UNIVERSITY OF KARACHI
UBIT

COMPILER CONSTRUCTION
LAB DOCUMENTATION

GROUP 1
MEMBERS NAME

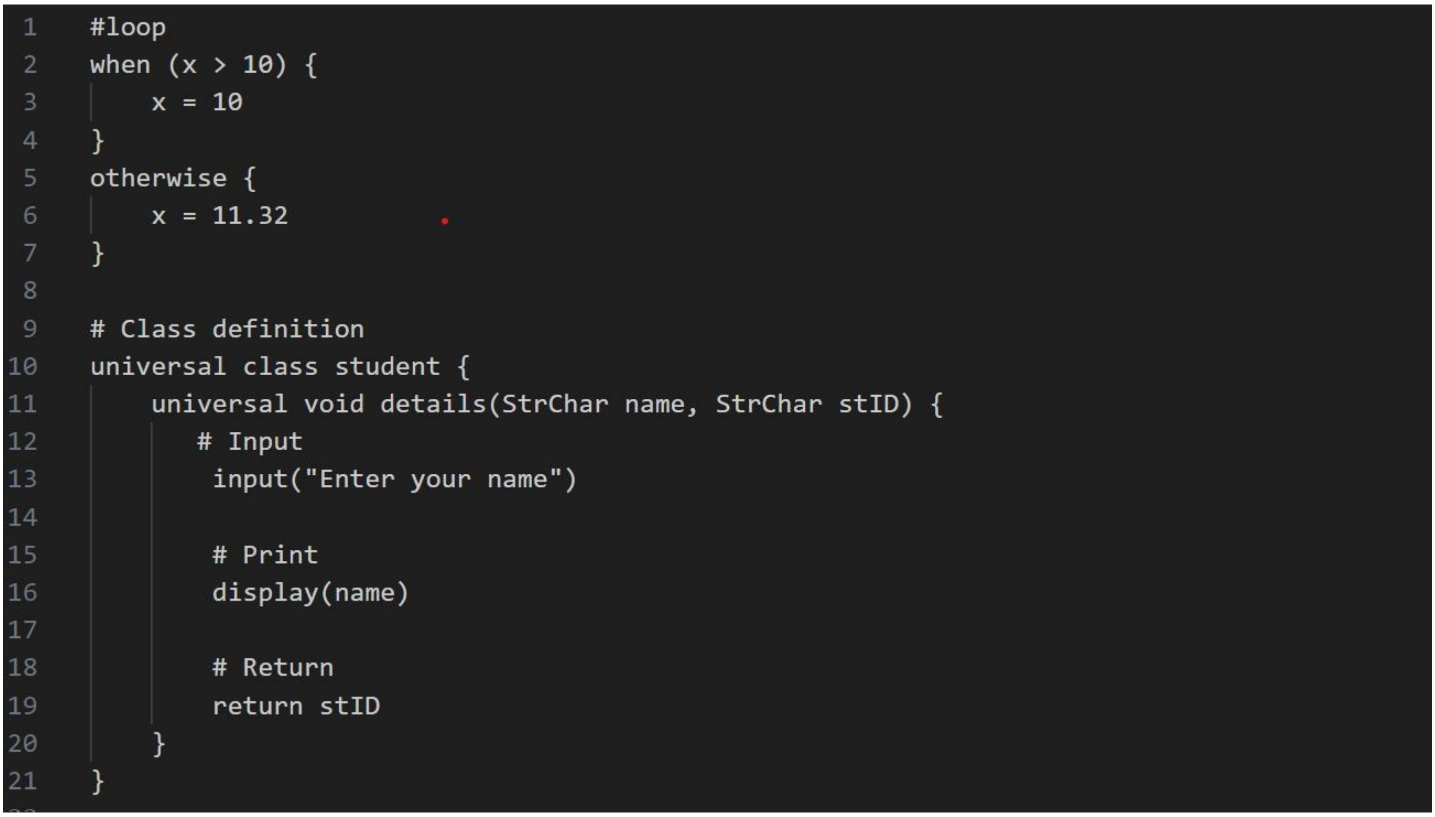
RIMSHA LARAIB (B21110006107)

MEHAK FATIMA (B21110006057)

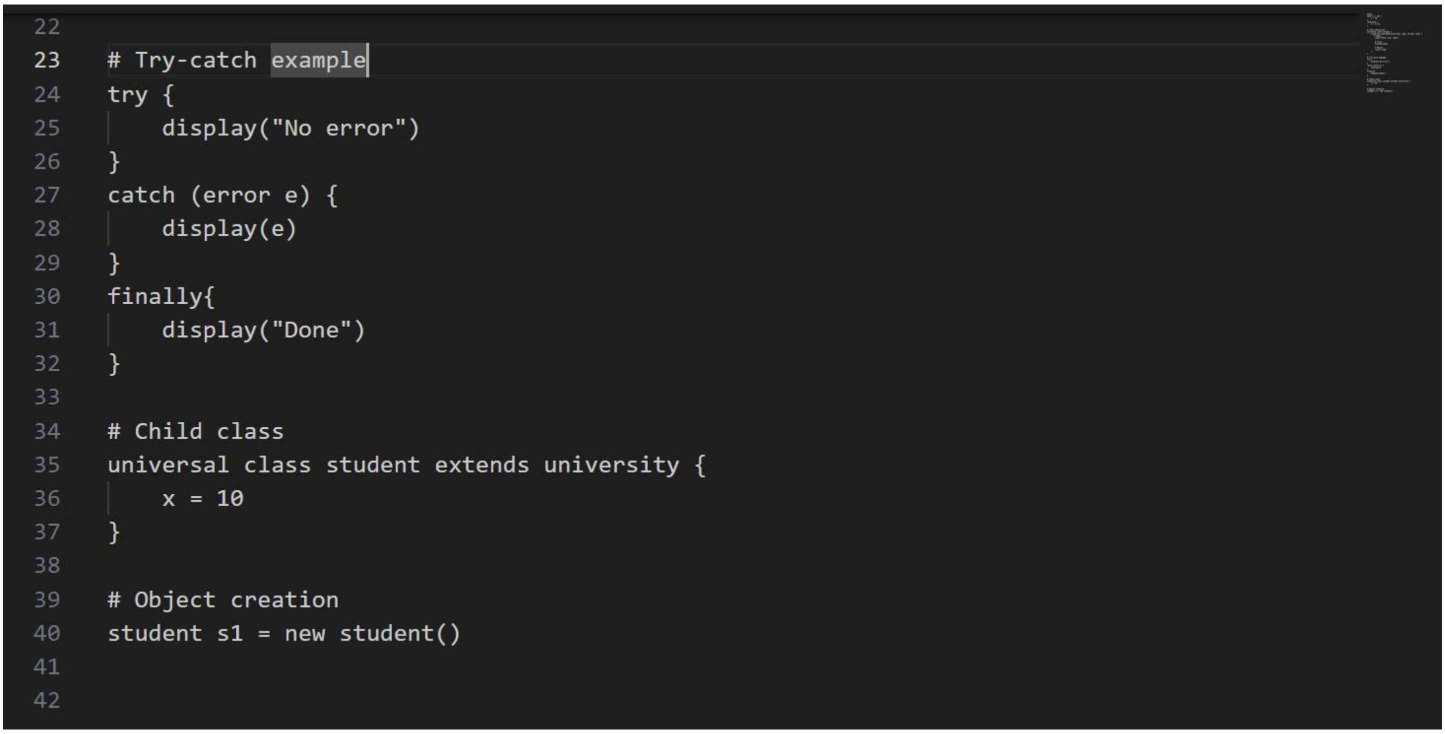
SUBMITTED TO: MA’AM ATTIA AGHA

LEXICAL ANALYZER

TEXT FILE:

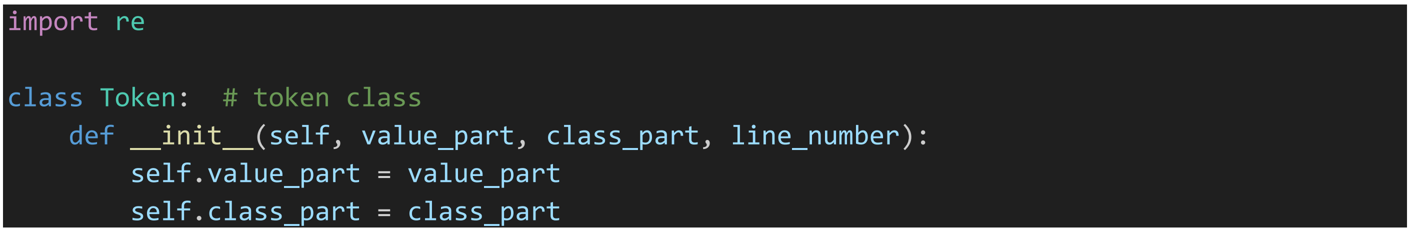


```
#loop
when (x > 10) {
    x = 10
}
otherwise {
    x = 11.32
}

# Class definition
universal class student {
    universal void details(StrChar name, StrChar stID) {
       # Input
        input("Enter your name")

        # Print
        display(name)

        # Return
        return stID
    }
}
```

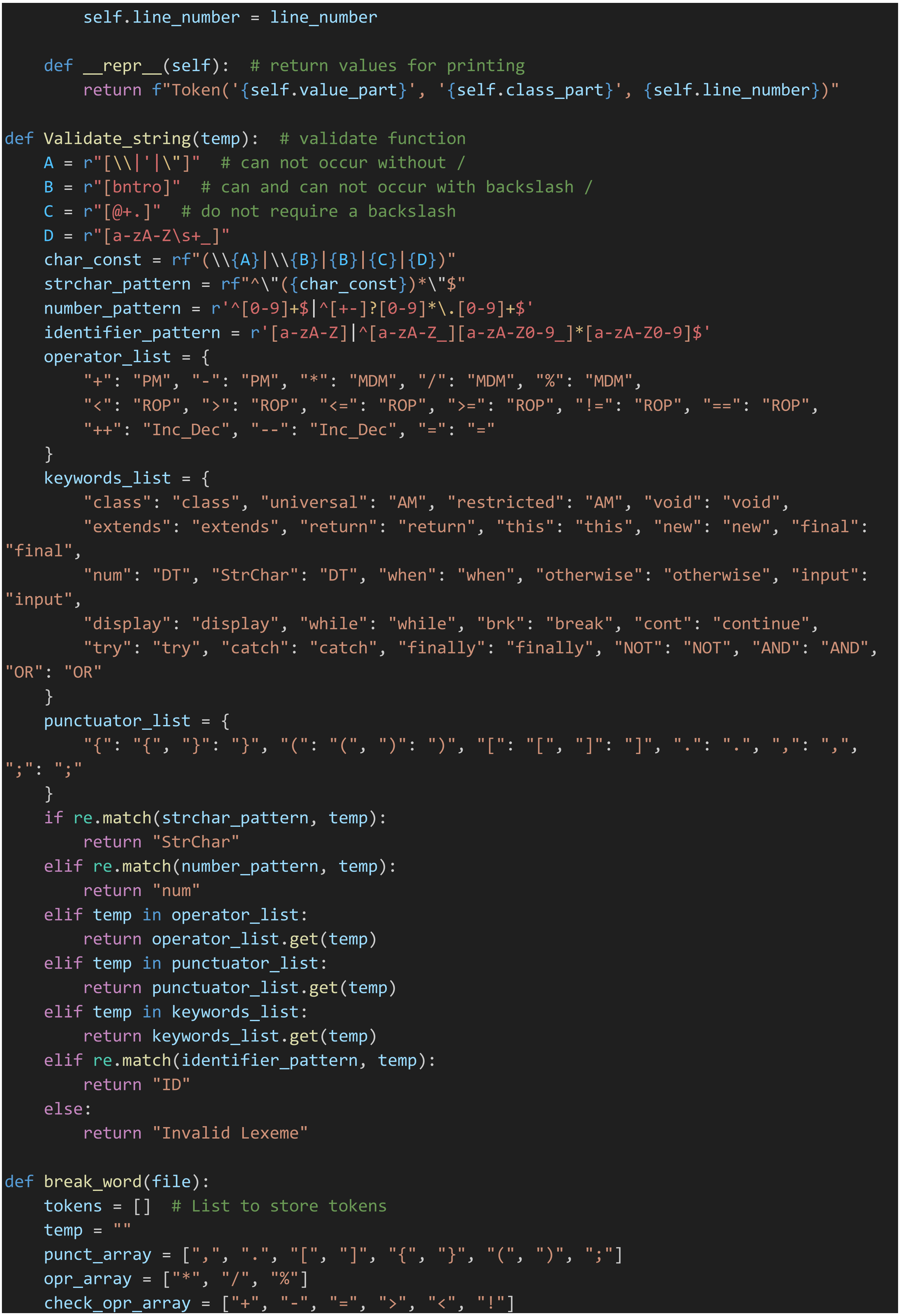


```

# Try-catch example
try {
    display("No error")
}
catch (error e) {
    display(e)
}
finally{
    display("Done")
}

# Child class
universal class student extends university {
    x = 10
}

# Object creation
student s1 = new student()


```

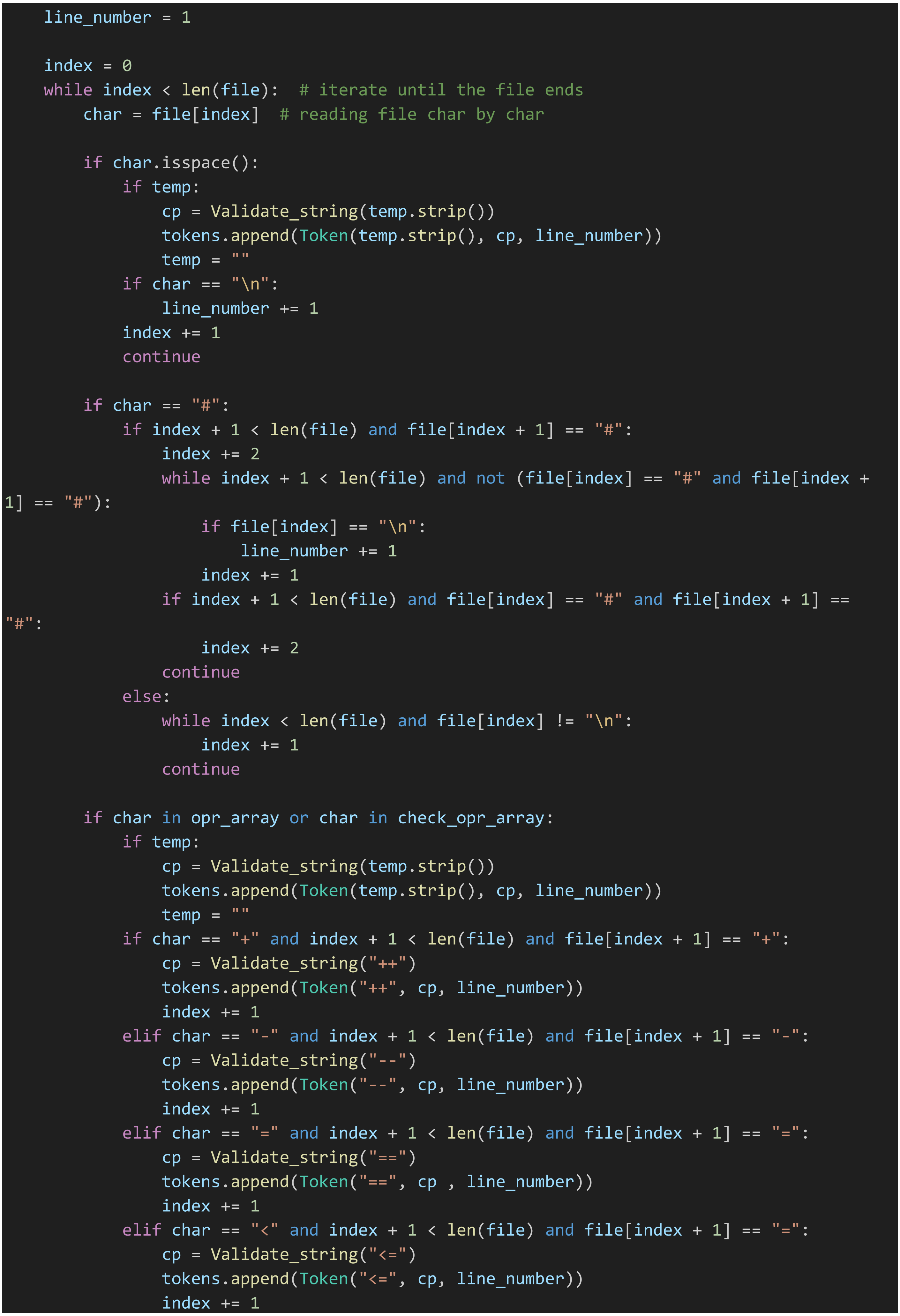
CODE:



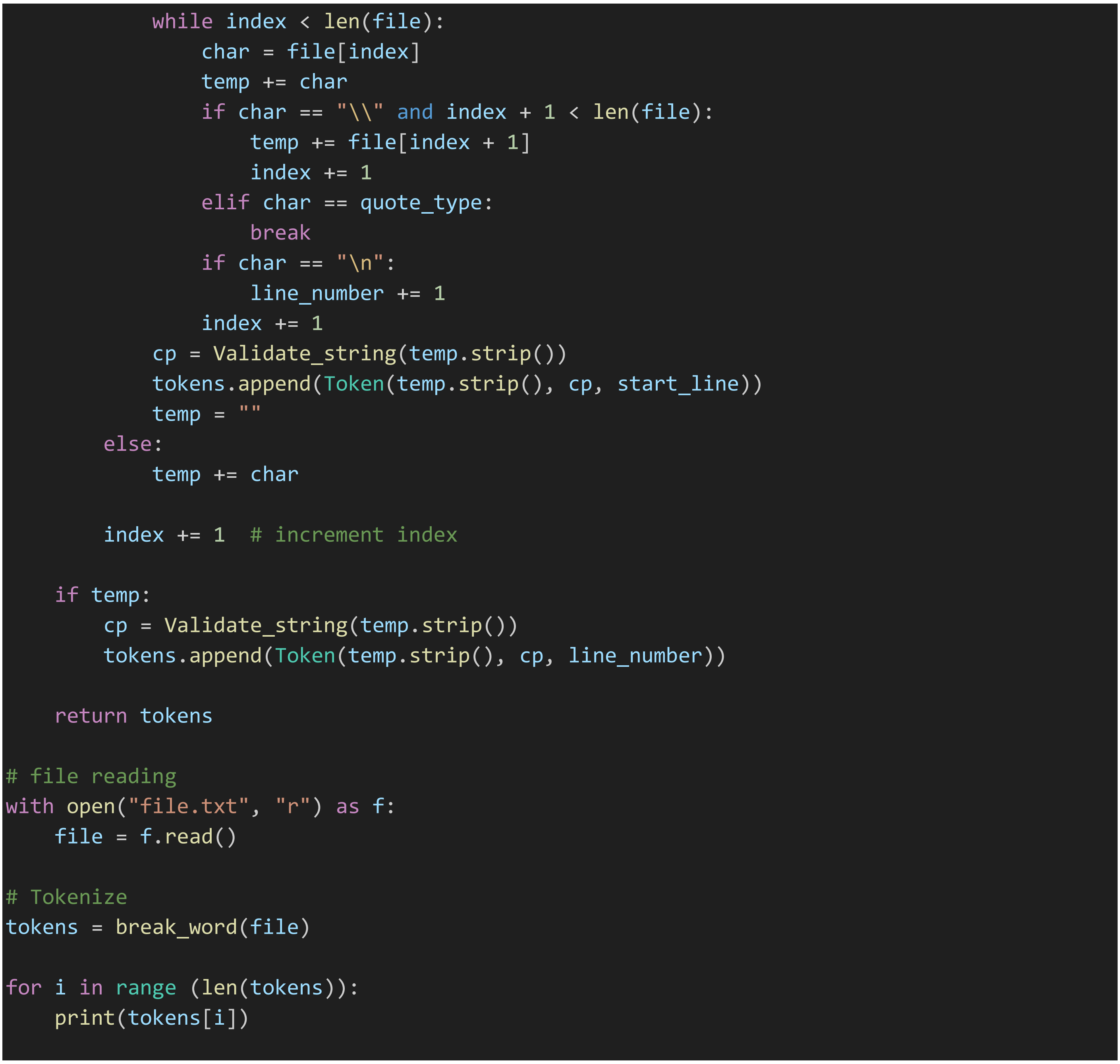
```
import re

class Token:  # token class
    def __init__(self, value_part, class_part, line_number):
        self.value_part = value_part
        self.class_part = class_part
```



```
        self.line_number = line_number

    def __repr__(self):  # return values for printing
        return f"Token('{self.value_part}', '{self.class_part}', {self.line_number})"

def Validate_string(temp):  # validate function
    A = r"[\\|'|\"]"  # can not occur without /
    B = r"[bntro]"  # can and can not occur with backslash /
    C = r"[@+.]"  # do not require a backslash
    D = r"[a-zA-Z\s+_]"
    char_const = rf"(\\{A}|\\{B}|{B}|{C}|{D})"
    strchar_pattern = rf"^\"({char_const})*\"$"
    number_pattern = r'^[0-9]+$|^[+-]?[0-9]*\.[0-9]+$'
    identifier_pattern = r'[a-zA-Z]|^[a-zA-Z_][a-zA-Z0-9_]*[a-zA-Z0-9]$'
    operator_list = {
        "+": "PM", "-": "PM", "*": "MDM", "/": "MDM", "%": "MDM",
        "<": "ROP", ">": "ROP", "<=": "ROP", ">=": "ROP", "!=": "ROP", "==": "ROP",
        "++": "Inc_Dec", "--": "Inc_Dec", "=": "="
    }
    keywords_list = {
        "class": "class", "universal": "AM", "restricted": "AM", "void": "void",
        "extends": "extends", "return": "return", "this": "this", "new": "new", "final":
"final",
        "num": "DT", "StrChar": "DT", "when": "when", "otherwise": "otherwise", "input":
"input",
        "display": "display", "while": "while", "brk": "break", "cont": "continue",
        "try": "try", "catch": "catch", "finally": "finally", "NOT": "NOT", "AND": "AND",
"OR": "OR"
    }
    punctuator_list = {
        "{": "{", "}": "}", "(": "(", ")": ")", "[": "[", "]": "]", ".": ".", ",": ",",
";": ";"
    }
    if re.match(strchar_pattern, temp):
        return "StrChar"
    elif re.match(number_pattern, temp):
        return "num"
    elif temp in operator_list:
        return operator_list.get(temp)
    elif temp in punctuator_list:
        return punctuator_list.get(temp)
    elif temp in keywords_list:
        return keywords_list.get(temp)
    elif re.match(identifier_pattern, temp):
        return "ID"
    else:
        return "Invalid Lexeme"

def break_word(file):
    tokens = []  # List to store tokens
    temp = ""
    punct_array = [",", ".", "[", "]", "{", "}", "(", ")", ";"]
    opr_array = ["*", "/", "%"]
    check_opr_array = ["+", "-", "=", ">", "<", "!"]
```

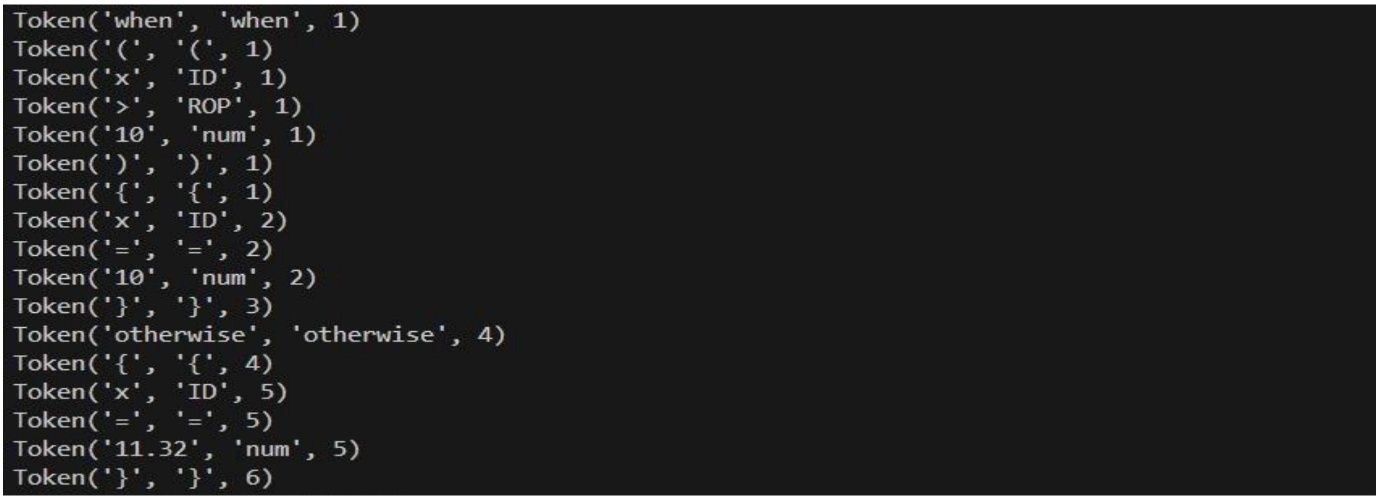


```
    line_number = 1

    index = 0
    while index < len(file):  # iterate until the file ends
        char = file[index]  # reading file char by char

        if char.isspace():
            if temp:
                cp = Validate_string(temp.strip())
                tokens.append(Token(temp.strip(), cp, line_number))
                temp = ""
            if char == "\n":
                line_number += 1
            index += 1
            continue

        if char == "#":
            if index + 1 < len(file) and file[index + 1] == "#":
                index += 2
                while index + 1 < len(file) and not (file[index] == "#" and file[index +
1] == "#"):
                    if file[index] == "\n":
                        line_number += 1
                    index += 1
                if index + 1 < len(file) and file[index] == "#" and file[index + 1] ==
"#":
                    index += 2
                continue
            else:
                while index < len(file) and file[index] != "\n":
                    index += 1
                continue

        if char in opr_array or char in check_opr_array:
            if temp:
                cp = Validate_string(temp.strip())
                tokens.append(Token(temp.strip(), cp, line_number))
                temp = ""
            if char == "+" and index + 1 < len(file) and file[index + 1] == "+":
                cp = Validate_string("++")
                tokens.append(Token("++", cp, line_number))
                index += 1
            elif char == "-" and index + 1 < len(file) and file[index + 1] == "-":
                cp = Validate_string("--")
                tokens.append(Token("--", cp, line_number))
                index += 1
            elif char == "=" and index + 1 < len(file) and file[index + 1] == "=":
                cp = Validate_string("==")
                tokens.append(Token("==", cp , line_number))
                index += 1
            elif char == "<" and index + 1 < len(file) and file[index + 1] == "=":
                cp = Validate_string("<=")
                tokens.append(Token("<=", cp, line_number))
                index += 1
            elif char == ">" and index + 1 < len(file) and file[index + 1] == "=":
                cp = Validate_string(">=")
                tokens.append(Token(">=", cp, line_number))
                index += 1
            elif char == "!" and index + 1 < len(file) and file[index + 1] == "=":
                cp = Validate_string("!=")
                tokens.append(Token("!=", cp, line_number))
                index += 1
            else:
                tokens.append(Token(char, Validate_string(char), line_number))
        elif char in punct_array:
            if char == '.':
                # Check for previous and next parts around '.'
                prev_part = temp
                next_part = file[index + 1:].lstrip()
                prev_word = re.findall(r'\w+', prev_part)[-1] if re.findall(r'\w+',
prev_part) else ''
                next_word = re.findall(r'\w+', next_part)[0] if re.findall(r'\w+',
next_part) else ''

                # Check if the previous part is a number and the next part starts with a
digit
                if prev_word.isdigit() and next_word.isdigit():
                    if '.' in temp:  # If there's already a dot in temp, break the word
                        if temp.strip():
                            cp = Validate_string(temp.strip())
                            tokens.append(Token(temp.strip(), cp, line_number))
                        temp = '.'  # Start a new token with the dot
                    else:
                        temp += char  # Add the '.' to temp since it's part of a number
                else:
                    if temp:
                        cp = Validate_string(temp.strip())
                        tokens.append(Token(temp.strip(), cp, line_number))
                        temp = ""
                    tokens.append(Token(char, Validate_string(char), line_number))  #
Treat the '.' as a punctuator
            else:
                if temp:
                    cp = Validate_string(temp.strip())
                    tokens.append(Token(temp.strip(), cp, line_number))
                    temp = ""
                tokens.append(Token(char, Validate_string(char), line_number))  # Add the
punctuation token

        elif char == "\"":
            if temp:
                cp = Validate_string(temp.strip())
                tokens.append(Token(temp.strip(), cp, line_number))
                temp = ""
            quote_type = char
            temp += char
            index += 1
            start_line = line_number
```

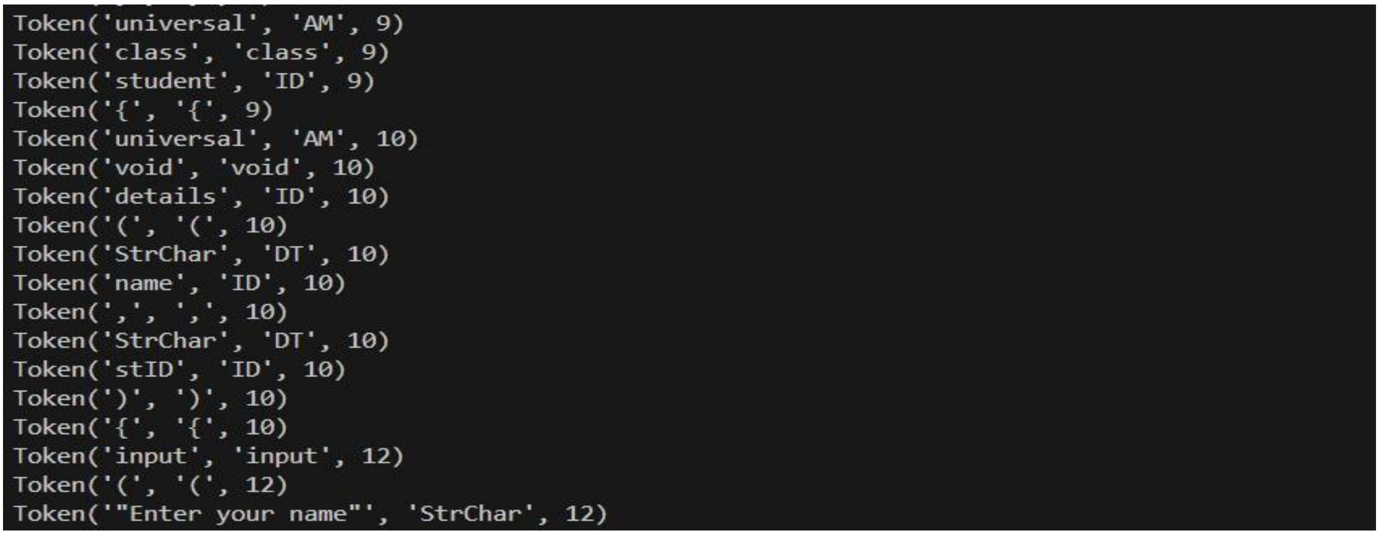


```
            while index < len(file):
                char = file[index]
                temp += char
                if char == "\\" and index + 1 < len(file):
                    temp += file[index + 1]
                    index += 1
                elif char == quote_type:
                    break
                if char == "\n":
                    line_number += 1
                index += 1
            cp = Validate_string(temp.strip())
            tokens.append(Token(temp.strip(), cp, start_line))
            temp = ""
        else:
            temp += char

        index += 1  # increment index

    if temp:
        cp = Validate_string(temp.strip())
        tokens.append(Token(temp.strip(), cp, line_number))

    return tokens

# file reading
with open("file.txt", "r") as f:
    file = f.read()

# Tokenize
tokens = break_word(file)

for i in range (len(tokens)):
    print(tokens[i])
```

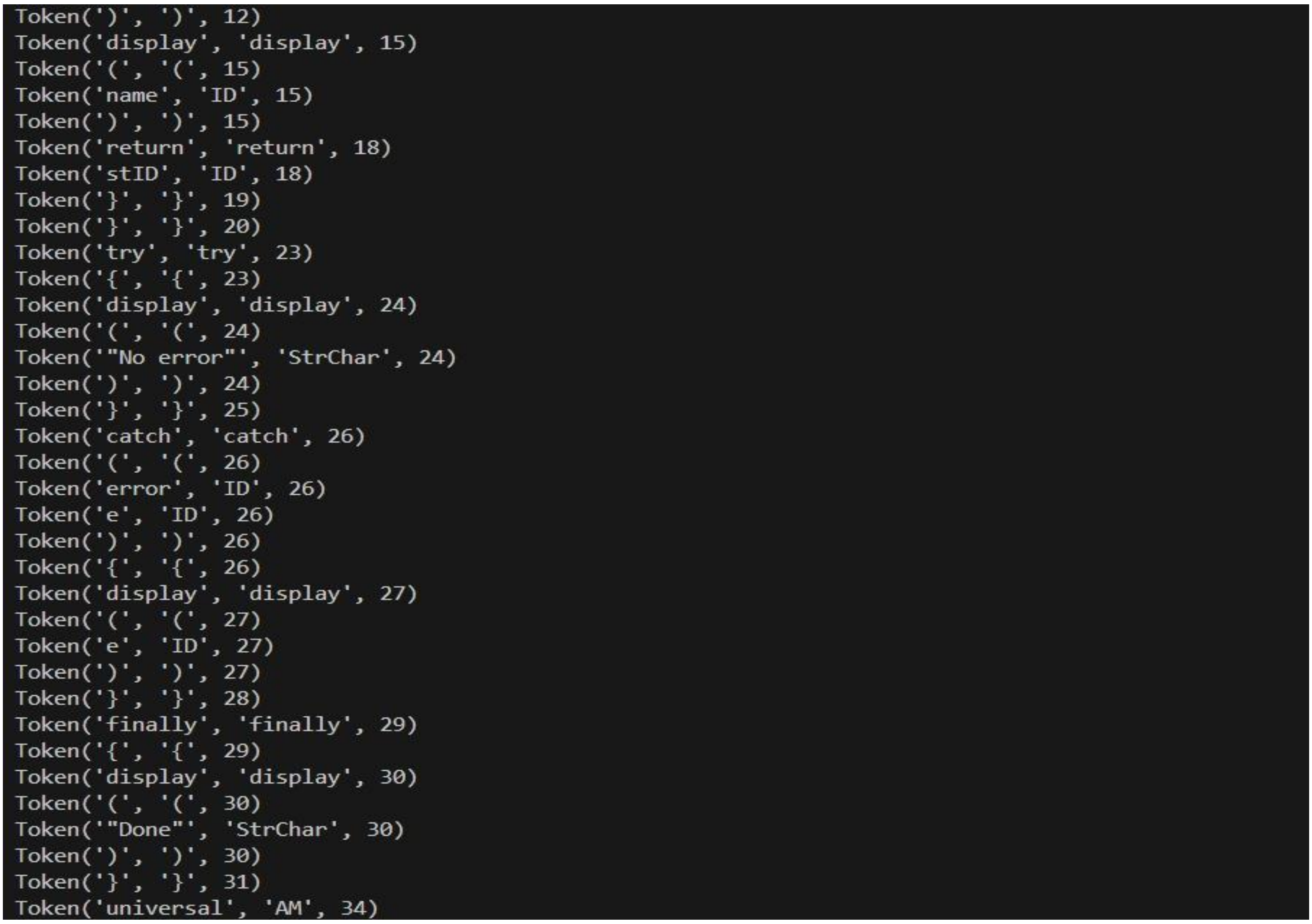
OUTPUT:



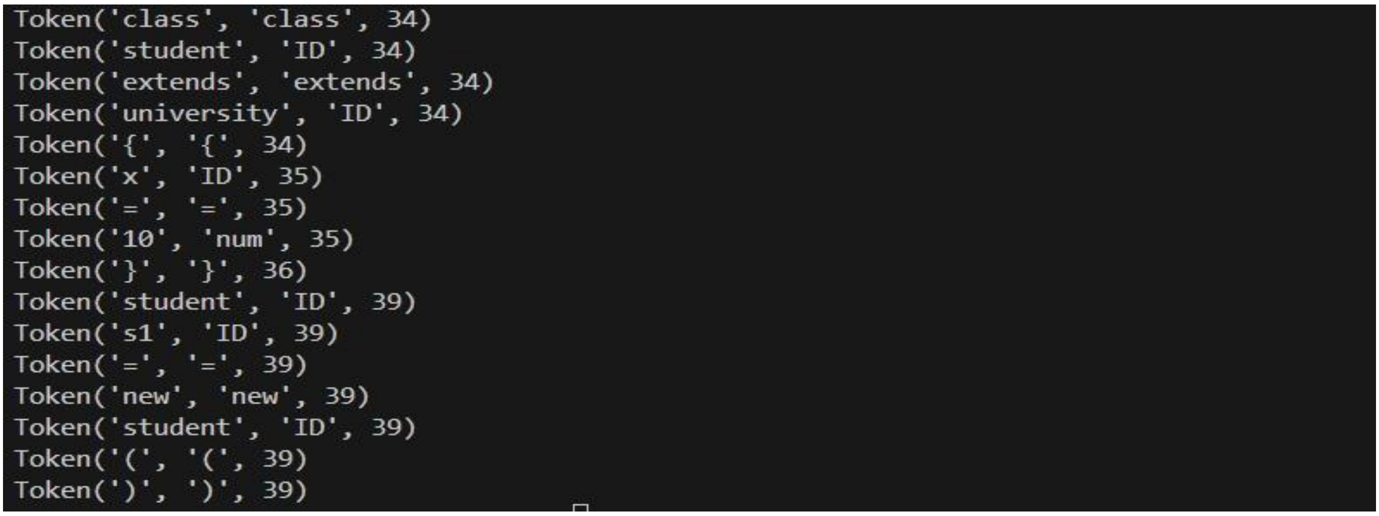
```
Token('when', 'when', 1)
Token('(', '(', 1)
Token('x', 'ID', 1)
Token('>', 'ROP', 1)
Token('10', 'num', 1)
Token(')', ')', 1)
Token('{', '{', 1)
Token('x', 'ID', 2)
Token('=', '=', 2)
Token('10', 'num', 2)
Token('}', '}', 3)
Token('otherwise', 'otherwise', 4)
Token('{', '{', 4)
Token('x', 'ID', 5)
Token('=', '=', 5)
Token('11.32', 'num', 5)
Token('}', '}', 6)
```



```
Token('universal', 'AM', 9)
Token('class', 'class', 9)
Token('student', 'ID', 9)
Token('{', '{', 9)
Token('universal', 'AM', 10)
Token('void', 'void', 10)
Token('details', 'ID', 10)
Token('(', '(', 10)
Token('StrChar', 'DT', 10)
Token('name', 'ID', 10)
Token(',', ',', 10)
Token('StrChar', 'DT', 10)
Token('stID', 'ID', 10)
Token(')', ')', 10)
Token('{', '{', 10)
Token('input', 'input', 12)
Token('(', '(', 12)
Token('"Enter your name"', 'StrChar', 12)
```



```
Token(')', ')', 12)
Token('display', 'display', 15)
Token('(', '(', 15)
Token('name', 'ID', 15)
Token(')', ')', 15)
Token('return', 'return', 18)
Token('stID', 'ID', 18)
Token('}', '}', 19)
Token('}', '}', 20)
Token('try', 'try', 23)
Token('{', '{', 23)
Token('display', 'display', 24)
Token('(', '(', 24)
Token('"No error"', 'StrChar', 24)
Token(')', ')', 24)
Token('}', '}', 25)
Token('catch', 'catch', 26)
Token('(', '(', 26)
Token('error', 'ID', 26)
Token('e', 'ID', 26)
Token(')', ')', 26)
Token('{', '{', 26)
Token('display', 'display', 27)
Token('(', '(', 27)
Token('e', 'ID', 27)
Token(')', ')', 27)
Token('}', '}', 28)
Token('finally', 'finally', 29)
Token('{', '{', 29)
Token('display', 'display', 30)
Token('(', '(', 30)
Token('"Done"', 'StrChar', 30)
Token(')', ')', 30)
Token('}', '}', 31)
Token('universal', 'AM', 34)
```

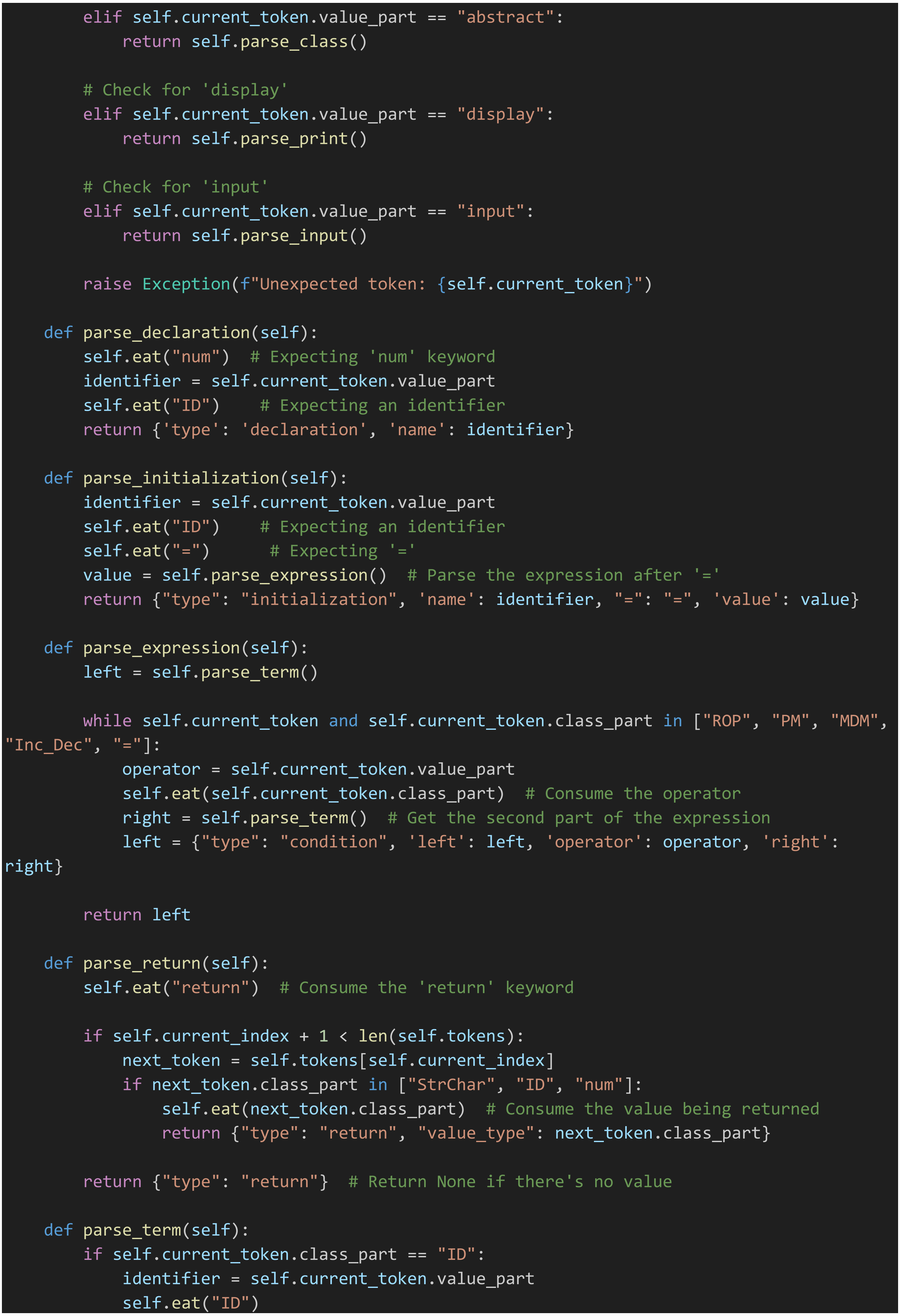


```
Token('class', 'class', 34)
Token('student', 'ID', 34)
Token('extends', 'extends', 34)
Token('university', 'ID', 34)
Token('{', '{', 34)
Token('x', 'ID', 35)
Token('=', '=', 35)
Token('10', 'num', 35)
Token('}', '}', 36)
Token('student', 'ID', 39)
Token('s1', 'ID', 39)
Token('=', '=', 39)
Token('new', 'new', 39)
Token('student', 'ID', 39)
Token('(', '(', 39)
Token(')', ')', 39)
```

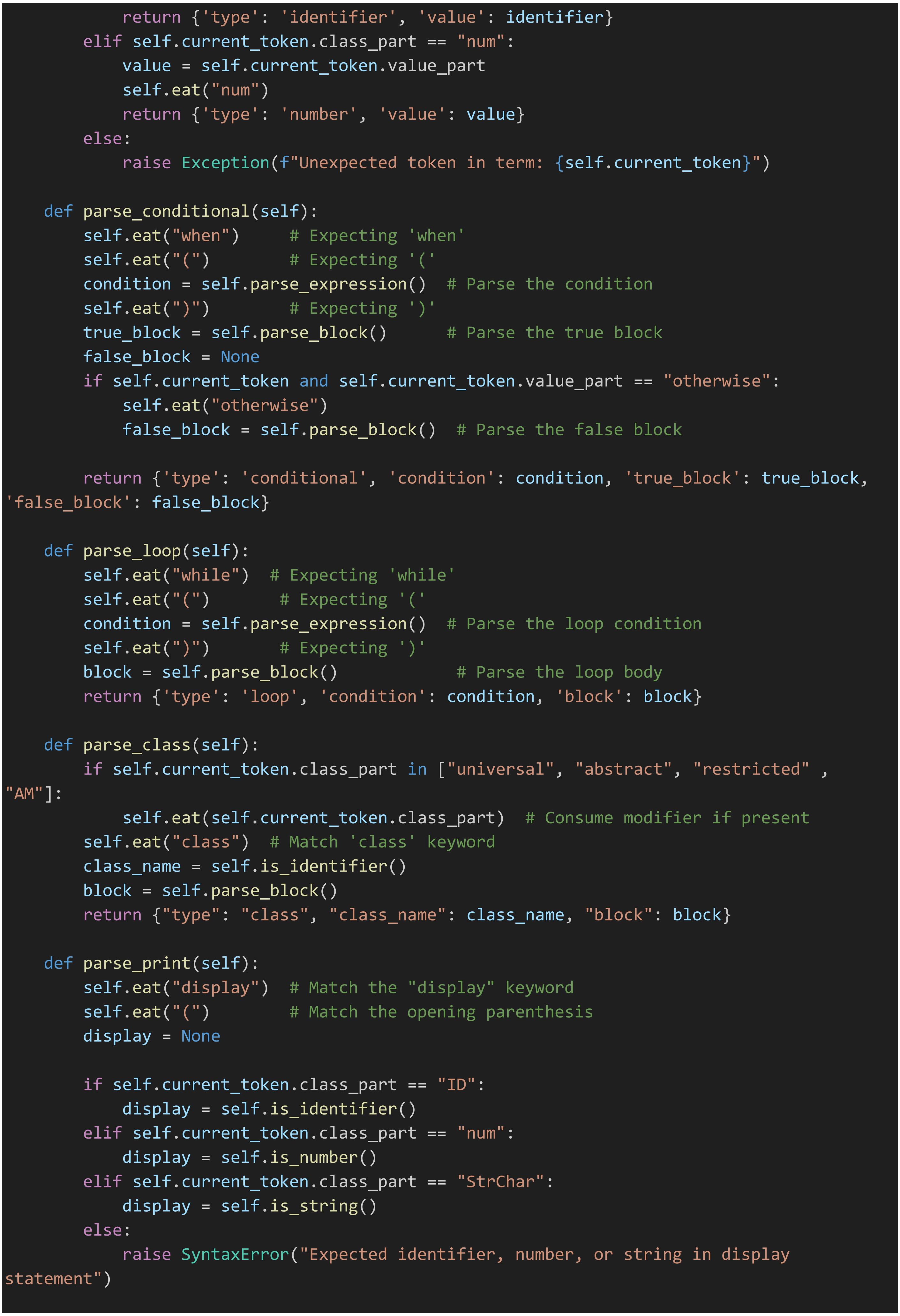
SYNTAX ANALYZER

CODE:

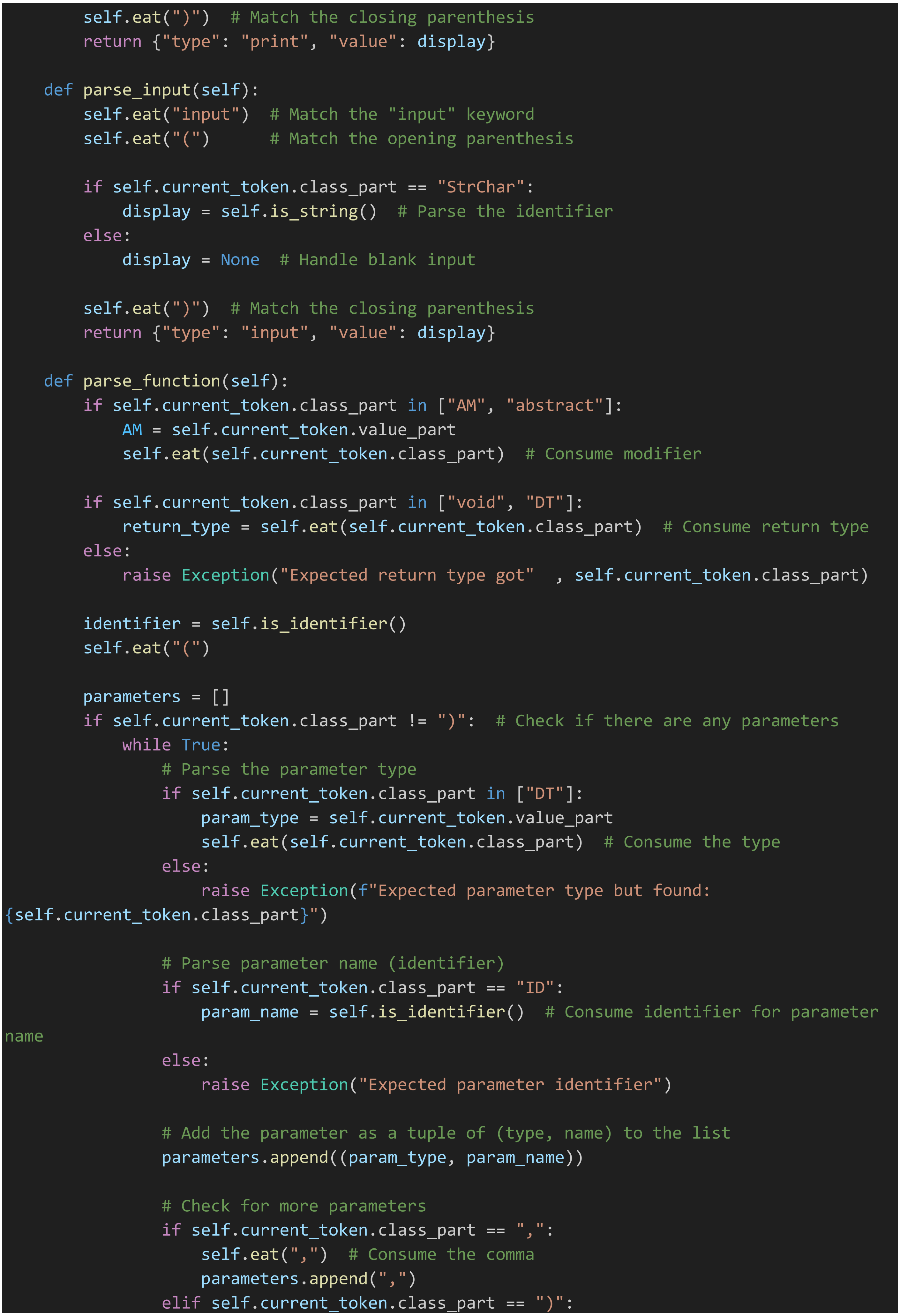
```
import json
from lexical import tokens  # Assuming tokens is a list of Token objects

class Token:
    def __init__(self, value_part, class_part, line_number):
        self.value_part = value_part
        self.class_part = class_part
        self.line_number = line_number

    def __repr__(self):
        return f"Token(value='{self.value_part}', type='{self.class_part}',
line={self.line_number})"

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.current_index = 0
        self.current_token = tokens[0] if tokens else None

    def eat(self, token_type):
        if self.current_token and self.current_token.class_part == token_type:
            self.current_index += 1
            if self.current_index < len(self.tokens):
                self.current_token = self.tokens[self.current_index]
            else:
                self.current_token = None
        else:
            raise Exception(f"Expected token type {token_type}, but got
{self.current_token}")

    def parse_program(self):
        statements = []
        while self.current_index < len(self.tokens):
            statements.append(self.parse_statement())
        return statements

    def parse_statement(self):
        # Check for 'break' statement
        if self.current_token.value_part == "break":
            self.eat("break")
            return {"type": "break"}

        # Check for 'continue' statement
        elif self.current_token.value_part == "continue":
            self.eat("continue")
            return {"type": "continue"}

        # Check for 'return' statement
        elif self.current_token.value_part == "return":
            return self.parse_return()

        # Check for identifier (ID)
        if self.current_token.class_part == "ID":
            if self.current_index + 1 < len(self.tokens):
                next_token = self.tokens[self.current_index + 1]

                if next_token.class_part == "(":
                    # Function call
                    return self.parse_funcCalling()
                elif next_token.class_part == "ID":
                    # Object creation
                    return self.parse_objects()
                elif next_token.class_part == ".":
                    # Object method calling
                    return self.parse_objCalling()

            # Default to initialization
            return self.parse_initialization()

        # Check for 'number' declaration
        elif self.current_token.class_part == "num":
            return self.parse_declaration()

        # Check for 'when' conditional block
        elif self.current_token.value_part == "when":
            return self.parse_conditional()

        # Check for 'while' loop
        elif self.current_token.value_part == "while":
            return self.parse_loop()

        # Check for 'try' block
        elif self.current_token.value_part == "try":
            return self.parse_tryCatch()

        # Check for class/function declarations
        elif self.current_token.class_part in ["AM", "abstract"]:
            if self.current_index + 1 < len(self.tokens):
                next_token = self.tokens[self.current_index + 1]
                if next_token.value_part == "class":
                    if self.current_index + 2 < len(self.tokens) and
self.tokens[self.current_index + 2].class_part == "ID":
                        if self.current_index + 3 < len(self.tokens) and
self.tokens[self.current_index + 3].value_part == "extends":
                            # Child class with inheritance
                            return self.parse_childClass()
                    return self.parse_class()  # Regular class declaration
                elif next_token.class_part in ["void", "ID", "DT"]:
                    # Function declaration
                    return self.parse_function()

        # Check for abstract class definition
```



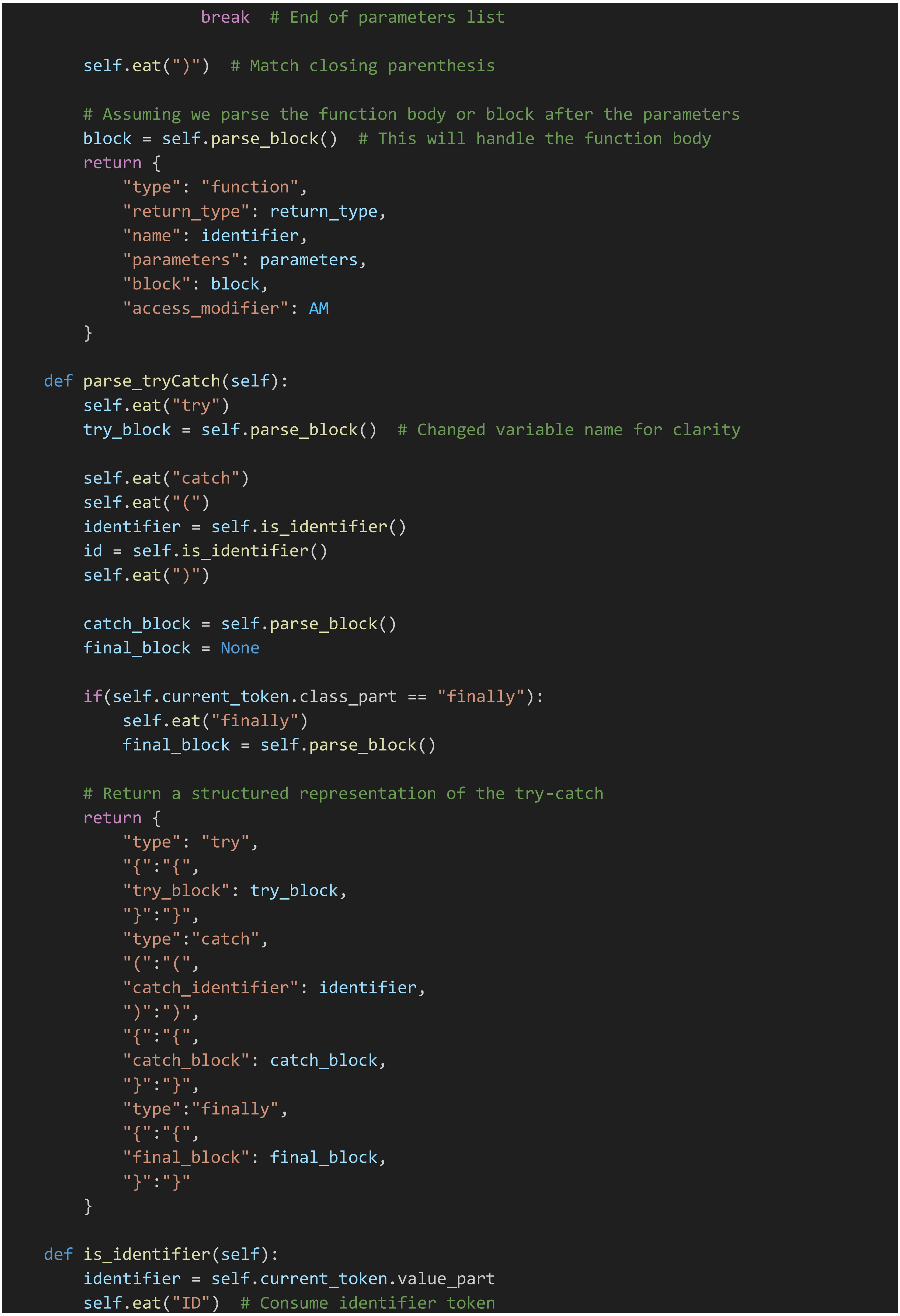
```
        elif self.current_token.value_part == "abstract":
            return self.parse_class()

        # Check for 'display'
        elif self.current_token.value_part == "display":
            return self.parse_print()

        # Check for 'input'
        elif self.current_token.value_part == "input":
            return self.parse_input()

        raise Exception(f"Unexpected token: {self.current_token}")

    def parse_declaration(self):
        self.eat("num")  # Expecting 'num' keyword
        identifier = self.current_token.value_part
        self.eat("ID")    # Expecting an identifier
        return {'type': 'declaration', 'name': identifier}

    def parse_initialization(self):
        identifier = self.current_token.value_part
        self.eat("ID")    # Expecting an identifier
        self.eat("=")      # Expecting '='
        value = self.parse_expression()  # Parse the expression after '='
        return {"type": "initialization", 'name': identifier, "=": "=", 'value': value}

    def parse_expression(self):
        left = self.parse_term()

        while self.current_token and self.current_token.class_part in ["ROP", "PM", "MDM",
"Inc_Dec", "="]:
            operator = self.current_token.value_part
            self.eat(self.current_token.class_part)  # Consume the operator
            right = self.parse_term()  # Get the second part of the expression
            left = {"type": "condition", 'left': left, 'operator': operator, 'right':
right}

        return left

    def parse_return(self):
        self.eat("return")  # Consume the 'return' keyword

        if self.current_index + 1 < len(self.tokens):
            next_token = self.tokens[self.current_index]
            if next_token.class_part in ["StrChar", "ID", "num"]:
                self.eat(next_token.class_part)  # Consume the value being returned
                return {"type": "return", "value_type": next_token.class_part}

        return {"type": "return"}  # Return None if there's no value

    def parse_term(self):
        if self.current_token.class_part == "ID":
            identifier = self.current_token.value_part
            self.eat("ID")
```



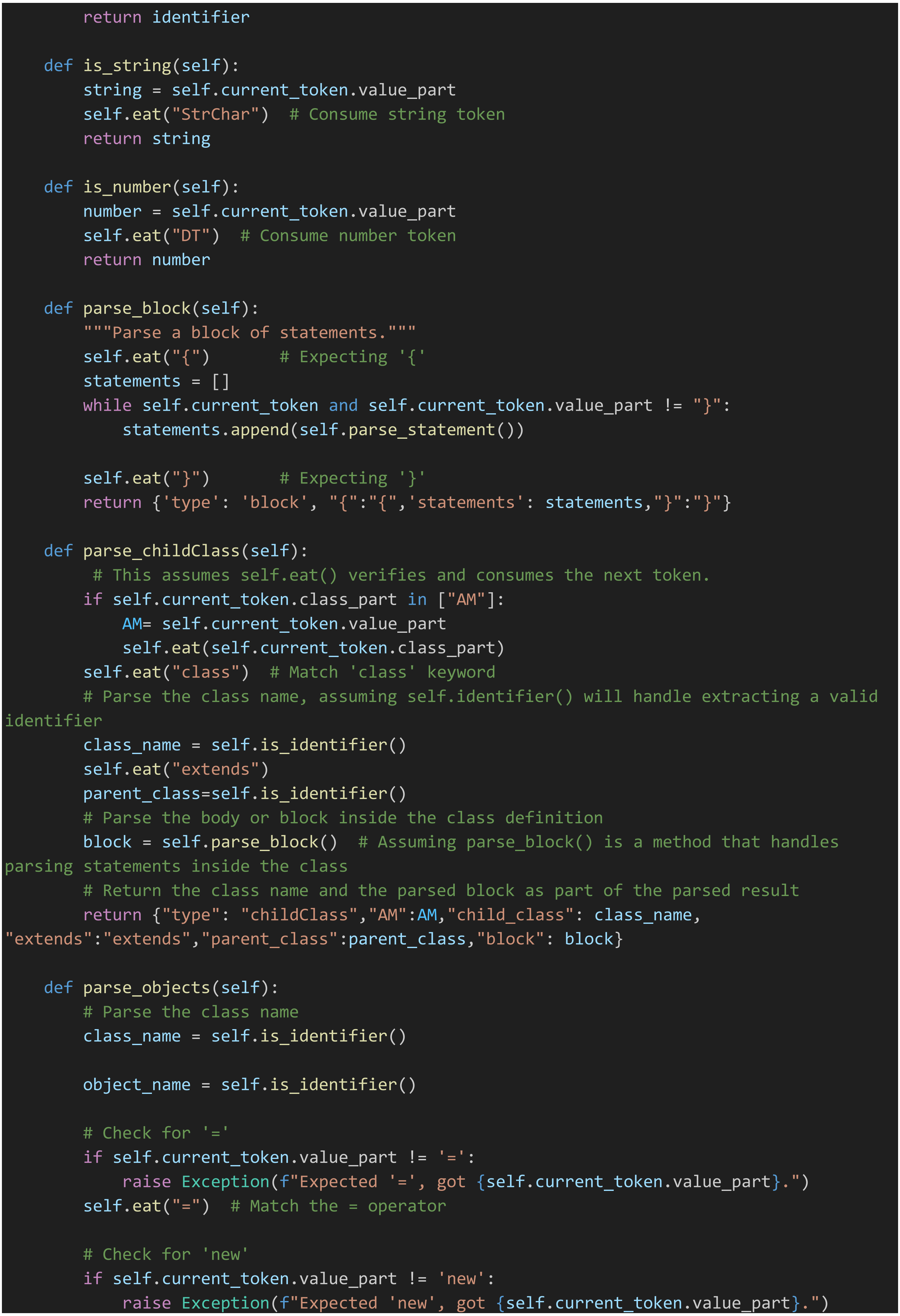
```
            return {'type': 'identifier', 'value': identifier}
        elif self.current_token.class_part == "num":
            value = self.current_token.value_part
            self.eat("num")
            return {'type': 'number', 'value': value}
        else:
            raise Exception(f"Unexpected token in term: {self.current_token}")

    def parse_conditional(self):
        self.eat("when")     # Expecting 'when'
        self.eat("(")        # Expecting '('
        condition = self.parse_expression()  # Parse the condition
        self.eat(")")        # Expecting ')'
        true_block = self.parse_block()      # Parse the true block
        false_block = None
        if self.current_token and self.current_token.value_part == "otherwise":
            self.eat("otherwise")
            false_block = self.parse_block()  # Parse the false block

        return {'type': 'conditional', 'condition': condition, 'true_block': true_block,
'false_block': false_block}

    def parse_loop(self):
        self.eat("while")  # Expecting 'while'
        self.eat("(")       # Expecting '('
        condition = self.parse_expression()  # Parse the loop condition
        self.eat(")")       # Expecting ')'
        block = self.parse_block()            # Parse the loop body
        return {'type': 'loop', 'condition': condition, 'block': block}

    def parse_class(self):
        if self.current_token.class_part in ["universal", "abstract", "restricted" ,
"AM"]:
            self.eat(self.current_token.class_part)  # Consume modifier if present
        self.eat("class")  # Match 'class' keyword
        class_name = self.is_identifier()
        block = self.parse_block()
        return {"type": "class", "class_name": class_name, "block": block}

    def parse_print(self):
        self.eat("display")  # Match the "display" keyword
        self.eat("(")        # Match the opening parenthesis
        display = None

        if self.current_token.class_part == "ID":
            display = self.is_identifier()
        elif self.current_token.class_part == "num":
            display = self.is_number()
        elif self.current_token.class_part == "StrChar":
            display = self.is_string()
        else:
            raise SyntaxError("Expected identifier, number, or string in display
statement")
```



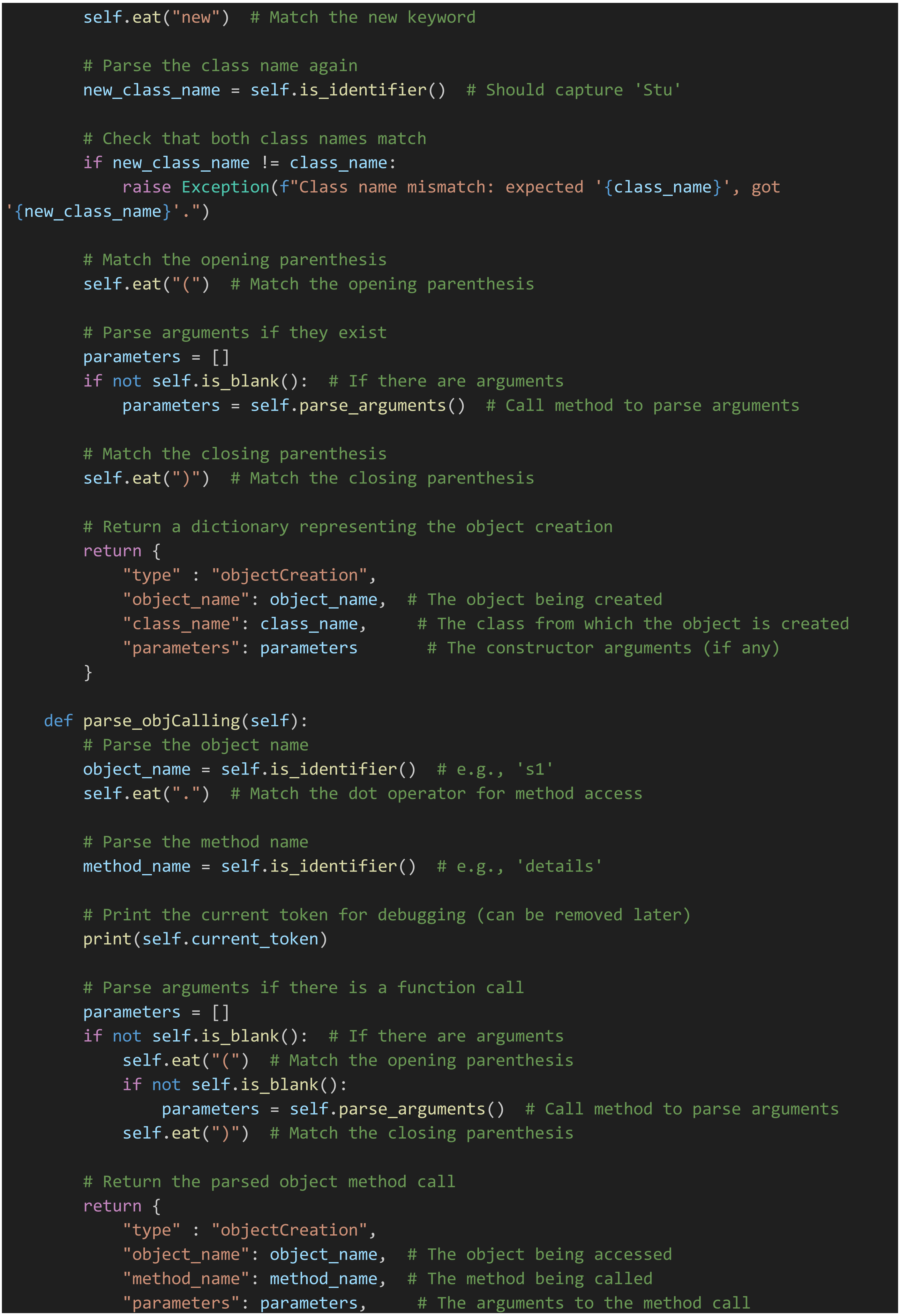
```
        self.eat(")")  # Match the closing parenthesis
        return {"type": "print", "value": display}

    def parse_input(self):
        self.eat("input")  # Match the "input" keyword
        self.eat("(")      # Match the opening parenthesis

        if self.current_token.class_part == "StrChar":
            display = self.is_string()  # Parse the identifier
        else:
            display = None  # Handle blank input

        self.eat(")")  # Match the closing parenthesis
        return {"type": "input", "value": display}

    def parse_function(self):
        if self.current_token.class_part in ["AM", "abstract"]:
            AM = self.current_token.value_part
            self.eat(self.current_token.class_part)  # Consume modifier

        if self.current_token.class_part in ["void", "DT"]:
            return_type = self.eat(self.current_token.class_part)  # Consume return type
        else:
            raise Exception("Expected return type got"  , self.current_token.class_part)

        identifier = self.is_identifier()
        self.eat("(")

        parameters = []
        if self.current_token.class_part != ")":  # Check if there are any parameters
            while True:
                # Parse the parameter type
                if self.current_token.class_part in ["DT"]:
                    param_type = self.current_token.value_part
                    self.eat(self.current_token.class_part)  # Consume the type
                else:
                    raise Exception(f"Expected parameter type but found:
{self.current_token.class_part}")

                # Parse parameter name (identifier)
                if self.current_token.class_part == "ID":
                    param_name = self.is_identifier()  # Consume identifier for parameter
name
                else:
                    raise Exception("Expected parameter identifier")

                # Add the parameter as a tuple of (type, name) to the list
                parameters.append((param_type, param_name))

                # Check for more parameters
                if self.current_token.class_part == ",":
                    self.eat(",")  # Consume the comma
                    parameters.append(",")
                elif self.current_token.class_part == ")":
```



```
                    break  # End of parameters list

        self.eat(")")  # Match closing parenthesis

        # Assuming we parse the function body or block after the parameters
        block = self.parse_block()  # This will handle the function body
        return {
            "type": "function",
            "return_type": return_type,
            "name": identifier,
            "parameters": parameters,
            "block": block,
            "access_modifier": AM
        }

    def parse_tryCatch(self):
        self.eat("try")
        try_block = self.parse_block()  # Changed variable name for clarity

        self.eat("catch")
        self.eat("(")
        identifier = self.is_identifier()
        id = self.is_identifier()
        self.eat(")")

        catch_block = self.parse_block()
        final_block = None

        if(self.current_token.class_part == "finally"):
            self.eat("finally")
            final_block = self.parse_block()

        # Return a structured representation of the try-catch
        return {
            "type": "try",
            "{":"{",
            "try_block": try_block,
            "}":"}",
            "type":"catch",
            "(":"(",
            "catch_identifier": identifier,
            ")":")",
            "{":"{",
            "catch_block": catch_block,
            "}":"}",
            "type":"finally",
            "{":"{",
            "final_block": final_block,
            "}":"}"
        }

    def is_identifier(self):
        identifier = self.current_token.value_part
        self.eat("ID")  # Consume identifier token
```



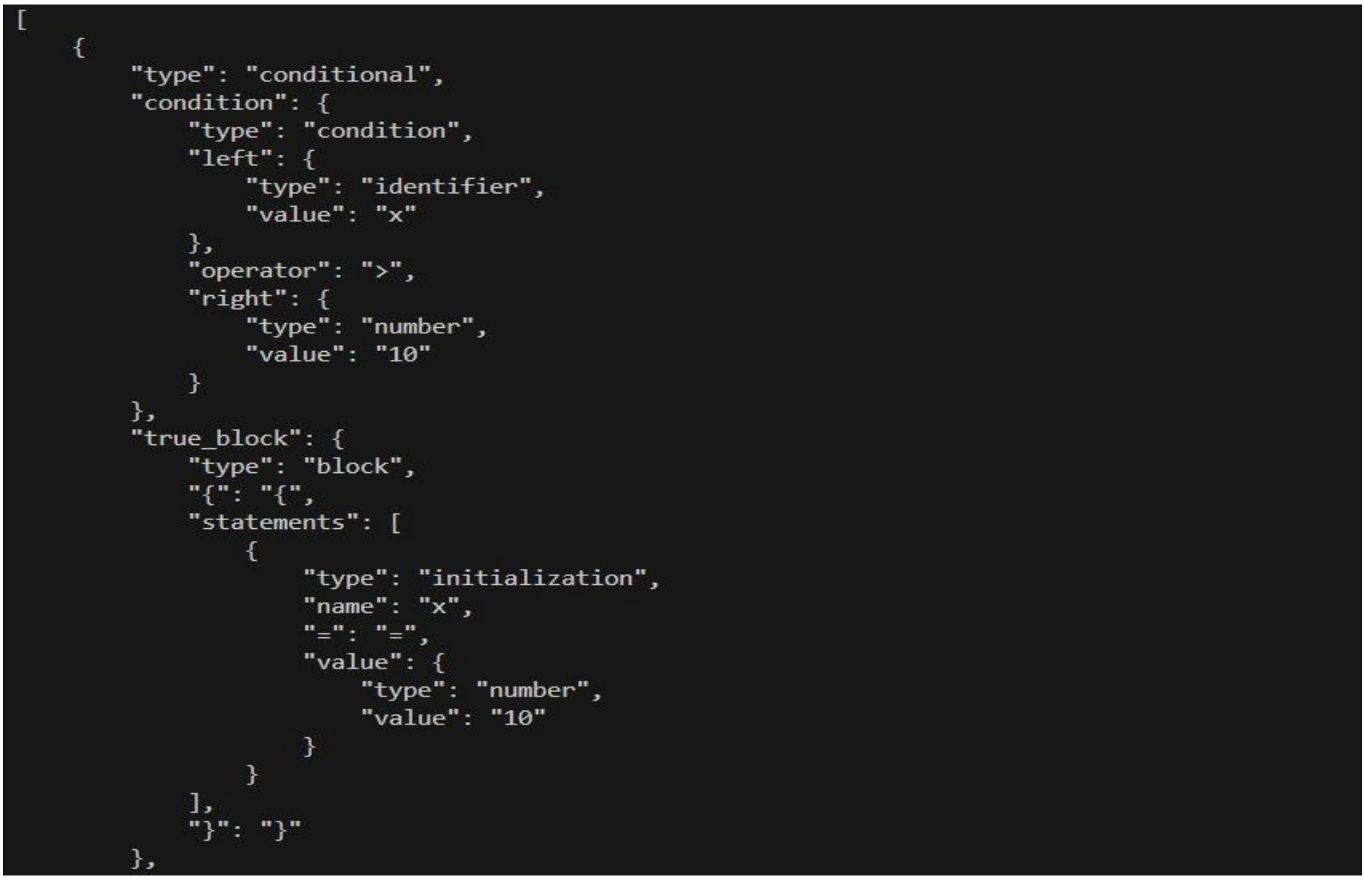
```
        return identifier

    def is_string(self):
        string = self.current_token.value_part
        self.eat("StrChar")  # Consume string token
        return string

    def is_number(self):
        number = self.current_token.value_part
        self.eat("DT")  # Consume number token
        return number

    def parse_block(self):
        """Parse a block of statements."""
        self.eat("{")       # Expecting '{'
        statements = []
        while self.current_token and self.current_token.value_part != "}":
            statements.append(self.parse_statement())

        self.eat("}")       # Expecting '}'
        return {'type': 'block', "{":"{",'statements': statements,"}":"}"}

    def parse_childClass(self):
         # This assumes self.eat() verifies and consumes the next token.
        if self.current_token.class_part in ["AM"]:
            AM= self.current_token.value_part
            self.eat(self.current_token.class_part)
        self.eat("class")  # Match 'class' keyword
        # Parse the class name, assuming self.identifier() will handle extracting a valid
identifier
        class_name = self.is_identifier()
        self.eat("extends")
        parent_class=self.is_identifier()
        # Parse the body or block inside the class definition
        block = self.parse_block()  # Assuming parse_block() is a method that handles
parsing statements inside the class
        # Return the class name and the parsed block as part of the parsed result
        return {"type": "childClass","AM":AM,"child_class": class_name,
"extends":"extends","parent_class":parent_class,"block": block}

    def parse_objects(self):
        # Parse the class name
        class_name = self.is_identifier()

        object_name = self.is_identifier()

        # Check for '='
        if self.current_token.value_part != '=':
            raise Exception(f"Expected '=', got {self.current_token.value_part}.")
        self.eat("=")  # Match the = operator

        # Check for 'new'
        if self.current_token.value_part != 'new':
            raise Exception(f"Expected 'new', got {self.current_token.value_part}.")
```



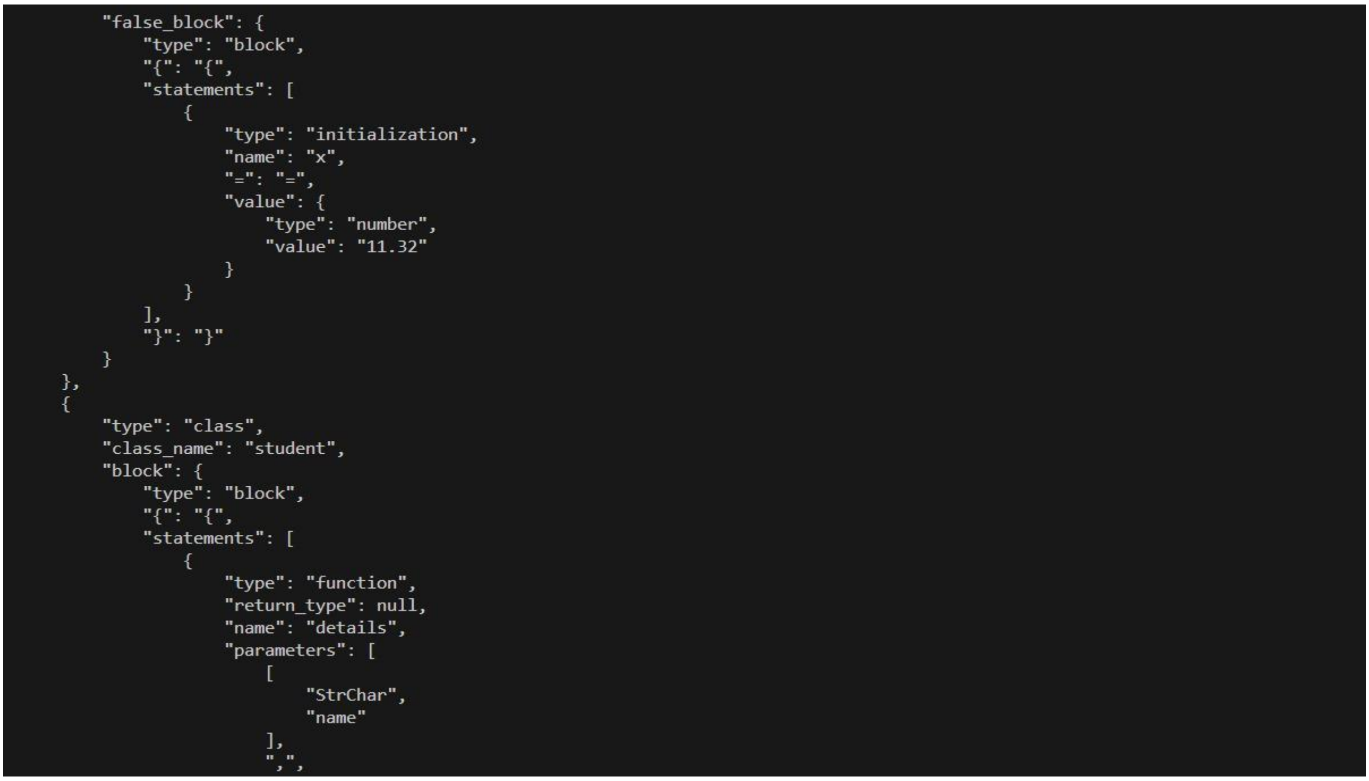
```
        self.eat("new")  # Match the new keyword

        # Parse the class name again
        new_class_name = self.is_identifier()  # Should capture 'Stu'

        # Check that both class names match
        if new_class_name != class_name:
            raise Exception(f"Class name mismatch: expected '{class_name}', got
'{new_class_name}'.")

        # Match the opening parenthesis
        self.eat("(")  # Match the opening parenthesis

        # Parse arguments if they exist
        parameters = []
        if not self.is_blank():  # If there are arguments
            parameters = self.parse_arguments()  # Call method to parse arguments

        # Match the closing parenthesis
        self.eat(")")  # Match the closing parenthesis

        # Return a dictionary representing the object creation
        return {
            "type" : "objectCreation",
            "object_name": object_name,  # The object being created
            "class_name": class_name,     # The class from which the object is created
            "parameters": parameters       # The constructor arguments (if any)
        }

    def parse_objCalling(self):
        # Parse the object name
        object_name = self.is_identifier()  # e.g., 's1'
        self.eat(".")  # Match the dot operator for method access

        # Parse the method name
        method_name = self.is_identifier()  # e.g., 'details'

        # Print the current token for debugging (can be removed later)
        print(self.current_token)

        # Parse arguments if there is a function call
        parameters = []
        if not self.is_blank():  # If there are arguments
            self.eat("(")  # Match the opening parenthesis
            if not self.is_blank():
                parameters = self.parse_arguments()  # Call method to parse arguments
            self.eat(")")  # Match the closing parenthesis

        # Return the parsed object method call
        return {
            "type" : "objectCreation",
            "object_name": object_name,  # The object being accessed
            "method_name": method_name,  # The method being called
            "parameters": parameters,     # The arguments to the method call
```

```python
        }

    def parse_parameter(self):
        # This method parses an individual expression; handle literals and identifiers
        if self.is_identifier():
            return self.is_identifier()  # Return the identifier

        if self.is_literal():  # Check if the current token is a literal
            return self.is_string()  # Return the literal value

        raise Exception(f"Unexpected token: {self.current_token.value_part}")

    def parse_arguments(self):
        arguments = []

        while True:
            # Parse each argument
            argument = self.parse_parameter()  # Parse a single expression (identifier/literal)
            arguments.append(argument)  # Add the argument to the list

            # Check for a comma to continue parsing more arguments
            if self.current_token.value_part == ',':
                self.eat(",")  # Match the comma
            else:
                break  # Exit the loop if there are no more arguments

        return arguments

    def is_blank(self):
        if(self.current_token==")"):
            return 0
        return 1

    def peek(self):
        if self.current_index + 1 < len(self.tokens):
            return self.tokens[self.current_index + 1].value_part
        return None

# Initialize and run parser on token list
parser = Parser(tokens)
ast = parser.parse_program()

# Print and visualize the AST
print(json.dumps(ast, indent=4))
```

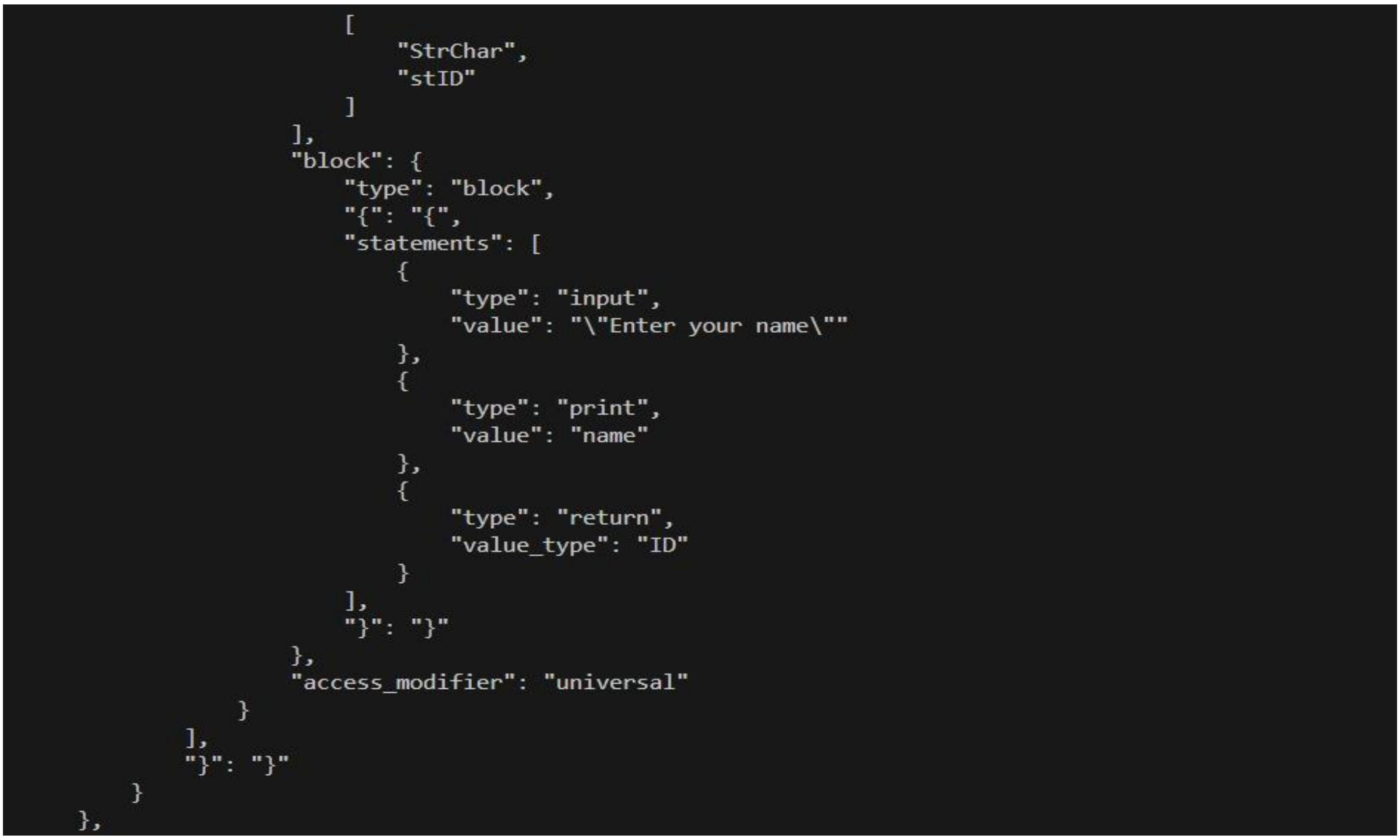
OUTPUT:



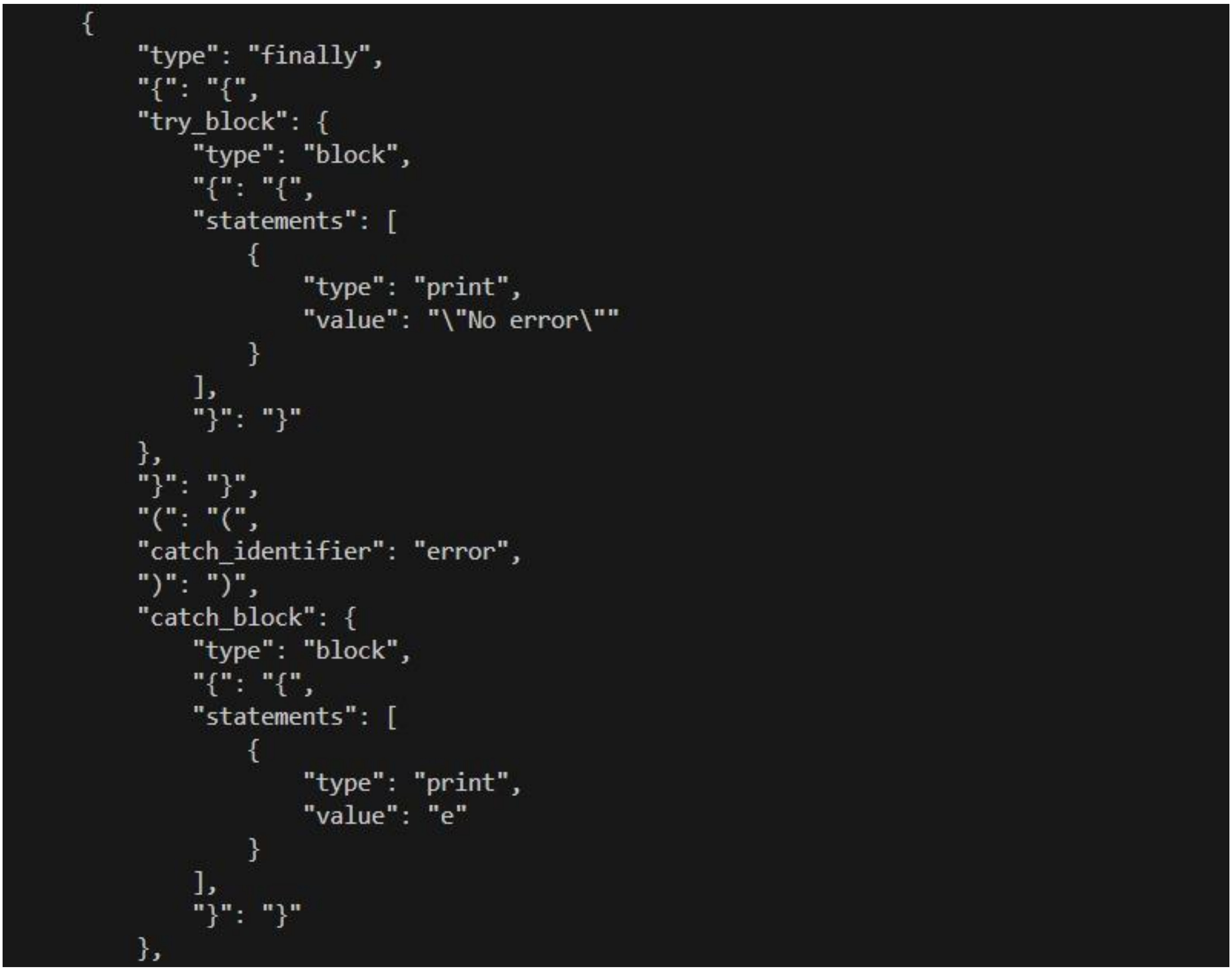
```
[
    {
        "type": "conditional",
        "condition": {
            "type": "condition",
            "left": {
                "type": "identifier",
                "value": "x"
            },
            "operator": ">",
            "right": {
                "type": "number",
                "value": "10"
            }
        },
        "true_block": {
            "type": "block",
            "{": "{",
            "statements": [
                {
                    "type": "initialization",
                    "name": "x",
                    "=": "=",
                    "value": {
                        "type": "number",
                        "value": "10"
                    }
                }
            ],
            "}": "}"
        },
```



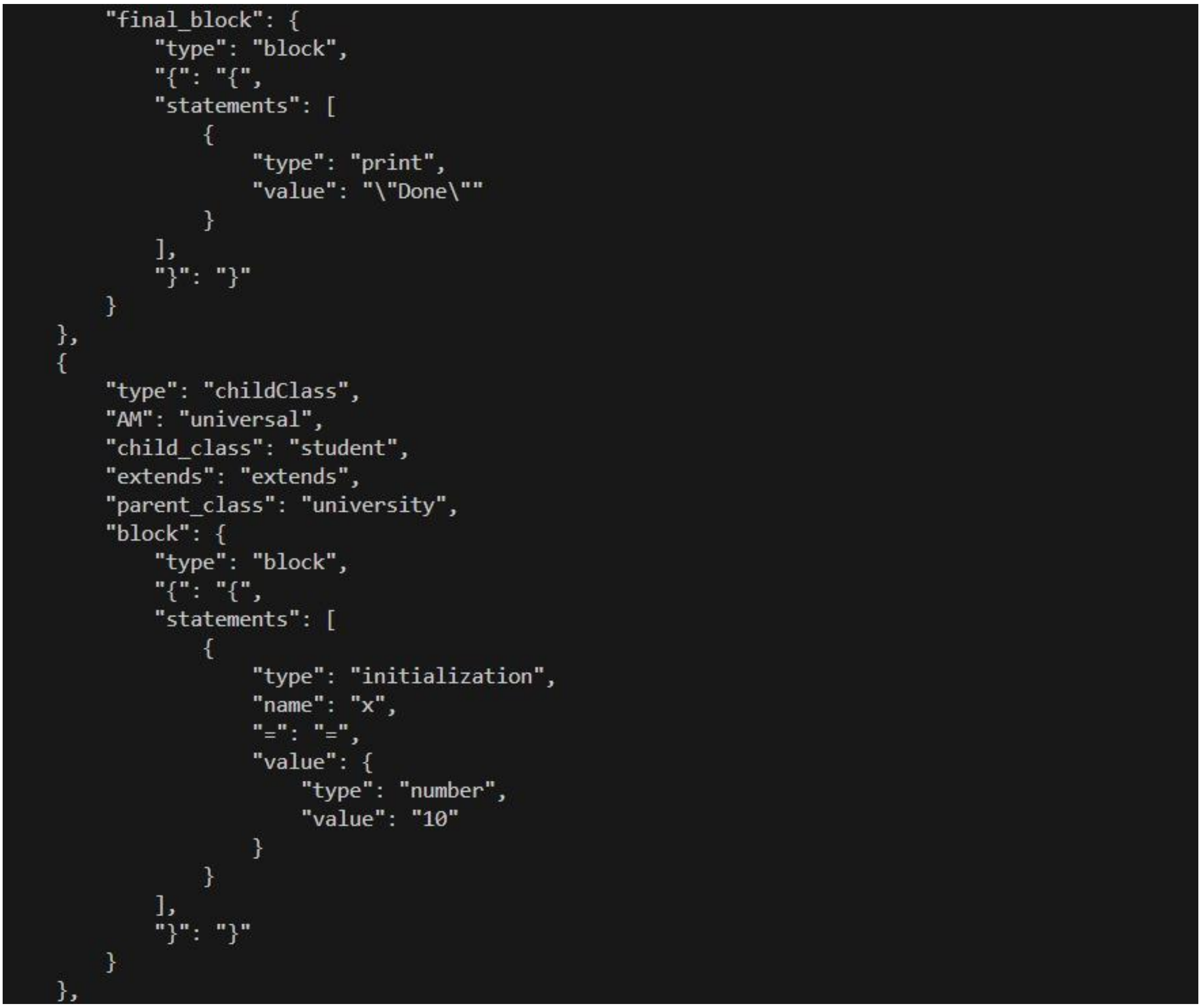
```
        "false_block": {
            "type": "block",
            "{": "{",
            "statements": [
                {
                    "type": "initialization",
                    "name": "x",
                    "=": "=",
                    "value": {
                        "type": "number",
                        "value": "11.32"
                    }
                }
            ],
            "}": "}"
        }
    },
    {
        "type": "class",
        "class_name": "student",
        "block": {
            "type": "block",
            "{": "{",
            "statements": [
                {
                    "type": "function",
                    "return_type": null,
                    "name": "details",
                    "parameters": [
                        [
                            "StrChar",
                            "name"
                        ],
                        ",",
```



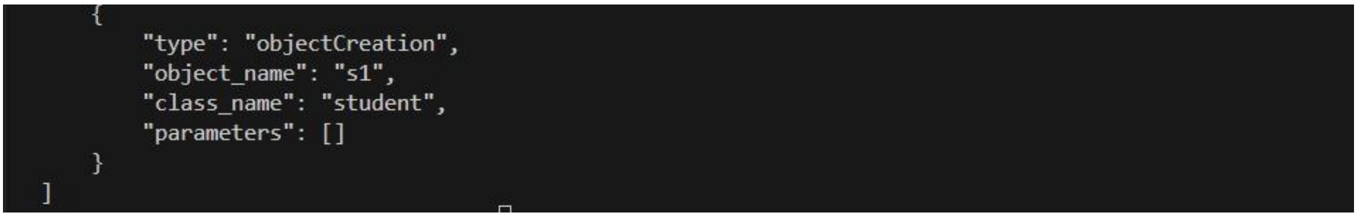
```
                    [
                        "StrChar",
                        "stID"
                    ]
                ],
                "block": {
                    "type": "block",
                    "{": "{",
                    "statements": [
                        {
                            "type": "input",
                            "value": "\"Enter your name\""
                        },
                        {
                            "type": "print",
                            "value": "name"
                        },
                        {
                            "type": "return",
                            "value_type": "ID"
                        }
                    ],
                    "}": "}"
                },
                "access_modifier": "universal"
            }
        ],
        "}": "}"
    }
},
```



```
{
    "type": "finally",
    "{": "{",
    "try_block": {
        "type": "block",
        "{": "{",
        "statements": [
            {
                "type": "print",
                "value": "\"No error\""
            }
        ],
        "}": "}"
    },
    "}": "}",
    "(": "(",
    "catch_identifier": "error",
    ")": ")",
    "catch_block": {
        "type": "block",
        "{": "{",
        "statements": [
            {
                "type": "print",
                "value": "e"
            }
        ],
        "}": "}"
    },
```



```
        "final_block": {
            "type": "block",
            "{": "{",
            "statements": [
                {
                    "type": "print",
                    "value": "\"Done\""
                }
            ],
            "}": "}"
        }
    },
    {
        "type": "childClass",
        "AM": "universal",
        "child_class": "student",
        "extends": "extends",
        "parent_class": "university",
        "block": {
            "type": "block",
            "{": "{",
            "statements": [
                {
                    "type": "initialization",
                    "name": "x",
                    "=": "=",
                    "value": {
                        "type": "number",
                        "value": "10"
                    }
                }
            ],
            "}": "}"
        }
    },
```

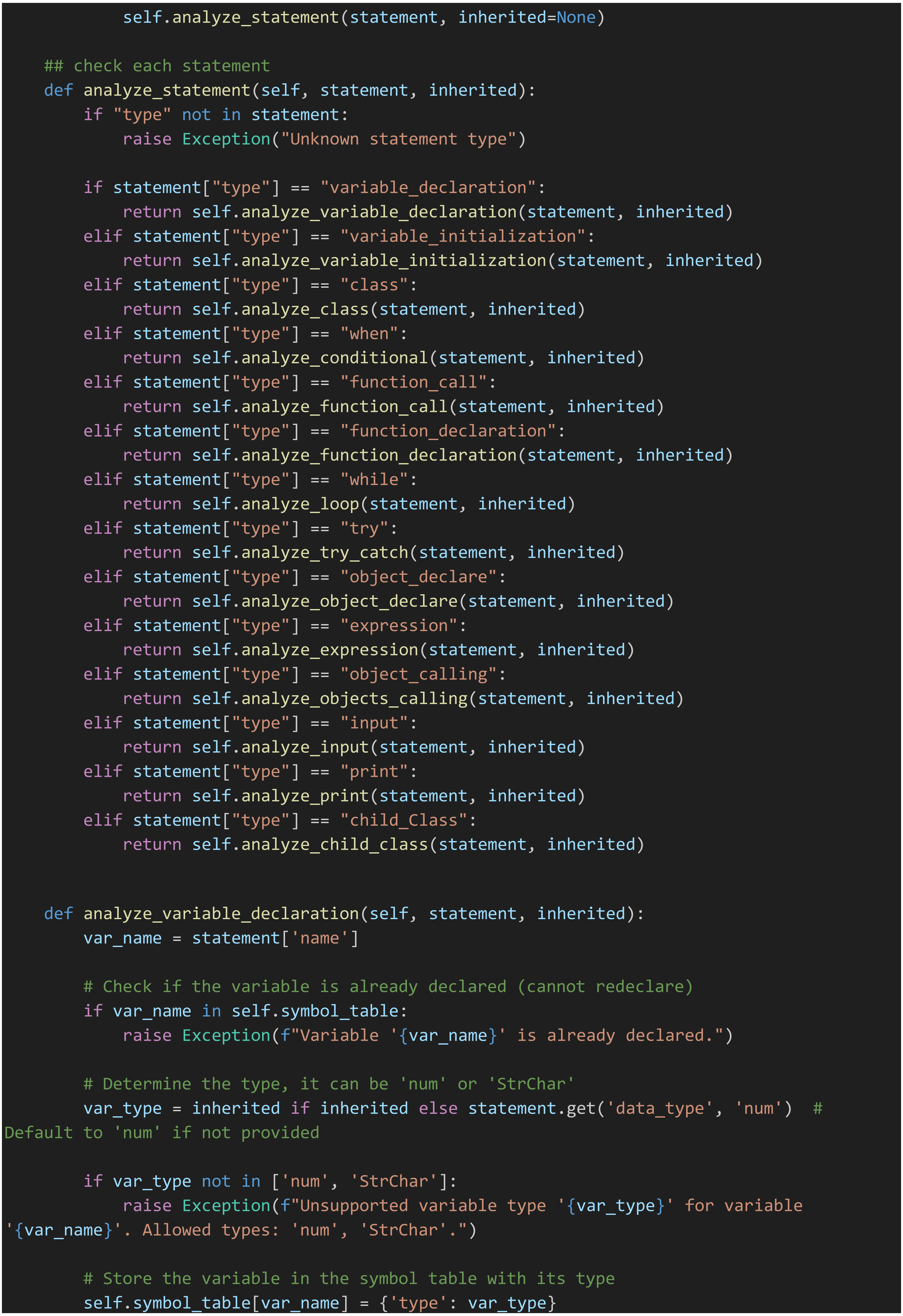


```
    {
        "type": "objectCreation",
        "object_name": "s1",
        "class_name": "student",
        "parameters": []
    }
]
```

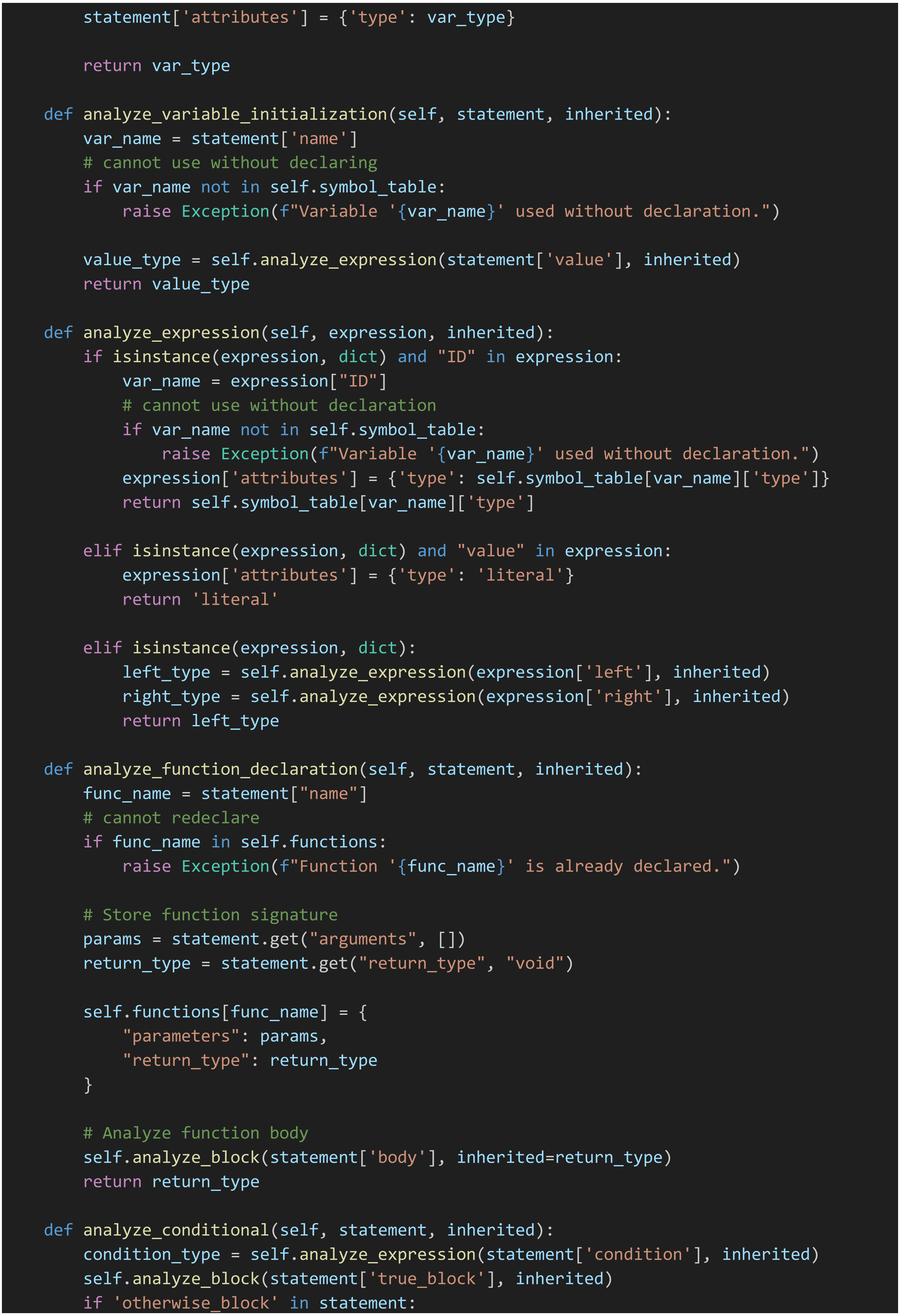
SEMANTIC ANALYZER

CODE:

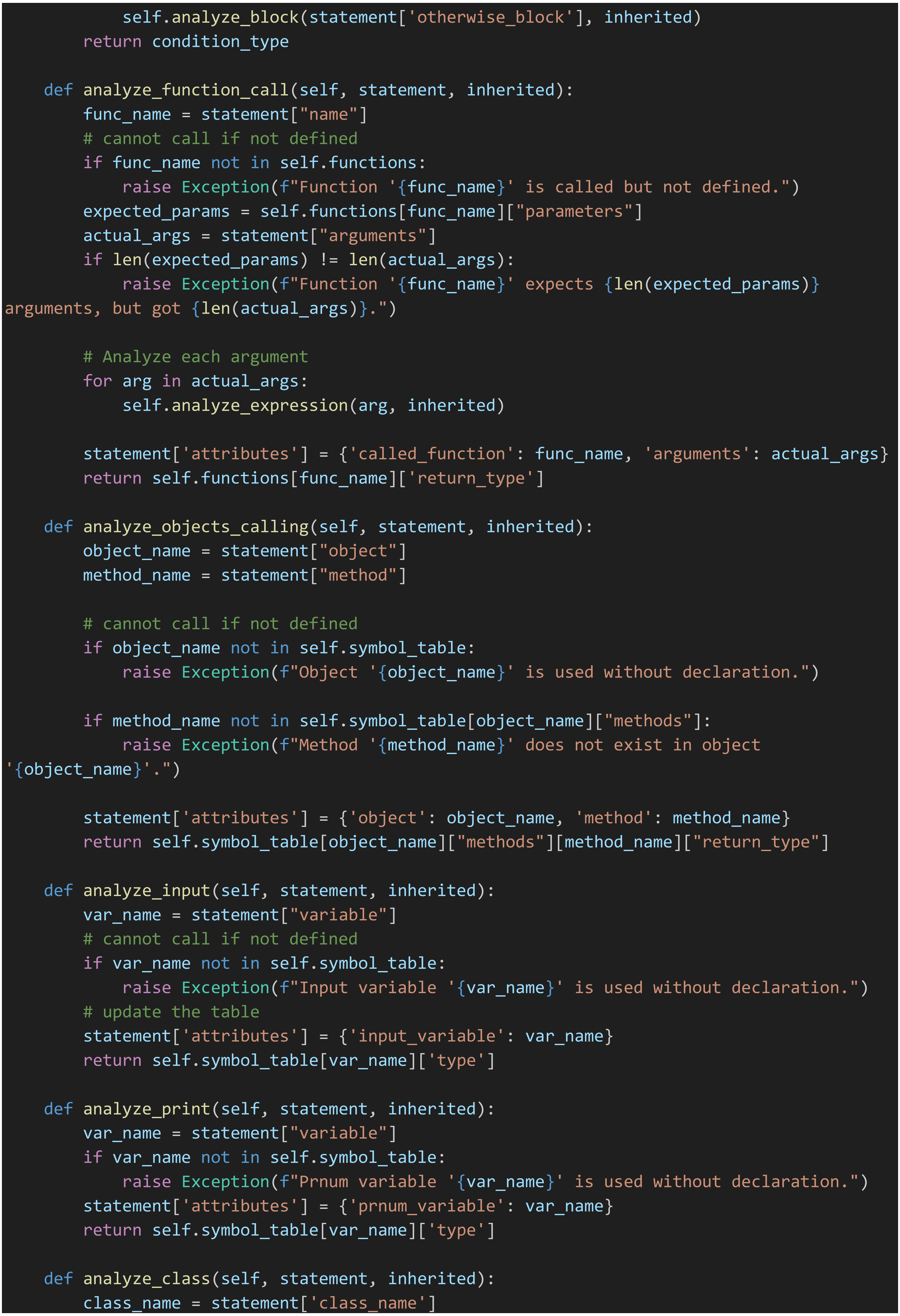
```
class SemanticAnalyzer:
    def __init__(self, ast):
        self.ast = ast  # Abstract syntax tree
        self.symbol_table = {}  # Store declared variables and their types
        self.functions = {}  # Store declared functions
        self.classes = {}  # Store declared classes
        self.current_scope = None  # Track the current scope

    def analyze(self):
        ## check the complete ast
        for statement in self.ast:
```



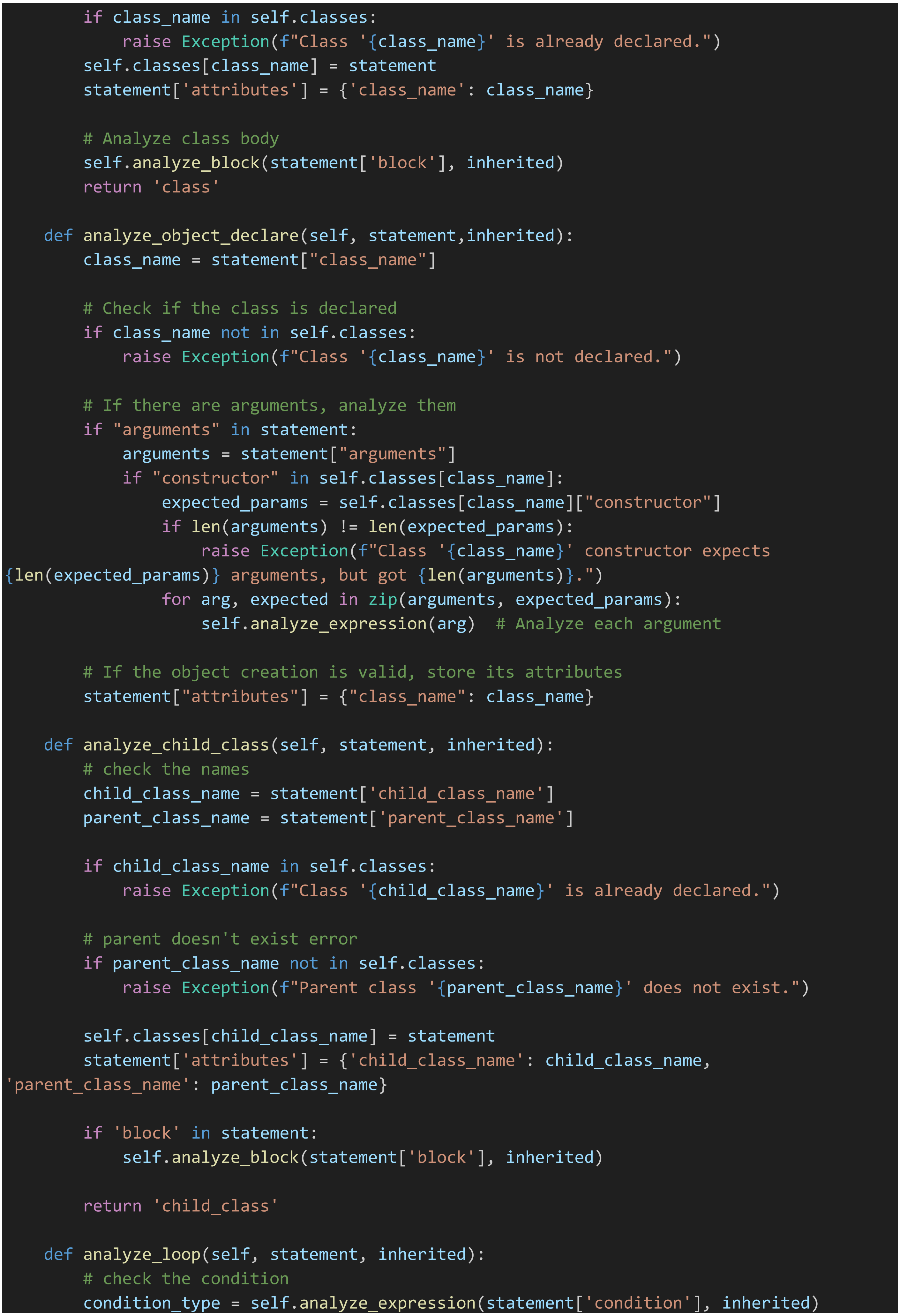
```
            self.analyze_statement(statement, inherited=None)

    ## check each statement
    def analyze_statement(self, statement, inherited):
        if "type" not in statement:
            raise Exception("Unknown statement type")

        if statement["type"] == "variable_declaration":
            return self.analyze_variable_declaration(statement, inherited)
        elif statement["type"] == "variable_initialization":
            return self.analyze_variable_initialization(statement, inherited)
        elif statement["type"] == "class":
            return self.analyze_class(statement, inherited)
        elif statement["type"] == "when":
            return self.analyze_conditional(statement, inherited)
        elif statement["type"] == "function_call":
            return self.analyze_function_call(statement, inherited)
        elif statement["type"] == "function_declaration":
            return self.analyze_function_declaration(statement, inherited)
        elif statement["type"] == "while":
            return self.analyze_loop(statement, inherited)
        elif statement["type"] == "try":
            return self.analyze_try_catch(statement, inherited)
        elif statement["type"] == "object_declare":
            return self.analyze_object_declare(statement, inherited)
        elif statement["type"] == "expression":
            return self.analyze_expression(statement, inherited)
        elif statement["type"] == "object_calling":
            return self.analyze_objects_calling(statement, inherited)
        elif statement["type"] == "input":
            return self.analyze_input(statement, inherited)
        elif statement["type"] == "print":
            return self.analyze_print(statement, inherited)
        elif statement["type"] == "child_Class":
            return self.analyze_child_class(statement, inherited)


    def analyze_variable_declaration(self, statement, inherited):
        var_name = statement['name']

        # Check if the variable is already declared (cannot redeclare)
        if var_name in self.symbol_table:
            raise Exception(f"Variable '{var_name}' is already declared.")

        # Determine the type, it can be 'num' or 'StrChar'
        var_type = inherited if inherited else statement.get('data_type', 'num')  #
Default to 'num' if not provided

        if var_type not in ['num', 'StrChar']:
            raise Exception(f"Unsupported variable type '{var_type}' for variable
'{var_name}'. Allowed types: 'num', 'StrChar'.")

        # Store the variable in the symbol table with its type
        self.symbol_table[var_name] = {'type': var_type}
```



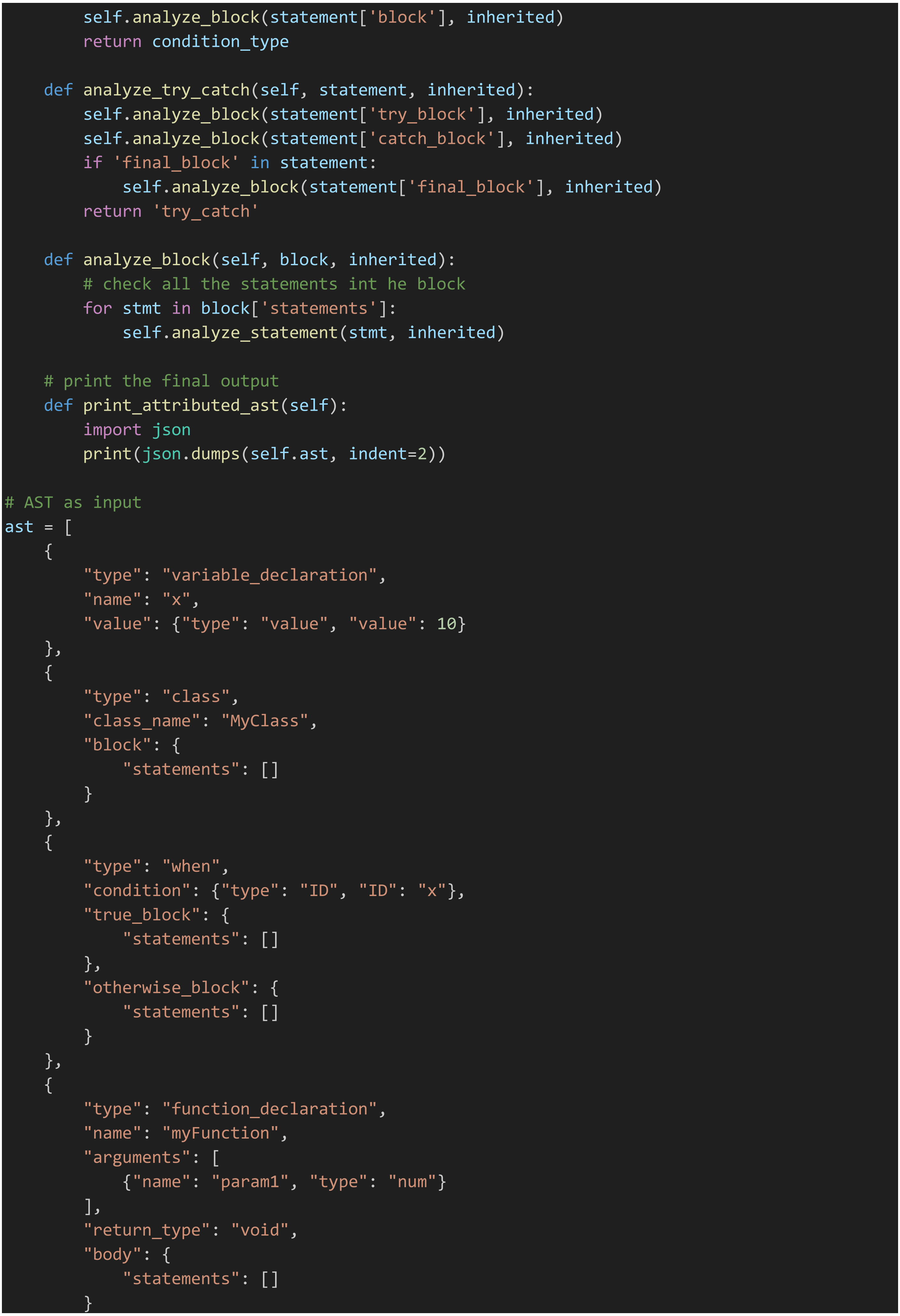
```python
        statement['attributes'] = {'type': var_type}

        return var_type

    def analyze_variable_initialization(self, statement, inherited):
        var_name = statement['name']
        # cannot use without declaring
        if var_name not in self.symbol_table:
            raise Exception(f"Variable '{var_name}' used without declaration.")

        value_type = self.analyze_expression(statement['value'], inherited)
        return value_type

    def analyze_expression(self, expression, inherited):
        if isinstance(expression, dict) and "ID" in expression:
            var_name = expression["ID"]
            # cannot use without declaration
            if var_name not in self.symbol_table:
                raise Exception(f"Variable '{var_name}' used without declaration.")
            expression['attributes'] = {'type': self.symbol_table[var_name]['type']}
            return self.symbol_table[var_name]['type']

        elif isinstance(expression, dict) and "value" in expression:
            expression['attributes'] = {'type': 'literal'}
            return 'literal'

        elif isinstance(expression, dict):
            left_type = self.analyze_expression(expression['left'], inherited)
            right_type = self.analyze_expression(expression['right'], inherited)
            return left_type

    def analyze_function_declaration(self, statement, inherited):
        func_name = statement["name"]
        # cannot redeclare
        if func_name in self.functions:
            raise Exception(f"Function '{func_name}' is already declared.")

        # Store function signature
        params = statement.get("arguments", [])
        return_type = statement.get("return_type", "void")

        self.functions[func_name] = {
            "parameters": params,
            "return_type": return_type
        }

        # Analyze function body
        self.analyze_block(statement['body'], inherited=return_type)
        return return_type

    def analyze_conditional(self, statement, inherited):
        condition_type = self.analyze_expression(statement['condition'], inherited)
        self.analyze_block(statement['true_block'], inherited)
        if 'otherwise_block' in statement:
```



```
            self.analyze_block(statement['otherwise_block'], inherited)
        return condition_type

    def analyze_function_call(self, statement, inherited):
        func_name = statement["name"]
        # cannot call if not defined
        if func_name not in self.functions:
            raise Exception(f"Function '{func_name}' is called but not defined.")
        expected_params = self.functions[func_name]["parameters"]
        actual_args = statement["arguments"]
        if len(expected_params) != len(actual_args):
            raise Exception(f"Function '{func_name}' expects {len(expected_params)}
arguments, but got {len(actual_args)}.")

        # Analyze each argument
        for arg in actual_args:
            self.analyze_expression(arg, inherited)

        statement['attributes'] = {'called_function': func_name, 'arguments': actual_args}
        return self.functions[func_name]['return_type']

    def analyze_objects_calling(self, statement, inherited):
        object_name = statement["object"]
        method_name = statement["method"]

        # cannot call if not defined
        if object_name not in self.symbol_table:
            raise Exception(f"Object '{object_name}' is used without declaration.")

        if method_name not in self.symbol_table[object_name]["methods"]:
            raise Exception(f"Method '{method_name}' does not exist in object
'{object_name}'.")

        statement['attributes'] = {'object': object_name, 'method': method_name}
        return self.symbol_table[object_name]["methods"][method_name]["return_type"]

    def analyze_input(self, statement, inherited):
        var_name = statement["variable"]
        # cannot call if not defined
        if var_name not in self.symbol_table:
            raise Exception(f"Input variable '{var_name}' is used without declaration.")
        # update the table
        statement['attributes'] = {'input_variable': var_name}
        return self.symbol_table[var_name]['type']

    def analyze_print(self, statement, inherited):
        var_name = statement["variable"]
        if var_name not in self.symbol_table:
            raise Exception(f"Prnum variable '{var_name}' is used without declaration.")
        statement['attributes'] = {'prnum_variable': var_name}
        return self.symbol_table[var_name]['type']

    def analyze_class(self, statement, inherited):
        class_name = statement['class_name']
```

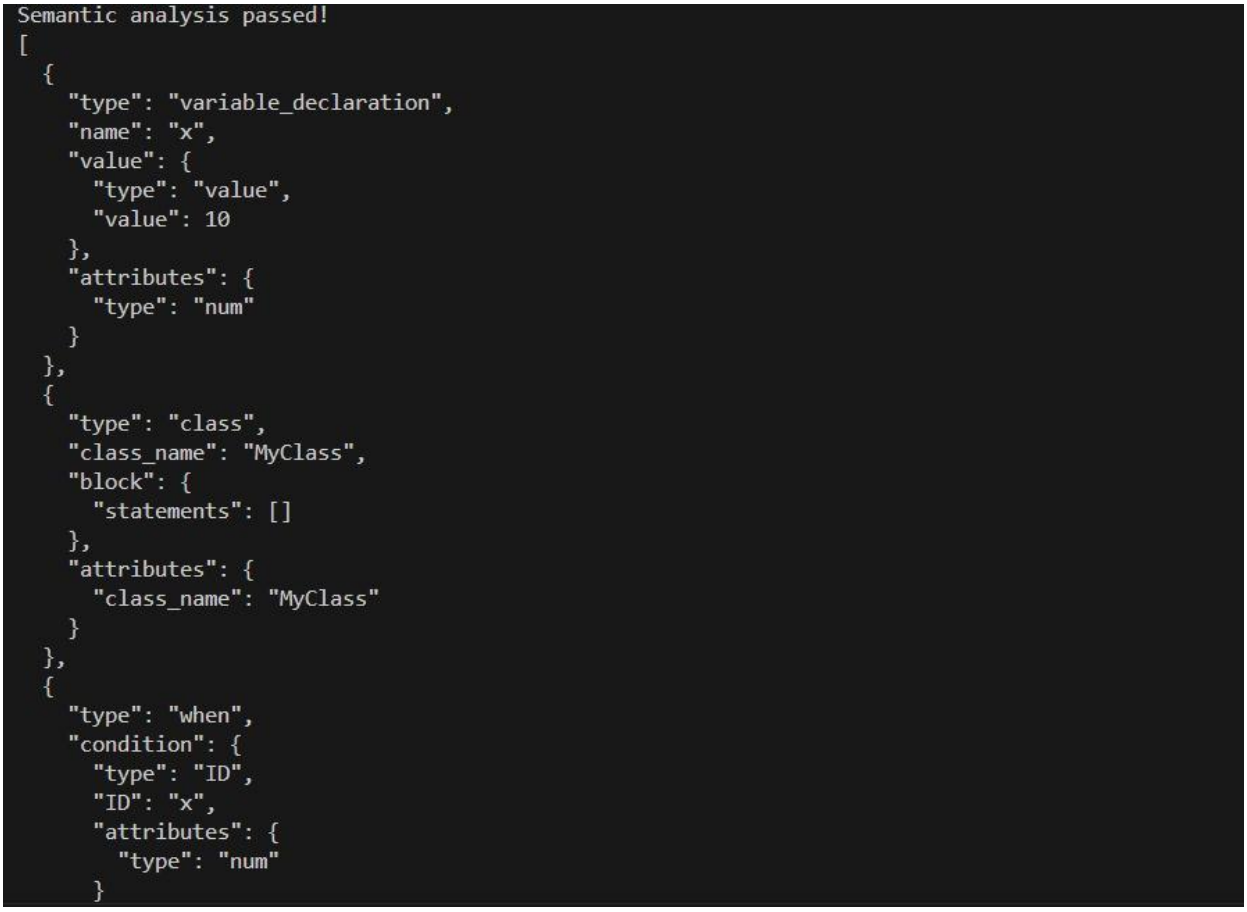


```
        if class_name in self.classes:
            raise Exception(f"Class '{class_name}' is already declared.")
        self.classes[class_name] = statement
        statement['attributes'] = {'class_name': class_name}

        # Analyze class body
        self.analyze_block(statement['block'], inherited)
        return 'class'

    def analyze_object_declare(self, statement,inherited):
        class_name = statement["class_name"]

        # Check if the class is declared
        if class_name not in self.classes:
            raise Exception(f"Class '{class_name}' is not declared.")

        # If there are arguments, analyze them
        if "arguments" in statement:
            arguments = statement["arguments"]
            if "constructor" in self.classes[class_name]:
                expected_params = self.classes[class_name]["constructor"]
                if len(arguments) != len(expected_params):
                    raise Exception(f"Class '{class_name}' constructor expects
{len(expected_params)} arguments, but got {len(arguments)}.")
                for arg, expected in zip(arguments, expected_params):
                    self.analyze_expression(arg)  # Analyze each argument

        # If the object creation is valid, store its attributes
        statement["attributes"] = {"class_name": class_name}

    def analyze_child_class(self, statement, inherited):
        # check the names
        child_class_name = statement['child_class_name']
        parent_class_name = statement['parent_class_name']

        if child_class_name in self.classes:
            raise Exception(f"Class '{child_class_name}' is already declared.")

        # parent doesn't exist error
        if parent_class_name not in self.classes:
            raise Exception(f"Parent class '{parent_class_name}' does not exist.")

        self.classes[child_class_name] = statement
        statement['attributes'] = {'child_class_name': child_class_name,
'parent_class_name': parent_class_name}

        if 'block' in statement:
            self.analyze_block(statement['block'], inherited)

        return 'child_class'

    def analyze_loop(self, statement, inherited):
        # check the condition
        condition_type = self.analyze_expression(statement['condition'], inherited)
```

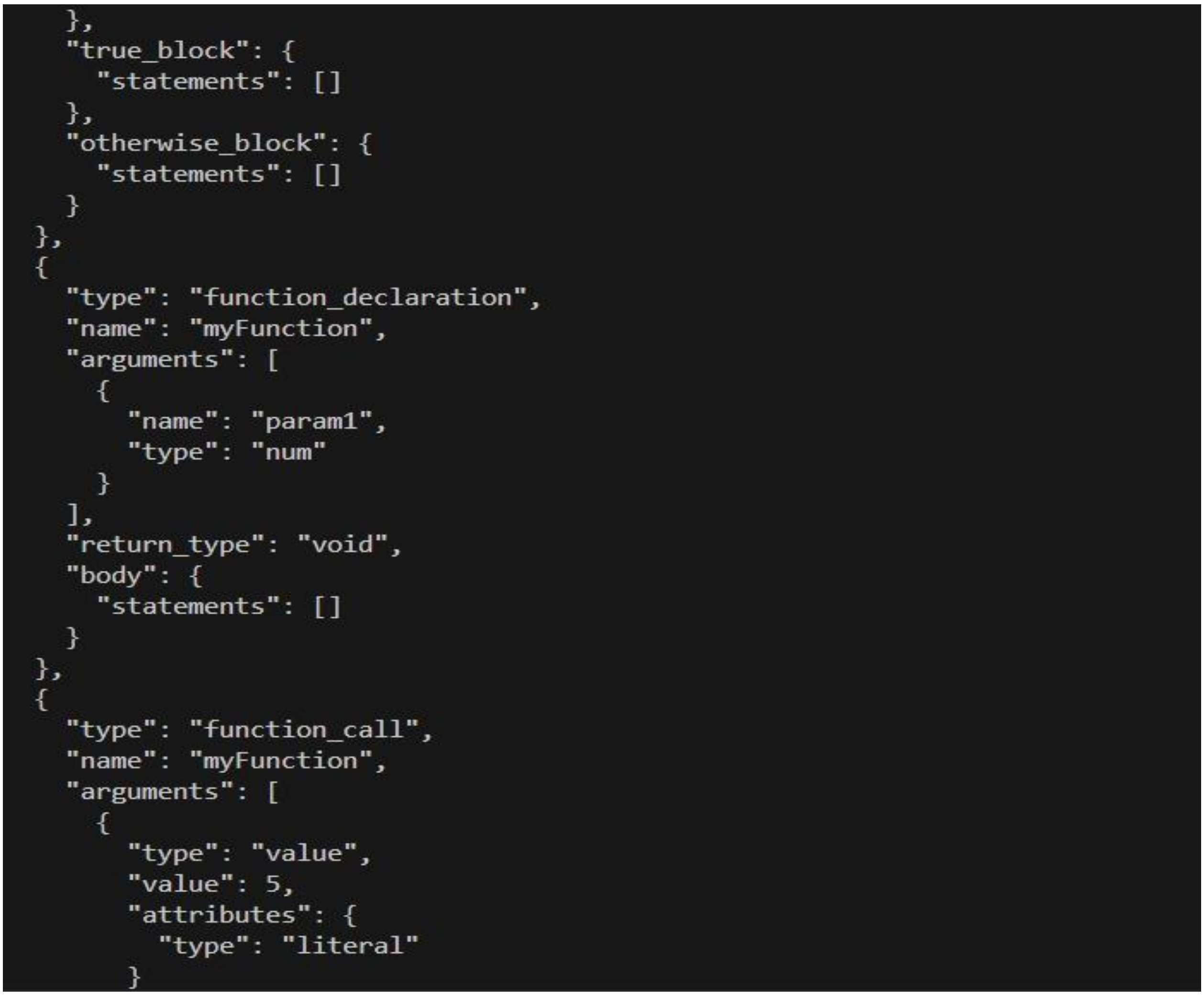


```
        self.analyze_block(statement['block'], inherited)
        return condition_type

    def analyze_try_catch(self, statement, inherited):
        self.analyze_block(statement['try_block'], inherited)
        self.analyze_block(statement['catch_block'], inherited)
        if 'final_block' in statement:
            self.analyze_block(statement['final_block'], inherited)
        return 'try_catch'

    def analyze_block(self, block, inherited):
        # check all the statements int he block
        for stmt in block['statements']:
            self.analyze_statement(stmt, inherited)

    # print the final output
    def print_attributed_ast(self):
        import json
        print(json.dumps(self.ast, indent=2))

# AST as input
ast = [
    {
        "type": "variable_declaration",
        "name": "x",
        "value": {"type": "value", "value": 10}
    },
    {
        "type": "class",
        "class_name": "MyClass",
        "block": {
            "statements": []
        }
    },
    {
        "type": "when",
        "condition": {"type": "ID", "ID": "x"},
        "true_block": {
            "statements": []
        },
        "otherwise_block": {
            "statements": []
        }
    },
    {
        "type": "function_declaration",
        "name": "myFunction",
        "arguments": [
            {"name": "param1", "type": "num"}
        ],
        "return_type": "void",
        "body": {
            "statements": []
        }
    },
    {
        "type": "function_call",
        "name": "myFunction",
        "arguments": [
            {"type": "value", "value": 5}
        ]
    },
    {
        "type": "while",
        "condition": {"type": "value", "value": 1},
        "block": {
            "statements": []
        }
    },
    {
        "type": "try",
        "try_block": {
            "statements": []
        },
        "catch_block": {
            "statements": []
        }
    },
    {
    "type": "object_declare",
    "class_name": "MyClass",
    "arguments": [
        {"type": "value", "value": 10},
        {"type": "value", "value": "example"}
    ]
}
]

analyzer = SemanticAnalyzer(ast)
try:
    analyzer.analyze()
    print("Semantic analysis passed!")
    analyzer.print_attributed_ast()  # Print the attributed AST
except Exception as e:
    print(f"Semantic analysis error: {e}")
```

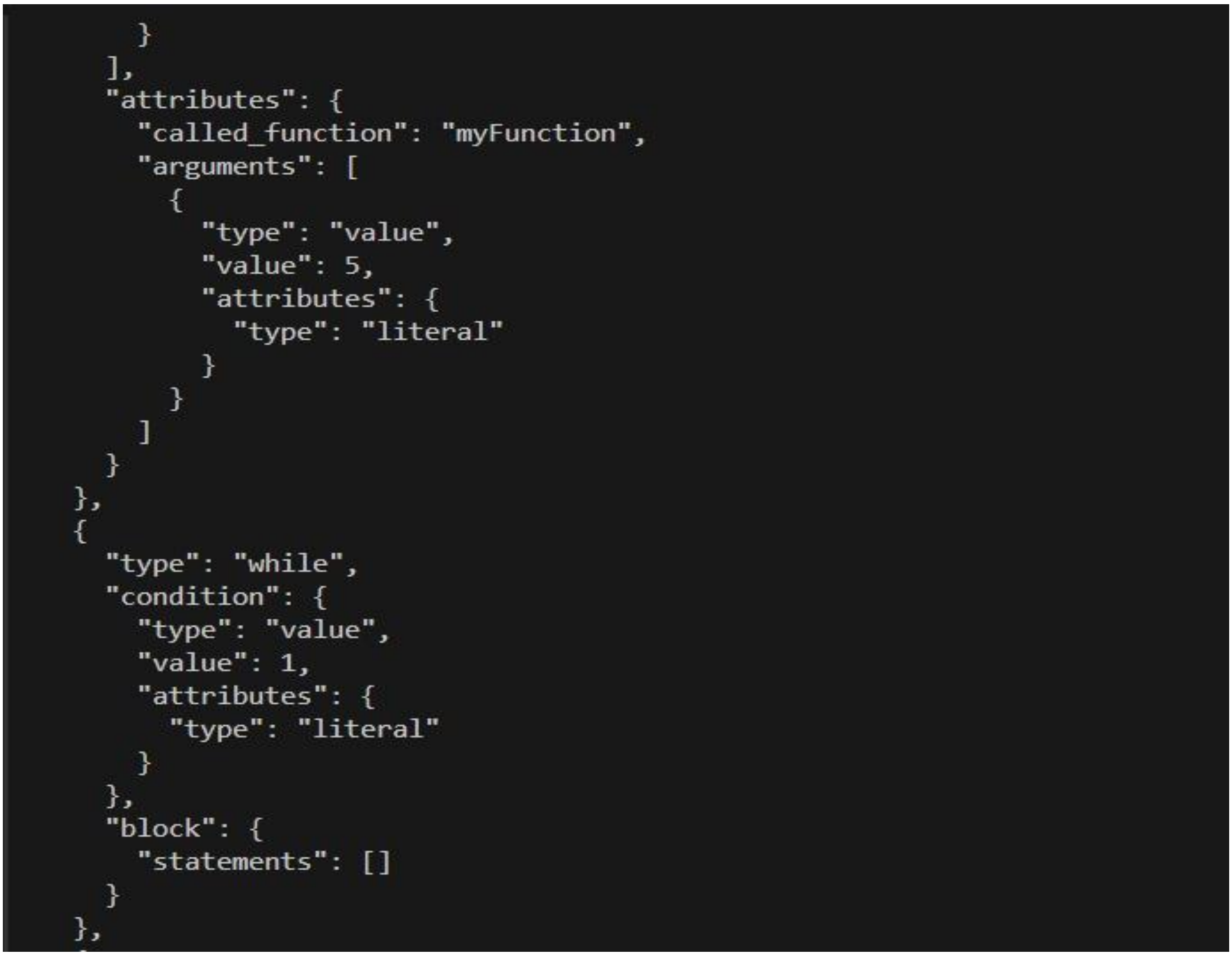
OUTPUT:



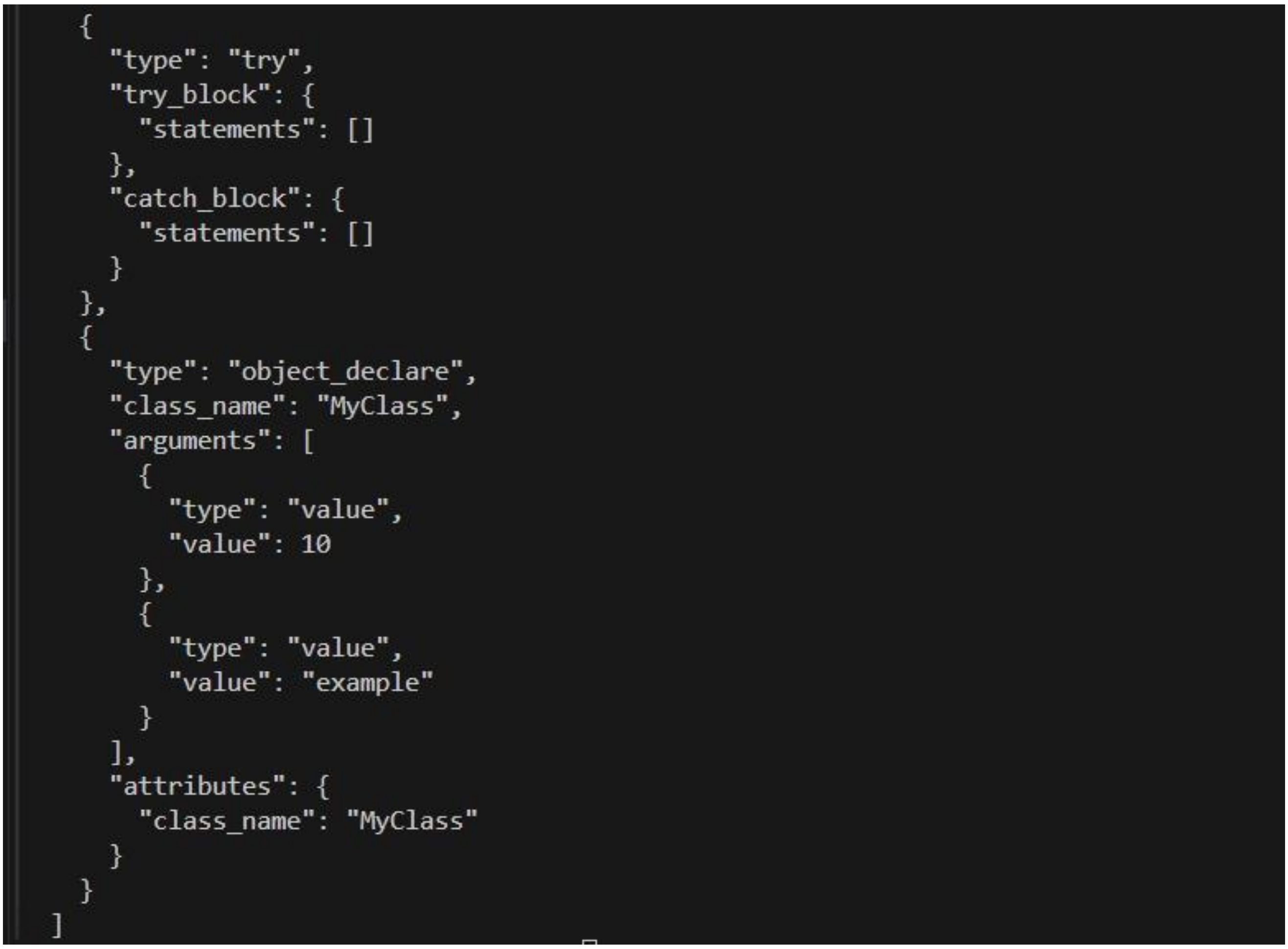
```
Semantic analysis passed!
[
  {
    "type": "variable_declaration",
    "name": "x",
    "value": {
      "type": "value",
      "value": 10
    },
    "attributes": {
      "type": "num"
    }
  },
  {
    "type": "class",
    "class_name": "MyClass",
    "block": {
      "statements": []
    },
    "attributes": {
      "class_name": "MyClass"
    }
  },
  {
    "type": "when",
    "condition": {
      "type": "ID",
      "ID": "x",
      "attributes": {
        "type": "num"
      }
```



```
    },
    "true_block": {
      "statements": []
    },
    "otherwise_block": {
      "statements": []
    }
  },
  {
    "type": "function_declaration",
    "name": "myFunction",
    "arguments": [
      {
        "name": "param1",
        "type": "num"
      }
    ],
    "return_type": "void",
    "body": {
      "statements": []
    }
  },
  {
    "type": "function_call",
    "name": "myFunction",
    "arguments": [
      {
        "type": "value",
        "value": 5,
        "attributes": {
          "type": "literal"
        }
```



```
      }
    ],
    "attributes": {
      "called_function": "myFunction",
      "arguments": [
        {
          "type": "value",
          "value": 5,
          "attributes": {
            "type": "literal"
          }
        }
      ]
    }
  },
  {
    "type": "while",
    "condition": {
      "type": "value",
      "value": 1,
      "attributes": {
        "type": "literal"
      }
    },
    "block": {
      "statements": []
    }
  },
```



```
  {
    "type": "try",
    "try_block": {
      "statements": []
    },
    "catch_block": {
      "statements": []
    }
  },
  {
    "type": "object_declare",
    "class_name": "MyClass",
    "arguments": [
      {
        "type": "value",
        "value": 10
      },
      {
        "type": "value",
        "value": "example"
      }
    ],
    "attributes": {
      "class_name": "MyClass"
    }
  }
]
```