

В одном из прошлых заданий мы [закодировали стек, хранящий числа](#). Сейчас мы придумаем небольшой язык программирования, который будет производить вычисления с этим стеком на манер программируемого калькулятора.

Один из моих студентов почему-то вместо "стек" всегда говорил "стёк". Чтобы увековечить его, я назову наш язык Стёком. Мы сделаем несколько версий языка, постепенно добавляя туда функциональность.

Первые версии мы будем очень наивно интерпретировать, затем мы опишем небольшую виртуальную машину, идейно похожую на виртуальную машину языка Java (JVM) и научимся преобразовывать текстовые программы в специальное промежуточное представление программ – байткод, который уже и будет выполняться на виртуальной машине.

## Стёк 1.0

Программы на языке Стёк 1.0 выполняются последовательно. Они состоят из команд двух видов: одни команды не имеют аргументов, другие имеют один аргумент.

Команды работают со стеком чисел. Вот некоторые команды:

- `PUSH <число>` кладёт число в стек.
- `IADD` вынимает из стека два числа, кладёт в стек результат их сложения.
- `IMUL` вынимает из стека два числа, кладёт в стек результат их умножения.
- `IPRINT` вынимает из стека одно число, выводит его на экран.
- `IREAD` считывает число и кладёт его на стек.
- `DUP` вынимает из стека одно число и кладёт его на вершину дважды.
- `STOP` останавливает программу.

Стек мы изображаем в скобках; в квадратных скобках мы даём описание одного элемента стека, полученного с помощью вычислений. Чтобы описать изменение стека мы будем писать так:

( верхние значения стека до изменения – верхние значения стека после изменения )

Например, для команды `IADD`: `( a b - [a+b] )`. Здесь `a` и `b` это имена для верхних значений в стеке, которые мы вводим для удобства. В стеке, конечно, могут лежать и другие значения глубже `( a b )`, но они не участвуют в вычислениях.

Попробуйте понять, что делает эта программа, просто посмотрев на неё:

`IREAD`

`IREAD`

`DUP`

`PUSH 10`

`IMUL`

IADD

IADD

IPRINT

STOP

Эта программа считывает со входа два числа, назовём их  $a$  и  $b$ . Проследим за тем, как меняется состояние стека при её выполнении.

IREAD (  $a$  )

IREAD (  $a$   $b$  )

DUP (  $a$   $b$   $b$  )

PUSH 10 (  $a$   $b$   $b$  10 )

IMUL (  $a$   $b$  [ $b*10$ ] )

IADD (  $a$  [ $b + b*10$ ] )

IADD ( [ $a + b + b*10$ ] )

IPRINT ( ) вывод [ $a + b + b*10$ ]

STOP

## Список команд Стёка

Вот полный список команд языка Стёк 1.0 и стековые диаграммы для них

- Манипуляции со стеком
  - PUSH  $n$  (  $- n$  ) кладёт число в стек.
  - POP (  $a -$  ) вынимает из стека одно число, ничего с ним не делает.
  - SWAP (  $a$   $b - b$   $a$  )
  - DUP (  $a - a$   $a$  ) вынимает из стека одно число и кладёт его на вершину дважды.
- Арифметика:
  - IADD (  $a$   $b - [a + b]$  )
  - ISUB (  $a$   $b - [a - b]$  ) (обратите внимание на порядок операндов! Сначала кладем в стек то, из чего вычитаем)
  - IMUL (  $a$   $b - [a * b]$  )
  - IDIV (  $a$   $b - [a / b]$  )
  - IMOD (  $a$   $b - [a \% b]$  )
  - INEG (  $a - [-a]$  )
- Чтение и запись
  - IPRINT (  $a -$  ) вынимает из стека одно число, выводит его на экран.

- `IREAD ( - a )` считывает число и кладёт его на стек.
- `ICMP (a b - [ if (a < b) then -1; else if (a > b) then 1; else 0 ] )`
- `STOP`

Мы реализуем интерпретатор этого языка, и напомним его декларативно, красиво и расширяемо. На протяжении этого и следующих уроков мы постепенно будем подбираться к созданию небольшой стековой виртуальной машины наподобие игрушечной JVM.

## Как описать команды Стёка на C?

Мы придумали команды двух форматов:

- без аргументов
- с одним аргументом (пока это только `PUSH`).

Используем наш навык описания сущностей, которые могут быть или чем-то одним, или чем-то другим [с помощью структур, перечислений и объединений](#). Так мы опишем схему, с помощью которой будем задавать программы, состоящие уже из конкретных инструкций.

```
enum opcode {
    BC_PUSH, BC_POP, BC_SWAP, BC_DUP,
    BC_IADD, BC_ISUB, BC_IMUL, BC_IDIV,
    BC_INEG,
    BC_IPRINT, BC_IREAD,
    BC_ICMP,
    BC_STOP
};

// Инструкция в одном из двух форматов: bc_noarg или bc_arg64
struct bc_noarg {
    enum opcode opcode;
};

struct bc_arg64 {
    enum opcode opcode;
    int64_t arg;
};
```

```
};
```

```
union ins {  
  
    enum opcode opcode;  
  
    struct bc_arg64 as_arg64;  
  
    struct bc_noarg as_noarg;  
  
};
```

Программа это массив команд типа `union ins`. Все команды имеют одинаковый размер `sizeof(union ins)`. Точный размер мы можем только предположить, т.к. размер `as_arg64` может быть и 9 байт, и 16 байт из-за [выравнивания](#).

При таком кодировании команд память расходуется неоптимально. Команды без аргументов занимают столько же места, сколько и команды с аргументами, поэтому в командах без аргументов как минимум 8 байт никак не используются. Более того, большинство команд Стёка как раз аргументов не имеют! Мы оптимизируем интерпретатор Стёка для более эффективного использования памяти в следующих версиях.

## Как закодировать программу на Стёке внутри программы на C?

Чтобы начать разрабатывать интерпретатор Стёка нам не потребуется разбирать текстовые файлы и вычленять оттуда команды языка, вроде `PUSH 42` или `DUP`. Закодируем программу как массив `union ins`; это позволит нам прописать программу на Стёке **прямо внутри программы на C**. Можно сказать, что язык Стёк мы *встроим* внутрь языка C: мы будем выражать программу на Стёке используя языковые конструкции C. Встраивание языков друг в друга и [предметно-ориентированные языки](#) часто сопровождают декларативный стиль программирования.

Вот как выглядит программа на Стёке внутри кода на C.

- объединения, соответствующие командам без аргументов, инициализируются тривиально, через фигурные скобки;
- объединения, соответствующие команде `push` с одним аргументом, инициализируется через поле `as_arg64`.

```
/* Код с прошлого шага: */
```

```
enum opcode {  
  
    BC_PUSH, BC_POP, BC_SWAP, BC_DUP,  
  
    BC_IADD, BC_ISUB, BC_IMUL, BC_IDIV,  
  
    BC_INEG,
```

```
BC_IPRINT, BC_IREAD,  
  
BC_ICMP,  
  
BC_STOP  
};
```

```
struct bc_noarg { enum opcode opcode; };  
  
struct bc_arg64 { enum opcode opcode; int64_t arg; };
```

```
union ins {  
  
    enum opcode opcode;  
  
    struct bc_arg64 as_arg64;  
  
    struct bc_noarg as_noarg;  
  
};
```

```
/* Программа: */
```

```
const union ins program[] = {  
  
    { BC_IREAD },  
  
    { BC_IREAD },  
  
    { BC_DUP },  
  
    { .as_arg64 = { BC_PUSH, .arg = 10 } },  
  
    { BC_IMUL },  
  
    { BC_IADD },  
  
    { BC_IADD },  
  
    { BC_IPRINT },
```

```
{ BC_STOP }
```

```
};
```

Это не так привычно, как писать команды в текстовом файле, но зато мы можем писать программы на Стёке уже сейчас, ещё не разработав для него [парсер](#) – часть интерпретатора, которая разбирает исходный текст на части и понимает, какие языковые конструкции там фигурируют.

Закодируйте таким образом следующую программу:

```
IREAD
```

```
PUSH 10
```

```
ISUB
```

```
PUSH 2
```

```
IDIV
```

```
IPRINT
```

```
STOP
```

1task – program

## Конструкция switch

Случается, что в программе необходимо закодировать ветвление, зависящее от конкретного значения целого числа. Мы умеем делать это с помощью `if`:

```
void f( int x ) {
```

```
    if (x == 0) { ... }
```

```
    else if (x == 1) { ... }
```

```
    else if (x == 5) { ... }
```

```
    ...
```

```
}
```

Существует специальная конструкция `switch`, которая устраивает ветвление по конкретным значениям числа. Пример выше можно переписать так:

```
void f( int x ) {  
  
    switch (x) {  
  
        case 0: { ... ; break; }  
  
        case 1: { ... ; break; }  
  
        case 5: { ... ; break; }  
  
        // Если ни одна из предыдущих веток не была выполнена, то мы выполним default  
  
        default: { ... ; break; }  
  
    }  
}
```

Обратите внимание на наличие `break` в каждой ветке!

Важное применение `switch` – для ветвления по элементам перечислений:

```
enum color { RED, YELLOW, GREEN };  
  
void f( enum color x ) {  
  
    switch (x) {  
  
        case RED: { ... ; break; }  
  
        case YELLOW: { ... ; break; }  
  
        case GREEN: { ... ; break; }  
  
  
  
        default: { ... ; break; }  
  
    }  
}
```

Другие полезные применения `switch` вы узнаете в уроке о конечных автоматах.


Хорошим стилем считается обязательно учесть все возможные значения перечисления в ветках `case`, а в ветке `default` добавить сообщение об ошибке и, возможно, аварийное завершение программы. Таким образом, если в будущем мы добавим элементы в перечисление и забудем их обработать в каких-то функциях, программа во время тестирования сообщит об ошибке. Иначе программа тихо продолжит работу, но не так, как мы ожидаем. Ошибки же лучше отлавливать как можно раньше.

В интерпретаторе Стёка можно сделать ветвление по коду инструкции, т.е. `enum opcode`.

## Отличия switch от цепочки if'ов


1. В `switch` нетривиальным образом устроена последовательность выполнения веток. Вначале происходит выбор ветки, в которую `case` мы попадаем. Но после завершения ветки мы не завершаем `switch`, а попадаем в следующую ветку.

Ожидание от `switch`



```
void f() {  
    int test = 1;  
  
    1 switch (test) {  
      case 0: { printf("Zero\n"); }  
      case 1: { printf("One\n"); }  
      case 2: { printf("Two\n"); }  
    }  
    2 printf("Done");  
}
```

Реальность



```
void f() {  
    int test = 1;  
  
    1 switch (test) {  
      case 0: { printf("Zero\n"); }  
      case 1: { printf("One\n"); }  
      case 2: { printf("Two\n"); }  
    }  
    2  
    3 printf("Done");  
}
```

1. Это позволяет выполнить действие для нескольких возможных значений:

```
2. switch (n) {  
3.  
4.     case 2:  
5.     case 3:  
6.     case 5:  
7.     case 7: { printf( "prime and less than 10" ); break; }  
8.     default: { printf( "not prime or greater than 10" ); break; }  
9.
```



```
}
```

Именно поэтому необходимо ставить `break` в конец каждой ветки, если вы после её завершения хотите выйти из `switch`.

10. В `switch` можно только проверять числа и символы на равенство. Нельзя исполнить ветку при условии `x < 5`. Нельзя также проверить переменную на равенство другой переменной или выражению:

```
11. switch (n) {  
12.     case 2+x: //ошибка!  
13.     ...  
    }
```

Поэтому используя `if` можно проверять больше типов условий.

14. Некоторые компиляторы подсказывают, когда вы покрыли не все возможные варианты в `switch`. Например, попытка скомпилировать следующий код:

```
15. enum direction {  
16.     NORTH, EAST, SOUTH, WEST  
17. };  
18.  
19. void f(enum direction dir) {  
20.     switch (dir) {  
21.         case NORTH: { printf("North"); break; }  
22.         case SOUTH: { printf("South"); break; }  
23.     }  
24. }
```

с помощью компилятора [GCC](#) приведёт к ошибке:

```
> gcc good-switch.c
```

```
good-switch.c: In function 'f':
```

```
good-switch.c:12:3: error: enumeration value 'EAST' not handled in switch [-Werror=switch]
```

```
12 |     switch (dir) {
```

```
    |     ^~~~~~
```

```
good-switch.c:12:3: error: enumeration value 'WEST' not handled in switch [-Werror=switch]
```

```
cc1: some warnings being treated as errors
```

## Интерпретатор Стёка

Теперь напишем первую версию интерпретатора. Начнём с инструкций: `IREAD`, `IPRINT`, `IADD`, `PUSH` и `STOP`.

У любого языка есть абстрактный вычислитель, и Стёк не исключение. Состояние его абстрактного вычислителя включает в себя:

- программу;
- [счётчик команд](#) – указатель на следующую инструкцию, которая будет выполнена;
- стек чисел.

```
struct vm_state {  
  
    const union ins *ip;  
  
    struct stack_data_stack;  
  
};
```

Мы инициализируем эту структуру и запускаем цикл интерпретации. На каждой итерации происходит следующее:

- выбор команды по адресу из счётчика команд `ip`;
- увеличение счётчика команд `ip` на размер команды,
- выполнение выбранной команды.

Реализуйте интерпретатор для нескольких инструкций языка с помощью `switch`. В тестах интерпретируется программа, считывающая число и прибавляющая к нему 10, а затем выводящая результат в поток вывода.

2task – program

# Декларативное программирование

Вернемся к примеру с описанием кнопок, на которые можно нажимать. Мы описывали кнопки массивом структур:

```
// контекст для обработчиков событий

struct context { int64_t counter; };


// У кнопки есть имя и обработчик

struct button {

    const char* label;

    void (*handler)( struct button*, struct context* );

};


// Две кнопки

struct button buttons[] = {

    { .label = "Say Meow" , .handler = print_meow_handler },

    { .label = "Status"   , .handler = print_ctx_handler },

};
```

В этом описании совмещены:

- данные в виде надписей на кнопках;
- логика работы, заданная указателями на функции.

Наше представление о задаче близко к этому описанию: мы думаем о кнопках и их реакциях на события, а не о последовательности вызовов функций. Поэтому такое [декларативное](#) описание облегчает понимание и написание программы.

В декларативном описании сразу видны логические ошибки, ведь явно указано, какой обработчик соответствует каждой кнопке. Можно легко и быстро изменять описание: достаточно поменять обработчик чтобы изменить реакцию на нажатие кнопки. А больше нас никакие изменения и не интересуют.

Модификация декларативно написанной программы часто более похожа на её настройку, а не переписывание кода, как будто мы правим конфигурационные файлы.

## Расширяемый интерпретатор

Интерпретатор, который мы написали с помощью `switch`, выглядит неплохо: в нём компактно собрана логика выполнения всех немногочисленных инструкций.

```
void interpret(struct vm_state *state) {  
  
    while (true) {  
  
        switch (state->ip->opcode) {  
  
            case BC_PUSH: {  
  
                ...  
  
                break;  
  
            }  
  
            case BC_IREAD: {  
  
                ...  
  
                break;  
  
            }  
  
            case BC_IADD: {  
  
                ...  
  
                break;  
  
            }  
  
            case BC_IPRINT: {  
  
                ...  
  
                break;  
  
            }  
  
            case BC_STOP: return;  
  
            default:  
  
                err("Not implemented");  
  
                return;  
  
        }  
  
    }  
  
}
```

```
state->ip = state->ip + 1;
```

```
}
```

```
}
```

Но полный набор инструкций больше, поэтому расширенная версия функции `interpret` займёт сотни строчек кода. Как следствие, код будет сложно анализировать, понимать и модифицировать. Почему?

Во-первых, если мы продолжим добавлять инструкции, то гигантский `switch` перестанет быть читаемым. Хорошо читаемый и понятный с первого взгляда код – это не роскошь, а необходимость в условиях постоянного стресса, сопровождающего работу промышленного программиста. Программирование даже в нормальных условиях это достаточно тяжёлый для мозга процесс, и продуктивность программиста очень разнится в зависимости от времени суток, количества часов сна и множества других факторов. А размер и сложность промышленных программ огромны: в одном лишь ядре Linux сейчас около 25 миллионов строчек кода. Хороший программист не напишет столько за всю жизнь.

Во-вторых, функцию `interpret` придётся дописывать для каждой новой добавленной инструкции. Поговорим об этом подробнее.

## Как избежать проблем послезавтра

Написав функцию, мы обычно тестируем её. Если тесты прошли успешно, мы мысленно помечаем эту функцию как *скорее всего надёжную*<sup>1</sup>.

Когда мы изменяем *скорее всего надёжную* функцию, нам нужно протестировать её заново. Легко внести небольшое изменение в код, которое полностью поменяет ход выполнения программы. Если логика работы функции изменилась, то старые тесты к ней уже не подходят, нужны новые.

Чтобы избежать постоянного пере-тестирования уже написанного кода, следует закладывать в программы потенциал для расширения. Мы предполагаем, какие именно новые возможности мы захотим добавить в будущем, и организуем код так, чтобы можно было добавить их **не изменяя уже написанных функций**.

По этим же соображениям глобальные изменяемые переменные – это такая большая проблема в промышленном программировании. Представим ситуацию, когда функции `f` и `g` взаимодействуют с глобальной переменной:

```
char global;
```

```
/* f и g мы уже протестировали и изучили */
```

```
char f() { /* чтения и запись в global */ }
```

```
char g() { /* чтения и запись в global */ }
```

```
/* Добавим функцию h */
```

```
char h() { /* чтения и запись в global */ }
```

```
/* Функция h может влиять на работу функций f и g изменяя global */
```

```
/* значит логика работы f и g могла измениться и нужно изучать и тестировать их  
опять */
```

Не так просто решить, в каких местах программа должна быть расширяемой, а в каких – нет. Не имеет смысла добавлять до бесконечности точки роста, делая максимально общее решение.

[1] Тестирование почти никогда не способно доказать отсутствие ошибок.

## Интерпретатор декларативно

Напомним, что сейчас интерпретатор выглядит вот так:

```
void interpret(struct vm_state *state) {  
  
    for (;;) {  
  
        switch (state->ip->opcode) {  
  
            case BC_PUSH: {  
  
                ...  
  
                break;  
  
            }  
  
            case BC_IREAD: {  
  
                ...  
  
                break;  
  
            }  
  
        }  
  
    }  
}
```

```

    case BC_IADD: {

        ...

        break;

    }

    case BC_IPRINT: {

        ...

        break;

    }

    case BC_STOP: return;

    default:

        err("Not implemented");

        return;

    }

    state->ip = state->ip + 1;

}

}

```

Наша задача -- модифицировать его так, чтобы можно было добавлять инструкции не переписывая функцию `interpret`. Оказывается, это достаточно легко сделать.

Любая инструкция совершает действие над состоянием. Также все типы инструкций имеют свои номера, в соответствии с перечислением `enum opcode`. Заведём массив указателей на функции, реализующие инструкции; указатель на реализацию инструкции с номером `nn` будет лежать в массиве по индексу `nn`.

---

### Sample Input:

90

---

### Sample Output:

100

## Больше инструкций

В Стёке есть несколько бинарных операций (`IADD`, `IDIV`, `ICMP`...). Эти операции **забирают два аргумента из стека, применяют к ним операцию и кладут результат обратно**.

В этом описании жирным выделена общая для всех бинарных операций часть. А если есть общая схема, значит, можно её вычленить в отдельную функцию, избежав дублирования кода:

```
void lift_binop( struct stack* s, int64_t (f)(int64_t, int64_t))
```

Эта функция принимает указатель на функцию `f`, которая и совершает полезные действия с двумя числами, например, возвращает их сумму. Функция `lift_binop` забирает со стека аргументы, передаёт их в `f` и кладёт в стек результат.

Имея `lift_binop`, бинарные операции будут выражаться вот так:

```
int64_t i64_add(int64_t a, int64_t b) { return a + b; }
```

```
void interpret_iadd( struct vm_state* state ) {
```

```
    lift_binop(&state->data_stack, i64_add);
```

```
}
```

Аналогичным образом `NEG` выражается с помощью функции `lift_unop` со следующей сигнатурой:

```
void lift_unop( struct stack* s, int64_t (f)(int64_t));
```

Реализуйте эти две функции, а также остальные необходимые инструкции.

## Тестирование

Первое число, которое считывается со входа, это номер программы от 0 до 4-х. Задача проверяется на этих программах с разными входными данными. То есть чтобы запустить нулевую программу на данных "3 58" введите "0 3 58"

Первые тестовые программы такие:

*Программа 0 (тест 1)*

```
iread
```

```
iread
```

```
dup
```



```
push 10
```

```
imul
```

```
iadd
```

```
iadd
```

```
iprint
```

```
stop
```

*Программа 1 (тест 2)*

```
iread
```

```
push 10
```

```
isub
```

```
push 2
```

```
idiv
```

```
iprint
```

```
stop
```

4task – program

## Декларативное описание инструкций

Напомним, что IP – это счётчик команд, часть состояния вычислителя, указывающая на следующую команду для выполнения.

Добавим в Стёк ещё несколько инструкций, которые будут влиять на значение IP:

- `JZ <arg>` вынимает одно значение из стека. Если оно нулевое, то сдвигаем текущее значение IP на `arg` размеров инструкций. Иначе переходим на следующую команду. Аргумент может быть отрицательным, что позволяет организовывать с помощью этой команды циклы.
- `JMP <arg>` прибавляет к текущему значению IP значение аргумента, умноженное на размер инструкции.

Сейчас интерпретатор:

- выбирает инструкцию из памяти;
- выполняет команду;
- сдвигает IP на следующую инструкцию.

```
void interpret(struct vm_state *state) {
    for (;;) {
        interpreters[state->ip->opcode]( state );
        state->ip = state->ip + 1;
    }
}
```

Мы бы хотели, чтобы прибавка к IP происходила только для тех инструкций, которые сами не изменяют IP, то есть для всех, кроме `JZ` и `JMP`.

Это означает, что нам придётся:

- или делать в интерпретаторе "специальные случаи" для JZ/JMP (что неудобно)
- ```
void interpret(struct vm_state *state) {
```
- ```
    for (;;) {
```
- ```
        interpreters[state->ip->opcode]( state );
```
- ```
        if (state->ip->opcode == BC_JMP || state->ip->opcode == BC_JZ) { ... }
```
- ```
        state->ip = state->ip + 1;
```
- ```
    }
```
- ```
}
```
- или в каждую "обычную" инструкцию вкодировать увеличение IP на размер инструкции
- ```
int64_t i64_add(int64_t a, int64_t b) { return a + b; }
```
- 
- ```
void interpret_iadd( struct vm_state* state ) {
```
- ```
    lift_binop(&state->data_stack, i64_add);
```
- ```
    state->ip = state->ip + 1;
```
- ```
}
```
- 
- 
- ```
void interpret(struct vm_state *state) {
```

- `for (;;) {`
- `interpreters[state->ip->opcode] ( state );`
- `}`

`}`

Призовём на помощь декларативный стиль программирования. Опишем инструкции декларативно. Для этого подумаем про список вопросов, отвечая на которые можно описать команду. Что нам интересно про конкретный тип инструкции? Как она называется, сколько у неё аргументов, изменяет ли она IP нетривиальным образом:

```
enum ins_arg_type { IAT_NOARG, IAT_I64 };
```

```
struct ins_descr {
    const char* mnemonic;          // мнемоника
    enum ins_arg_type argtype;      // тип аргументов: 0, 1 численных аргументов
    bool affects_ip;               // изменяет ли инструкция IP?
};
```

Создадим массив `instructions`, в котором приведём эти описания.

```
const struct ins_descr instructions[] = {
    /*          mnemonic  argtype  affects_ip */
    [BC_PUSH]   = { "push",  IAT_I64,  false },
    [BC_IADD]   = { "iadd",  IAT_NOARG, false },
    [BC_ISUB]   = { "isub",  IAT_NOARG, false },
    [BC_IMUL]   = { "imul",  IAT_NOARG, false },
    [BC_IDIV]   = { "idiv",  IAT_NOARG, false },
    [BC_IMOD]   = { "imod",  IAT_NOARG, false },
    [BC_INEG]   = { "ineg",  IAT_NOARG, false },
    [BC_IPRINT] = { "iprint", IAT_NOARG, false },
    [BC_IREAD]  = { "iread",  IAT_NOARG, false },
    [BC_SWAP]   = { "swap",  IAT_NOARG, false },
}
```

```

[BC_POP]    = { "pop",    IAT_NOARG,    false    },
[BC_DUP]    = { "dup",    IAT_NOARG,    false    },
[BC_ICMP]   = { "icmp",   IAT_NOARG,    false    },
[BC_JMP]    = { "jmp",    IAT_I64     ,    true    },
[BC_JZ]     = { "jz",     IAT_I64     ,    true    },
[BC_STOP]   = { "stop",   IAT_NOARG,    true     }

};

```

Мы придумали достаточно универсальный формат описания инструкций. Большая часть смысла инструкций будет выводиться из него. На функции, реализующие инструкции, остаётся только маленькая, специфичная, и самая важная смысловая часть, которую нужно закодировать отдельно для каждой инструкции.

Интерпретатор можно переписать:

```

void interpret(struct vm_state *state) {
    for (;;) {
        const enum opcode op = state->ip->opcode;

        interpreters[op]( state );

        if (! instructions[op].affects_ip) {
            state->ip = state->ip + 1;
        }
    }
}

```

## Проверки состояния стека

Добавим надёжности в наш интерпретатор. Бывают ситуации, когда в стеке недостаточно элементов, чтобы осуществить операцию, например, когда в стеке одно число, а мы выполняем инструкцию `IADD`. Мы бы хотели отлавливать такие ситуации, останавливать программу и выводить сообщение об ошибке. Для этого добавим два поля в описание инструкций:

```

enum ins_arg_type { IAT_NOARG, IAT_I64 };

```

```

struct ins_descr {

    const char*      mnemonic;      // мнемоника

    enum ins_arg_type argtype;      // тип аргументов: 0, 1 численных аргументов

    bool             affects_ip;    // изменяет ли инструкция IP?

    // Новые поля:

    size_t           stack_min;     // минимальное количество элементов для
инструкции

    int64_t          stack_delta;   // сколько требуется аргументов в стеке

};

```

Мы используем эти поля следующим образом:

- `stack_delta` чтобы обнаружить ситуации, когда стек переполняется. Например, в стеке столько элементов, сколько он только может вместить, и мы выполняем `PUSH`.
- `stack_min` чтобы обнаружить ситуации, когда в стеке недостаточно элементов. Например, в стеке один элемент, мы выполняем `IADD`.

Допишите интерпретатор так, чтобы при недостатке или избытке элементов в стеке выводилось сообщение об ошибке и программа завершалась. Если в стеке слишком мало элементов для совершения операции, выведите "Stack underflow\n", если слишком много -- "Stack overflow\n".

Для тестирования мы настроили интерпретатор так, чтобы стек вмещал не более пяти элементов.

---

### Sample Input:

```
push 10 push 10 push 10 push 10 push 10 push 10
```

---

### Sample Output:

```
Stack overflow
```

5task – program