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

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

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

```
1 def load_file3(fd):
2     for line in fd:
3         #....
4         yield processed_line
5
6 def execute(fname):
7     for cmd in load_file3(fname):
8         #.....
```

```
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]
```

```
1
2 class Forth(object):
3     # ....
4     def run(self):
5         for s in self._statements:
6             if hasattr(self, s.split()[0]):
7                 method = getattr(self, s.split()[0])
8                 method()
```

LEAP vs EAFP

```
1 def add_LEAP(stack):
2     if len(stack) < 2:
3         raise SomeNiceClass()
4     stack.append(stack.pop() + stack.pop())
5
6 def add_EAFP(stack):
7     stack.append(stack.pop() + stack.pop())
```

Code duplication

```
1 def add():
2     p1 = convert_to_number(pop())
3     p2 = convert_to_number(pop())
4     ...
5
6 def sub():
7     p1 = convert_to_number(pop())
8     p2 = convert_to_number(pop())
9     ....
```

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

```
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()
```

```
1 class Forth(__Stack__):
2     def put(self):
3         pass
4
5     def pop(self):
6         pass
```

Lexer => Parser => Compiler => Executor

Lexer + Parser

```
1       def parse_file(fd):
2           for lineno, raw_line in enumerate(fd):
3               line = raw_line.strip()
4               if line == "" or line.startswith("#"):
5                   continue
6               try:
7                   if " " not in line:
8                       yield lineno, (line, None)
9                   else:
10                      cmd, param = line.split(" ", 1)
11                      if param.startswith('"') and param.endswith(
12                          param = param[1:-1]
13                      else:
14                          try:
15                              param = int(param)
16                          except ValueError:
17                              param = float(param)
```

```
18                      yield lineno, (cmd, param)
19              except Exception as x:
20                  print >>sys.stderr, "Parse error at line", lineno
21                  raise
```

Compiler + Executor

```
1  def execute(stack, fd):
2      for lineno, (cmd, param) in parse_file(fd):
3          if cmd == "put":
4              stack.append(param)
5          elif cmd == "add":
6              stack.append(stack.pop() + stack.pop())
7          elif cmd == "sub":
8              stack.append(stack.pop() - stack.pop())
9          elif cmd == "print":
10             print stack.pop()
11         else:
12             raise ValueError("Unknown command {} at line {}".forn
```

Command

```
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

```
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

```
1 def add_2_0(stack):
2     #....
3
4 def add_2_0(stack, s1, s2):
5     #...
6
7 def put(stack, p1):
8     stack.append(p1)
```