens[current - 1].lexeme
            if match("punctuation", ";"):
                print(f"Matched Assignment: {name} = {value};")
                return True
            else:
                print(f"[Syntax Error] Line {tokens[current - 1].line}: Missing ';'")
    current = start
    return False

def print_stmt():
    global current
    start = current
    if match("keyword", "print"):
        if match("identifier"):
            id_ = tokens[current - 1].lexeme
            if match("punctuation", ";"):
                print(f"Matched Print: print {id_};")
                return True
            else:
                print(f"[Syntax Error] Line {tokens[current - 1].line}: Missing ';'")
    current = start
    return False

def expression():
    return match("identifier") or match("constant")

def match(type_, lexeme=None):
    global current
    if current >= len(tokens):
        return False
    if tokens[current].type == type_ and (lexeme is None or tokens[current].lexeme == lexeme):
        current += 1
        return True
    return False

def peek():
    return tokens[current] if current < len(tokens) else None

def symbol_exists(name):
    return any(s.name == name for s in symbol_table)

# Run the program
user_code = read_multiline_input()
```

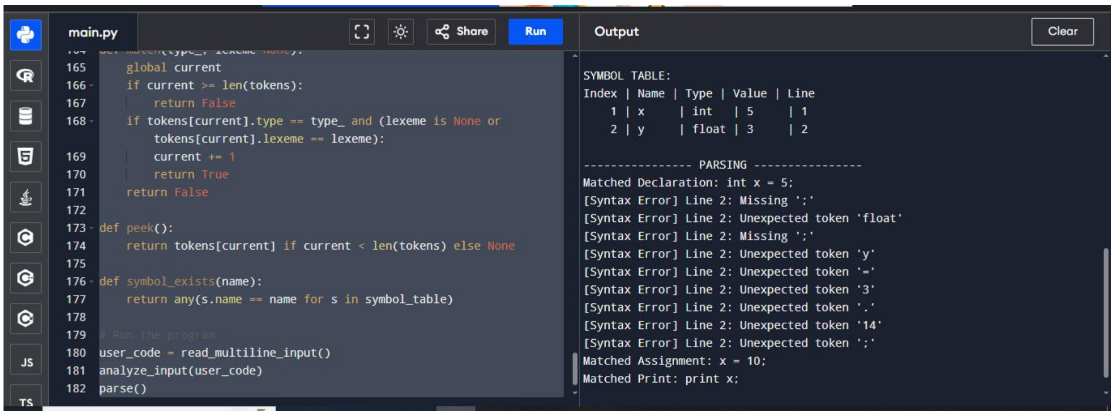
```
analyze_input(user_code)
parse()
```

Output:



```python
    global current
    if current >= len(tokens):
        return False
    if tokens[current].type == type_ and (lexeme is None or
        tokens[current].lexeme == lexeme):
        current += 1
        return True
    return False

def peek():
    return tokens[current] if current < len(tokens) else None

def symbol_exists(name):
    return any(s.name == name for s in symbol_table)

# Run the program
user_code = read_multiline_input()
analyze_input(user_code)
parse()
```

```
Enter your code (press Enter twice to finish):
int x = 5;
float y = 3.14;
x = 10;
print x;



TOKENS:
Line 1: keyword -> int
Line 1: identifier -> x
Line 1: operator -> =
Line 1: constant -> 5
Line 1: punctuation -> ;
Line 2: keyword -> float
Line 2: identifier -> y
Line 2: operator -> =
Line 2: constant -> 3
Line 2: punctuation -> .
```



```python
    global current
    if current >= len(tokens):
        return False
    if tokens[current].type == type_ and (lexeme is None or
        tokens[current].lexeme == lexeme):
        current += 1
        return True
    return False

def peek():
    return tokens[current] if current < len(tokens) else None

def symbol_exists(name):
    return any(s.name == name for s in symbol_table)

# Run the program
user_code = read_multiline_input()
analyze_input(user_code)
parse()
```

```

SYMBOL TABLE:
Index | Name | Type | Value | Line
    1 | x     | int   | 5     | 1
    2 | y     | float | 3     | 2

--------------- PARSING ---------------
Matched Declaration: int x = 5;
[Syntax Error] Line 2: Missing ';'
[Syntax Error] Line 2: Unexpected token 'float'
[Syntax Error] Line 2: Missing ';'
[Syntax Error] Line 2: Unexpected token 'y'
[Syntax Error] Line 2: Unexpected token '='
[Syntax Error] Line 2: Unexpected token '3'
[Syntax Error] Line 2: Unexpected token '.'
[Syntax Error] Line 2: Unexpected token '14'
[Syntax Error] Line 2: Unexpected token ';'
Matched Assignment: x = 10;
Matched Print: print x;
```