**Explanation of the Recursive-Descent Parser Program-**

We use the following BNF grammar-

```
<expr> ::= <term> | <term> <addop> <expr>
<term> ::= <factor> | <factor> <mulop> <term>
<factor> ::= <digit> | '(' <expr> ')'
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/'
<digit> ::= '0' | '1' | ... | '9'
```

1. We define a class **Parser** that takes an **input expression and splits it into tokens**.
2. The **pos** variable keeps track of the current position in the token list.
3. The **expr *method corresponds to the top-level grammar rule for arithmetic expressions. It first calls the *term** method to parse the left-hand side of the expression, then checks for any addition or subtraction operators and applies them to the result of term and the right-hand side of the expression (which is parsed by calling term again). This process continues as long as there are more operators to apply.
4. The **term** method corresponds to the multiplication and division grammar rule. It works similarly to expr, but with multiplication and division operators.
5. The **factor** method handles parentheses and individual digits. If the current token is **'('**, it recursively calls expr to parse the expression inside the parentheses. If the current token is a **digit**, it converts it to an integer and returns it. Otherwise, it raises a **ValueError**.
6. In the main block, we prompt the user to enter an arithmetic expression and create a Parser object with that input. We then call expr on the Parser object to parse the expression and print the result. If there is a ValueError during parsing, we catch it and print an error message.

```
import re

class Parser:
    def __init__(self, expr):
        self.tokens = re.findall(r'\d+|\+|\-|\*|\/|\(|\)', expr)
        self.pos = 0

    def expr(self):
        result = self.term()
        while self.pos < len(self.tokens) and self.tokens[self.pos] in ('+', '-'):
            op = self.tokens[self.pos]
            self.pos += 1
            rhs = self.term()
            if op == '+':
                result += rhs
            else:
                result -= rhs
        return result

    def term(self):
        result = self.factor()
        while self.pos < len(self.tokens) and self.tokens[self.pos] in ('*', '/'):
            op = self.tokens[self.pos]
            self.pos += 1
            rhs = self.factor()
            if op == '*':
                result *= rhs
            else:
                result /= rhs
        return result

    def factor(self):
        if self.pos < len(self.tokens) and self.tokens[self.pos] == '(':
            self.pos += 1
            result = self.expr()
            if self.pos >= len(self.tokens) or self.tokens[self.pos] != ')':
                raise ValueError('Expected )')
            self.pos += 1
            return result
        elif self.pos < len(self.tokens) and self.tokens[self.pos].isdigit():
            result = int(self.tokens[self.pos])
            self.pos += 1
            return result
        else:
            raise ValueError('Expected digit or (expr)')

if __name__ == '__main__':
    expr = input('Enter an arithmetic expression: ')
    parser = Parser(expr)
    try:
        result = parser.expr()
        print('Result:', result)
    except ValueError as e:
        print('Error:', e)
```

```
Enter an arithmetic expression: 2+-1
Error: Expected digit or (expr)
```

Explanation of the code-

class Parser:

This defines a class called Parser that we will use to parse arithmetic expressions.

```
def __init__(self, expr):
    self.tokens = expr.split()
    self.pos = 0
```

This is the constructor for the Parser class. It takes an arithmetic expression expr as input, splits it into individual tokens using the split() method, and stores the resulting list of tokens in the tokens attribute. It also initializes the pos attribute to 0, which keeps track of the current position in the list of tokens as the parser processes the expression.

```
def expr(self):
    result = self.term()
    while self.pos < len(self.tokens) and self.tokens[self.pos] in ('+', '-'):
        op = self.tokens[self.pos]
        self.pos += 1
        rhs = self.term()
        if op == '+':
            result += rhs
        else:
            result -= rhs
    return result
```

This is the expr() method, which is responsible for parsing an arithmetic expression. It calls the term() method to parse the first term of the expression, then enters a loop that continues as long as there are still tokens in the expression and the current token is a plus or minus sign. In each iteration of the loop, it gets the current operator, advances to the next token, parses the next term, and applies the operator to the result so far and the new term. When the loop is finished, it returns the final result.

```
def term(self):
    result = self.factor()
    while self.pos < len(self.tokens) and self.tokens[self.pos] in ('*', '/'):
        op = self.tokens[self.pos]
        self.pos += 1
        rhs = self.factor()
        if op == '*':
            result *= rhs
        else:
            result /= rhs
    return result
```

This is the term() method, which is responsible for parsing a term in the arithmetic expression. It calls the factor() method to parse the first factor of the term, then enters a loop that continues as long as there are still tokens in the expression and the current token is a multiplication or division sign. In each iteration of the loop, it gets the current operator, advances to the next token, parses the next factor, and applies the operator to the result so far and the new factor. When the loop is finished, it returns the final result.

```
def factor(self):
    if self.pos < len(self.tokens) and self.tokens[self.pos] == '(':
        self.pos += 1
        result = self.expr()
        if self.pos >= len(self.tokens) or self.tokens[self.pos] != ')':
            raise ValueError('Expected )')
        self.pos += 1
        return result
    elif self.pos < len(self.tokens) and self.tokens[self.pos].isdigit():
        result = int(self.tokens[self.pos])
        self.pos += 1
        return result
    else:
        raise ValueError('Expected digit or (expr)')
```

The factor method handles the lowest-level expression units in the arithmetic expression, i.e., factors, which can either be a single digit or an expression enclosed in parentheses. The method first checks if the current token is an opening parenthesis (. If it is, the method recursively calls the expr method to evaluate the expression inside the parentheses. It then checks if the next token is a closing parenthesis ). If it is not, it raises a ValueError with the message "Expected )". If the closing parenthesis is found, the method advances the position counter and returns the result of the evaluated expression inside the parentheses. If the current token is not an opening parenthesis, the method checks if it is a digit using the isdigit method. If it is a digit, it converts the token to an integer and returns it. If the current token is neither an opening parenthesis nor a digit, the method raises a ValueError with the message "Expected digit or (expr)".

if **name** == '**main**': expr = input('Enter an arithmetic expression: ') parser = Parser(expr) try: result = parser.expr() print('Result:', result) except ValueError as e: print('Error:', e)

The if **name** == '**main**': block checks if the module is being run as the main program (as opposed to being imported as a module). If the module is being run as the main program, it prompts the user to enter an arithmetic expression and creates an instance of the Parser class with the input expression. The try block attempts to evaluate the input expression by calling the expr method of the Parser object. If an error occurs during parsing, a ValueError is raised and caught by the except block, which prints an error message along with the specific error message raised during parsing. If parsing is successful, the result is printed.

✓ 4s completed at 10:57 AM