```
# Lab 3B: Recursive descent parser
#    BNF grammar:
#      <expr> -> <term>
#              | <expr> + <term>
#      <term> -> <factor>
#              | <term> *  <factor>
#      <factor> -> digit
#   Assignment:
#   Add logic for the following BNF grammar:
#      <expr> -> <term>
#              | <expr> + <term>
#              | <expr> - <term>
#      <term> -> <factor>
#              | <term> * <factor>
#              | <term> / <factor>
#      <factor> -> digit | (<expr>)
#
# in EBNF grammar:
#    <expr> -> <term> {(+ | -) <term>}
#    <term> -> <factor> {(* | /) <factor>}
#    <factor> -> number | (<expr>)
class LexParse:
    def __init__(self):
        self.cache = {}

    def start_process(self, text):
        self.text = text

        self.pos = -1
        self.len = len(text) - 1
        token = self.start()
        self.assert_end()

        return token

    def assert_end(self):
        if self.pos < self.len:
            raise ParseError(
                self.pos + 1,
                'Expected end of string but got %s',
                self.text[self.pos + 1]
            )

    def skip_whitespace(self):
        while self.pos < self.len and self.text[self.pos + 1] in " \f\v\r\t\n":
            self.pos += 1

    def split_char_ranges(self, chars):
        try:
            return self.cache[chars]
        except KeyError:
            pass
        token = []
        index = 0
        length = len(chars)
        while index < length:
            if index + 2 < length and chars[index + 1] == '-':
                if chars[index] >= chars[index + 2]:
                    raise ValueError('Bad character range')
                token.append(chars[index:index + 3])
                index += 3
            else:
                token.append(chars[index])
                index += 1
        self.cache[chars] = token
        return token

    def char(self, chars=None):
        if self.pos >= self.len:
            raise ParseError(
                self.pos + 1,
                'Expected %s but got end of string',
                'character' if chars is None else '[%s]' % chars
            )
        next_char = self.text[self.pos + 1]
        if chars == None:
            self.pos += 1
            return next_char
        for char_range in self.split_char_ranges(chars):
            if len(char_range) == 1:
                if next_char == char_range:
                    self.pos += 1
                    return next_char
            elif char_range[0] <= next_char <= char_range[2]:
                self.pos += 1
                return next_char
        raise ParseError(
            self.pos + 1,
            'Expected %s but got %s',
            'character' if chars is None else '[%s]' % chars,
            next_char
        )

    def operation(self, *operations):
        self.skip_whitespace()
        if self.pos >= self.len:
            raise ParseError(
                self.pos + 1,
                'Expected %s but got end of string',
                ','.join(operations)
            )
        for operation in operations:
            low = self.pos + 1
            high = low + len(operation)
            if self.text[low:high] == operation:
                self.pos += len(operation)
                self.skip_whitespace()

                return operation
        raise ParseError(
            self.pos + 1,
            'Expected %s but got %s',
            ','.join(operations),
            self.text[self.pos + 1],
        )

    def match(self, *syntax_rules):
        self.skip_whitespace()
        last_error_pos = -1
        last_exception = None
        last_error_syntax_rules = []
        for name in syntax_rules:
            initial_pos = self.pos

            try:
                token_value = getattr(self, name)()
                self.skip_whitespace()
                return token_value
            except ParseError as e:
                self.pos = initial_pos
                if e.pos > last_error_pos:
                    last_exception = e
                    last_error_pos = e.pos
                    last_error_syntax_rules.clear()
                    last_error_syntax_rules.append(rule)
                elif e.pos == last_error_pos:
                    last_error_syntax_rules.append(rule)
        if len(last_error_syntax_rules) == 1:
            raise last_exception
        else:
            raise ParseError(
                last_error_pos,
                'Expected %s but got %s',
                ','.join(last_error_syntax_rules),
                self.text[last_error_pos]
            )

    def maybe_char(self, chars=None):
        try:
            return self.char(chars)
        except ParseError:
            return None

    def maybe_match(self, *syntax_rules):
        try:
            return self.match(*syntax_rules)
        except ParseError:
            return None

    def maybe_operation(self, *operations):
        try:
            return self.operation(*operations)
        except ParseError:
            return None


class ParseError(Exception):
    def __init__(self, pos, msg, *args):
        self.pos = pos
        self.msg = msg
        self.args = args

    def __str__(self):
        return '%s at position %s' % (self.msg % self.args, self.pos)


class CalcParser(LexParse):
```

```
def start(self):
    return self.expression()

def expression(self):

    token_value = self.match('term')

    while True:
        op = self.maybe_operation('+', '-')
        if op is None:
            break
        term = self.match('term')
        if op == '+':
            token_value += term
        elif op == '-':
            token_value -= term

    return token_value

def term(self):
    token_value = self.match('factor')

    while True:
        op = self.maybe_operation('*', '/', ')')
        if op is None or op == ')':
            break
        term = self.match('factor')

        if op == '*':
            token_value *= term
        elif op == '/':
            token_value /= term

    return token_value

def factor(self):
    # your code for the following rule:
    #   <factor>  ->  (<expr>)
    #

    op = self.maybe_operation('(')
    if op is None:
        return self.match('number')
    elif op == '(':
        return self.match('term')

def number(self):
    chars = []
    sign = self.maybe_operation('+', '-')
    if sign is not None:
        chars.append(sign)
    chars.append(self.char('0-9'))
    while True:
        char = self.maybe_char('0-9')
        if char is None:
            break
        chars.append(char)
    if self.maybe_char('.'):
        chars.append('.')
```

```
            chars.append(self.char('0-9'))
            while True:
                char = self.maybe_char('0-9')
                if char is None:
                    break
                chars.append(char)
        token = float(''.join(chars))

        return token


if __name__ == '__main__':
    parser = CalcParser()
    print('Please enter a math expression:')
    print(parser.start_process(input()))
```