https://github.com/iuliaaai/LFTC

```
class State(enumerate):
    NORMAL = 'q'
    ERROR = 'e'
    BACK = 'b'
    FINAL = 'f'


class Configuration:
    def __init__(self, starting_symbol):
        self.state = State.NORMAL
        self.index = 0
        # array containing productions (ex: ('S', ['0', 'B']) or terminals
        self.work_stack = []
        # array containing symbols (terminals or non-terminals)
        self.input_stack = [starting_symbol]


def recursive_descendant(grammar, sequence):
    config = Configuration(grammar.S)
    while config.state != State.FINAL and config.state != State.ERROR:
        if config.state == State.NORMAL:
            if len(config.input_stack) == 0 and config.index == len(sequence):
                success(config)
            elif len(config.input_stack) == 0:
                momentary_insuccess(config)
            else:
                if config.input_stack[0] in grammar.getNonTerminals():
                    expand(config, grammar)
                else:
                    if config.index == len(sequence):
                        momentary_insuccess(config)
                    elif config.input_stack[0] == 'E':
                        config.work_stack.append('E')
                        config.input_stack = config.input_stack[1:]
                    elif config.input_stack[0] == sequence[config.index]:
                        advance(config)
                    else:
                        momentary_insuccess(config)
        else:
            if config.state == State.BACK:
                if config.work_stack[-1] in grammar.E:
                    if config.work_stack[-1] == 'E':
                        config.work_stack.pop(-1)
                    else:
                        back(config)
```

```
            else:
                another_try(config, grammar)

    prod_rules = []
    if config.state == State.ERROR:
        return False, []
    else:
        for prod in config.work_stack:
            if len(prod) > 1:
                if prod[0] in grammar.P.keys():
                    if prod[1] in grammar.P[prod[0]]:
                        prod_rules.append(prod)

    return True, prod_rules


def expand(config, grammar):
    # head of input stack is a non-terminal
    non_term = config.input_stack[0]
    first_prod_rhs = grammar.getProductionsFor(non_term)[0]  # array of symbols
    config.work_stack.append((non_term, first_prod_rhs))
    # remove first elem from input stack and replace it with its production
    config.input_stack = first_prod_rhs + config.input_stack[1:]


def advance(config):
    # head of input stack is a terminal = current symbol from input
    config.index += 1
    config.work_stack.append(config.input_stack[0])
    config.input_stack = config.input_stack[1:]


def momentary_insuccess(config):
    # head of input stack is a terminal ≠ current symbol from input
    config.state = State.BACK


def back(config):
    # head of working stack is a terminal
    config.index -= 1
    terminal = config.work_stack.pop(-1)
    config.input_stack = [terminal] + config.input_stack


def another_try(config, grammar):
    # head of working stack is a non-terminal
    (lhs, rhs) = config.work_stack[-1]
```

```
    productions = [production for production in grammar.getProductionsFor(lhs)]
    next_prod = get_next_production(rhs, productions)
    if next_prod:
        config.state = State.NORMAL
        config.work_stack.pop(-1)
        config.work_stack.append((lhs, next_prod))
        config.input_stack = config.input_stack[len(rhs):]
        config.input_stack = next_prod + config.input_stack
    elif config.index == 0 and lhs == grammar.S:
        config.state = State.ERROR
    else:
        config.work_stack.pop(-1)
        if rhs == ['E']:
            config.input_stack = [lhs] + config.input_stack
        else:
            config.input_stack = [lhs] + config.input_stack[len(rhs):]


def success(config):
    config.state = State.FINAL


# helper function
def get_next_production(prod, prods):
    for i in range(len(prods)):
        if prod == prods[i] and i < len(prods) - 1:
            return prods[i + 1]
    return None
```