```
stack.insert(0, val)
stack.pop(0)
```

```python
def load_file(fname):
    lines = []
    with open(fname) as fd:
        for line in fd:
            #....
            lines.append(processed_line)
    return lines
```

```python
def load_file2(fname):
    with open(fname) as fd:
        for line in fd:
            #....
            yield processed_line
```

```python
def load_file3(fd):
    for line in fd:
        #....
        yield processed_line

def execute(fname):
    for cmd in load_file3(fname):
        #.....
```

```python
class Forth(object):
    stack = []


class Forth(object):
    def __init__(self):
        self._stack = []


class Forth(object):
    def __init__(self, statements):
        self.current_parameter = None

    def execute(self):
        for ....:
            #....
            self.current_parameter = some_data
            run_command
```

```
a, b = self.stack.pop(), self.stack.pop()
a[func1()] = func2()


cmd = s.split()[0]
param = s.split()[1]
```

```python

class Forth(object):
    # ....
    def run(self):
        for s in self._statements:
            if hasattr(self, s.split()[0]):
                method = getattr(self, s.split()[0])
                method()
```

LEAP vs EAFP

```python
def add_LEAP(stack):
    if len(stack) < 2:
        raise SomeNiceClass()
    stack.append(stack.pop() + stack.pop())

def add_EAFP(stack):
    stack.append(stack.pop() + stack.pop())
```

Code duplication

```
def add():
    p1 = convert_to_number(pop())
    p2 = convert_to_number(pop())
    ...

def sub():
    p1 = convert_to_number(pop())
    p2 = convert_to_number(pop())
    ....
```

SOLID (SRP, OCR, LSP, ISP, DIP), YAGNI

```python
class Forth(object):
    def __init__(self, statements):

    def put(self):

    def pop(self):

    def add(self):

    def sub(self):

    def print_(self):

    def run(self):

def execute(file_name):
    Forth(source_code).run()
```

```python
class Forth(__Stack__):
    def put(self):
        pass

    def pop(self):
        pass
```

Lexer => Parser => Compiler => Executor

Lexer + Parser

```python
def parse_file(fd):
    for lineno, raw_line in enumerate(fd):
        line = raw_line.strip()
        if line == "" or line.startswith("#"):
            continue
        try:
            if " " not in line:
                yield lineno, (line, None)
            else:
                cmd, param = line.split(" ", 1)
                if param.startswith('"') and param.endswith(
                    param = param[1:-1]
                else:
                    try:
                        param = int(param)
                    except ValueError:
                        param = float(param)
                yield lineno, (cmd, param)
        except Exception as x:
            print >>sys.stderr, "Parse error at line", linen
            raise
```

Compiler + Executor

```
def execute(stack, fd):
    for lineno, (cmd, param) in parse_file(fd):
        if cmd == "put":
            stack.append(param)
        elif cmd == "add":
            stack.append(stack.pop() + stack.pop())
        elif cmd == "sub":
            stack.append(stack.pop() - stack.pop())
        elif cmd == "print":
            print stack.pop()
        else:
            raise ValueError("Unknown command {} at line {}".for
```

Command

```python
class Command(object):
    name = None
    param_count = None
    minimum_stack_size = None
    def __init__(self, **params):
        self.params = params
    def execute(self, stack):
        if len(stack) < self.minimum_stack_size:

class Add(Command):
    name = 'add'
    param_count = 0
    minimum_stack_size = 2
    def execute(self, stack):
        stack.append(stack.pop() + stack.pop())

all_commands = {Add.name: Add, ...}
```

Command

```
class Command_0_0(object):
    def __init__(self, **params):
        self.name = params

    def execute(self, stack):
        if len(stack) < self.minimum_stack_size:
            .....

class Add_0_2(Command):
    pass

all_command_classes = [Add_0_2, ..]
all_commands = {}
```

Command

```python
for cmd in all_command_classes:
    uname, stack_sz, num_params = cmd.__name__.split("_")
    cmd.name = uname.lower()
    cmd.param_count = int(num_params)
    ...
    all_commands[cmd.name] = cmd
```

Command

```
def add_2_0(stack):
    #....

def add_2_0(stack, s1, s2):
    #...

def put(stack, p1):
    stack.append(p1)
```