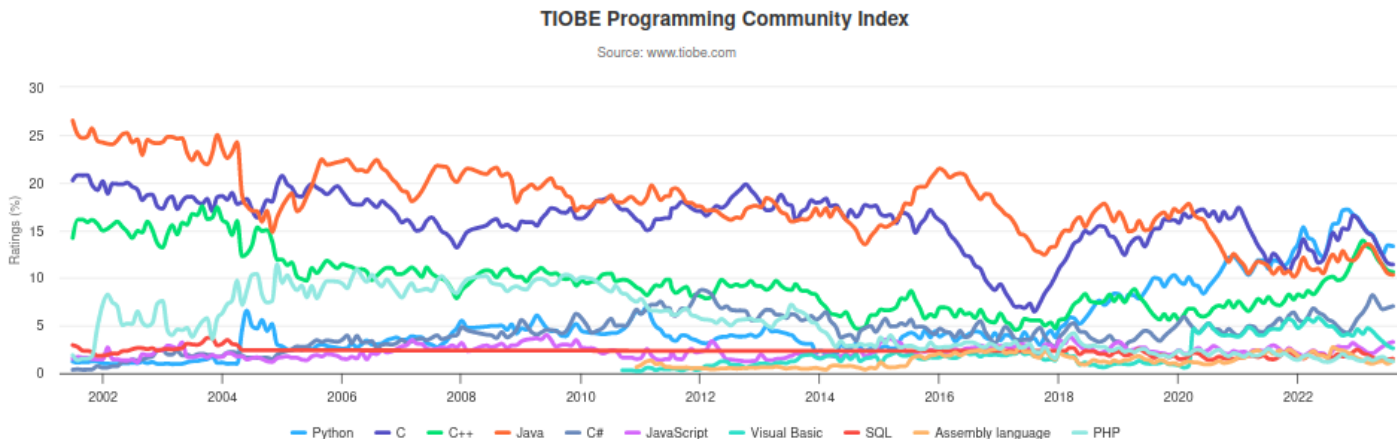


Способы задания однозначности в архитектурах ВС. Связь с типизацией



Начальный взгляд

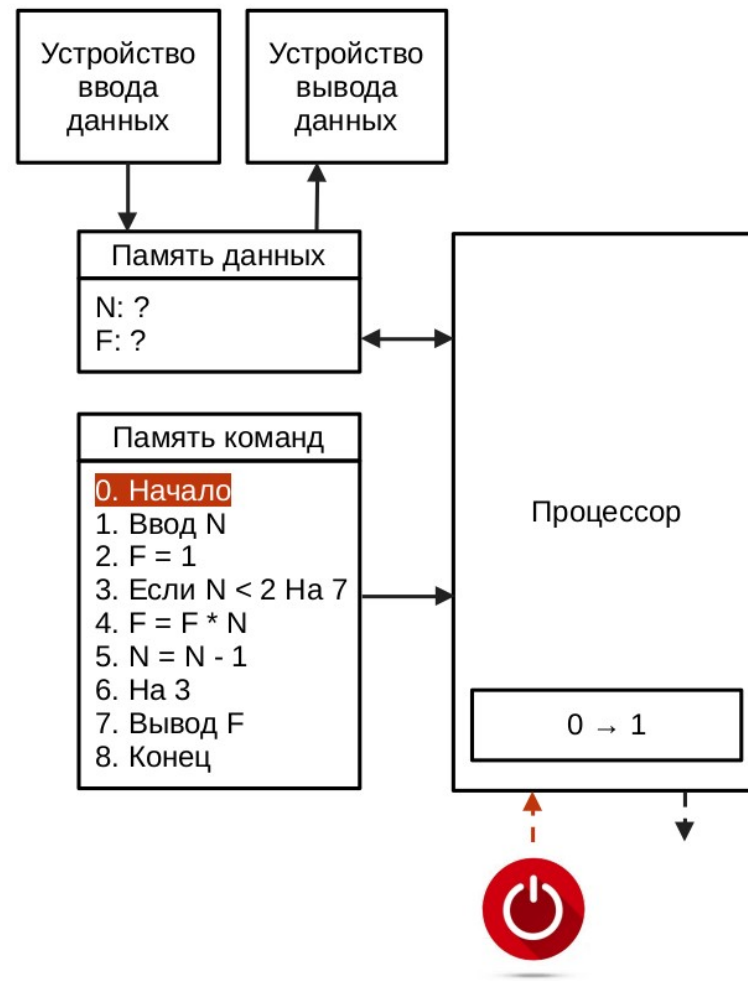
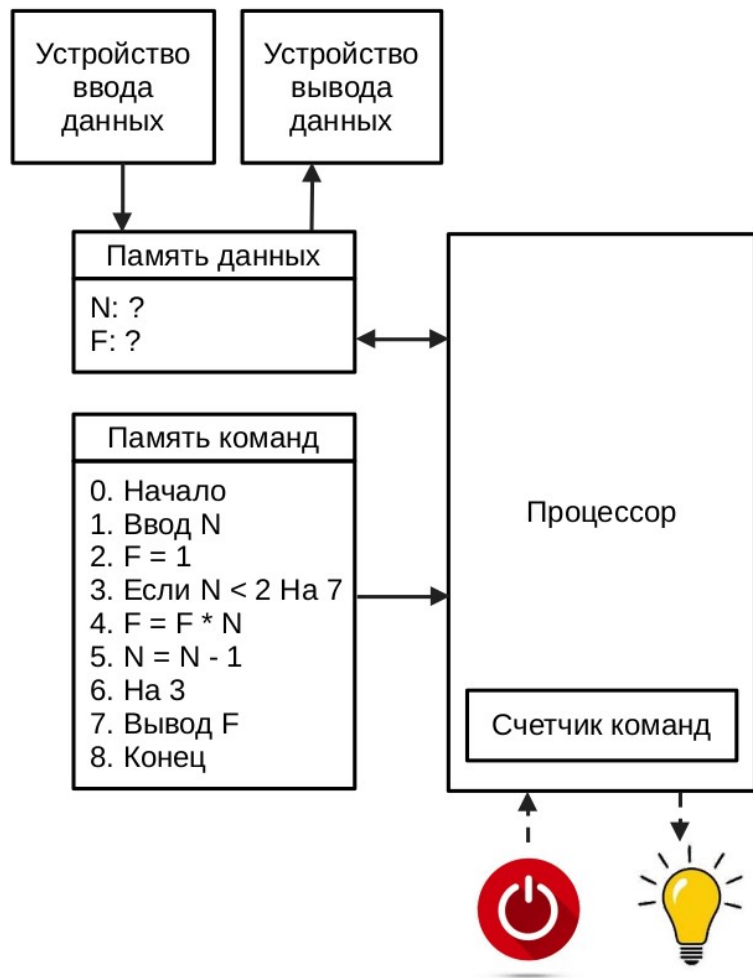
1. Начало
2. Конец
3. Ввод
4. Вывод
5. Операция
6. Условный переход
7. Безусловный переход



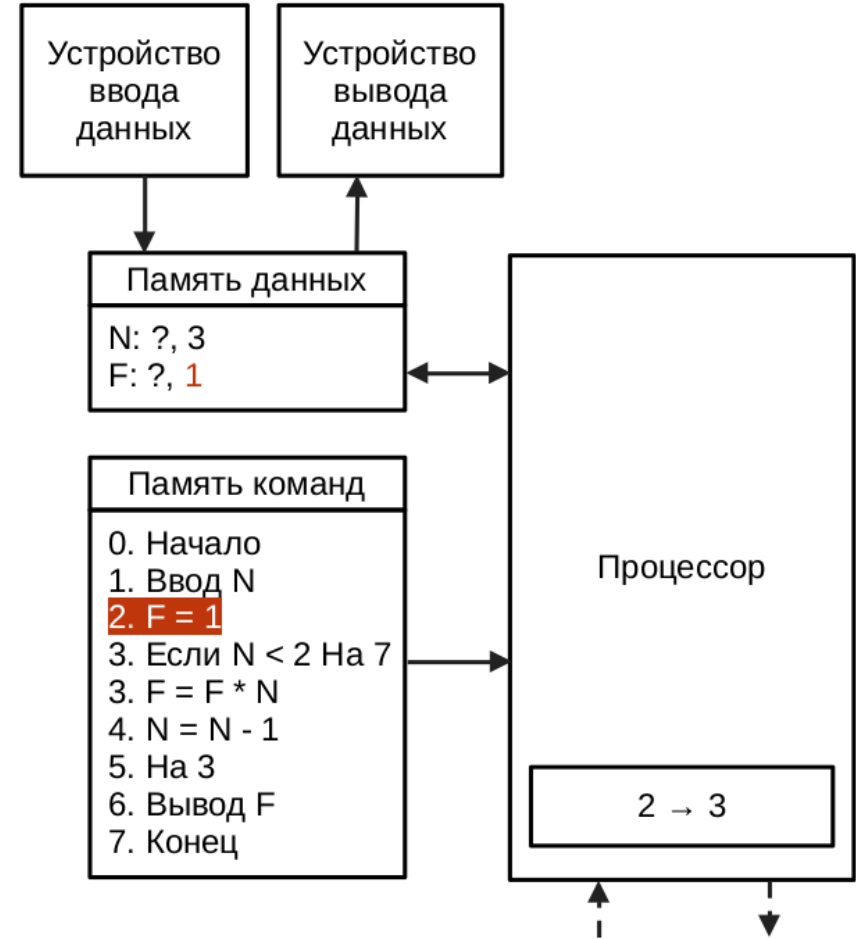
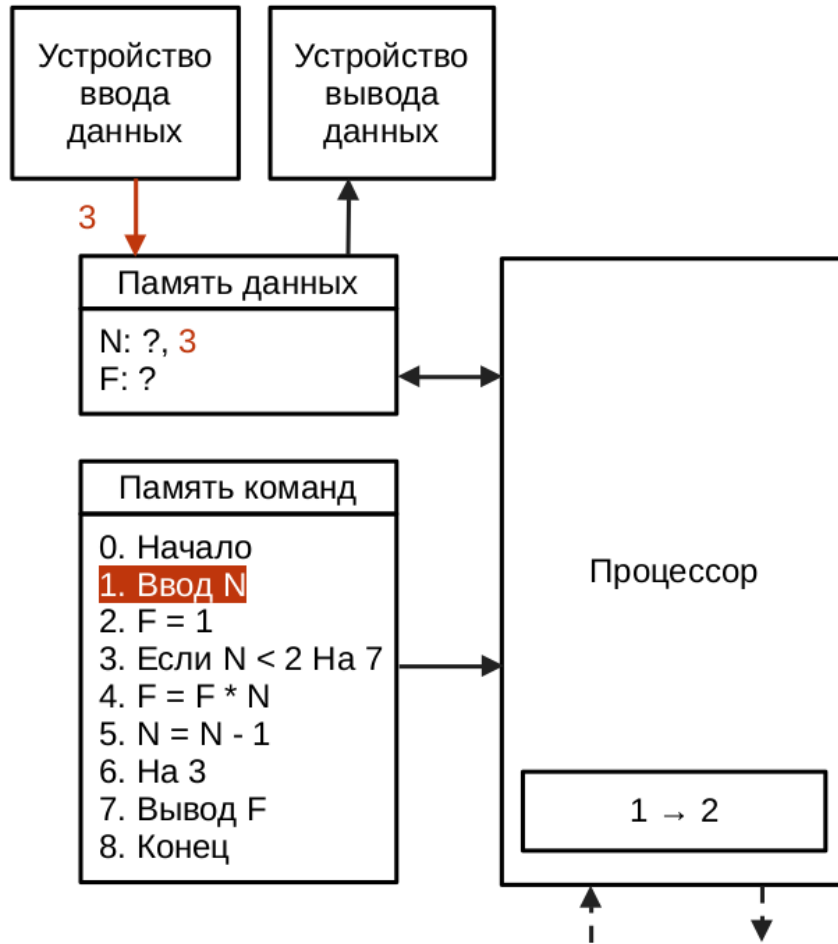
Вычисление факториала. Что внутри?

0. Начало
1. Ввод N
2. $F = 1$
3. Если $N < 2$ На 7
4. $F = F * N$
5. $N = N - 1$
6. На 3
7. Вывод F
8. Конец

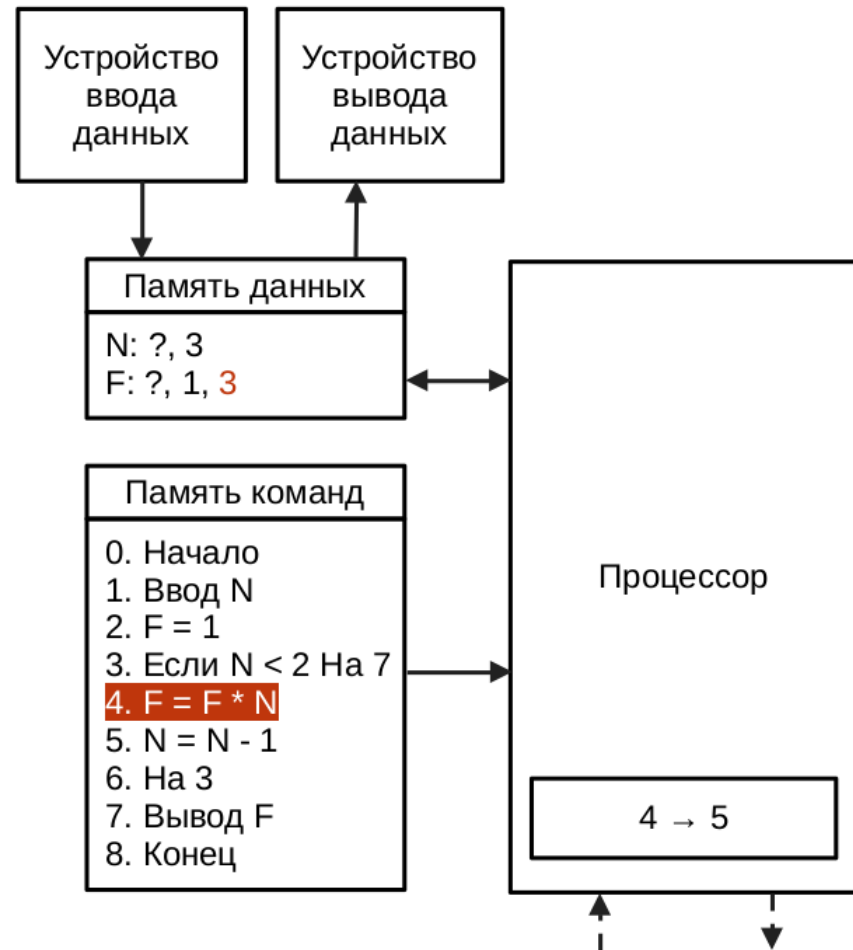
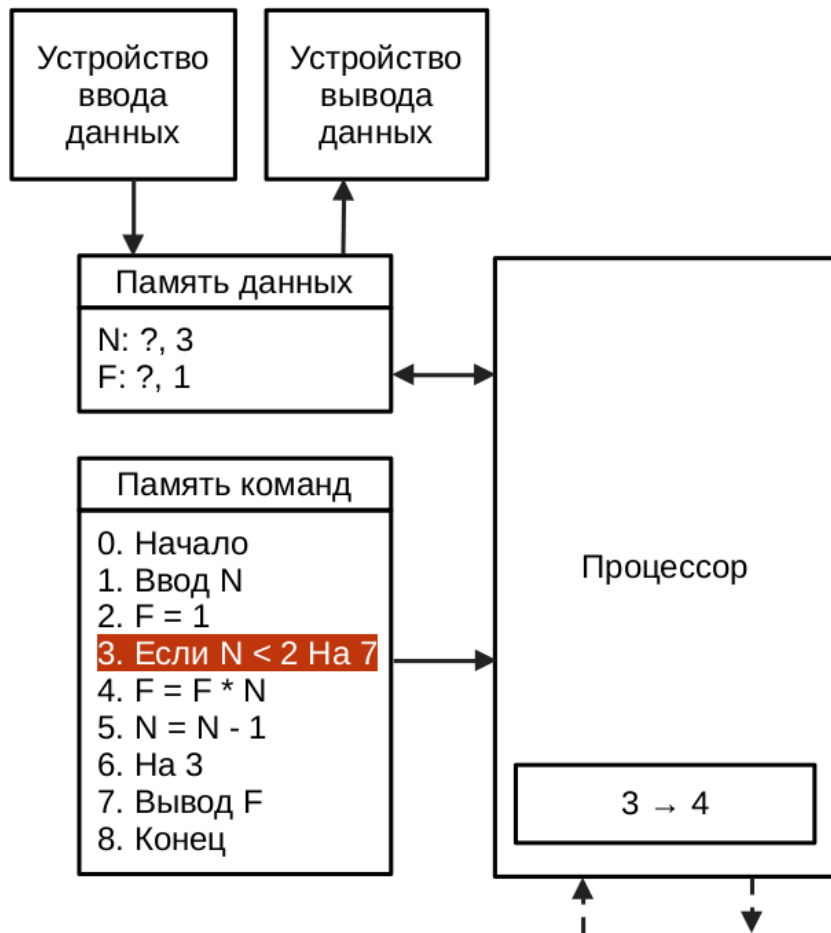
Что внутри?



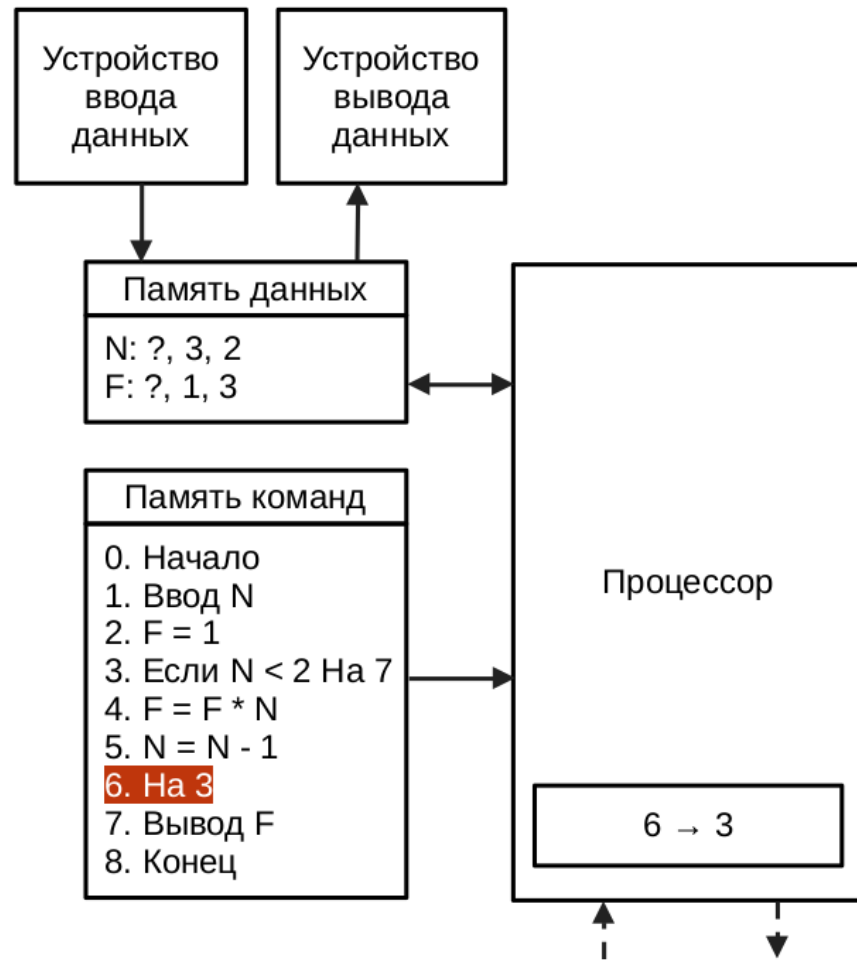
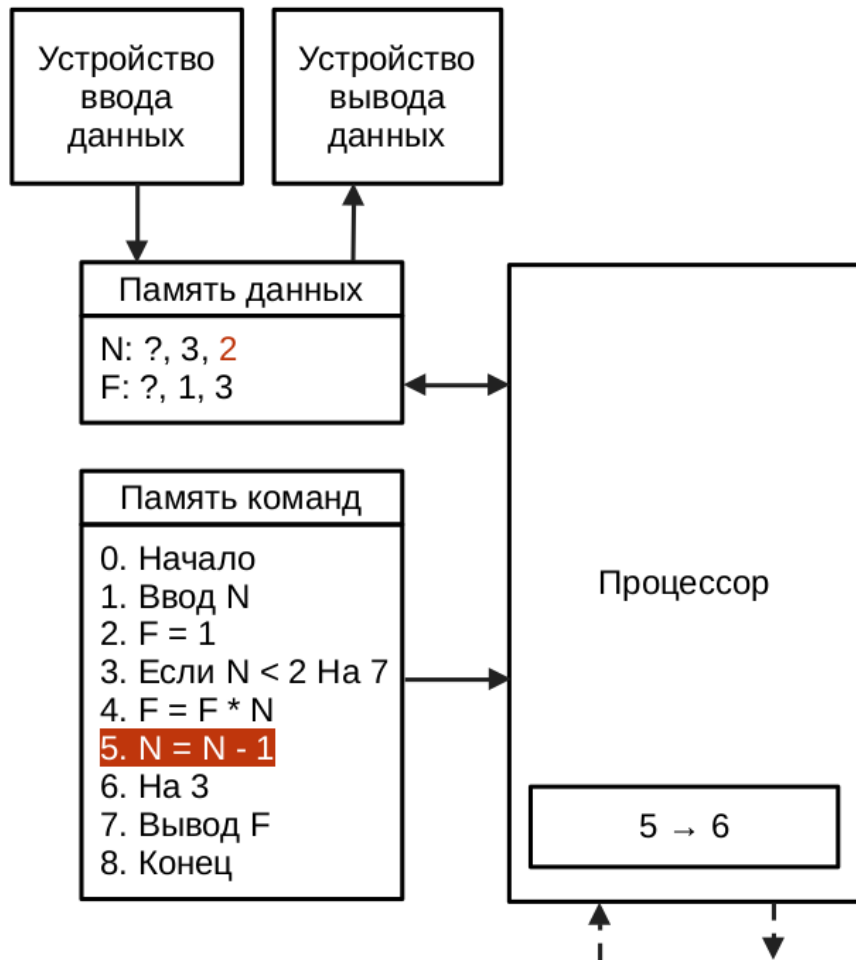
Что внутри?



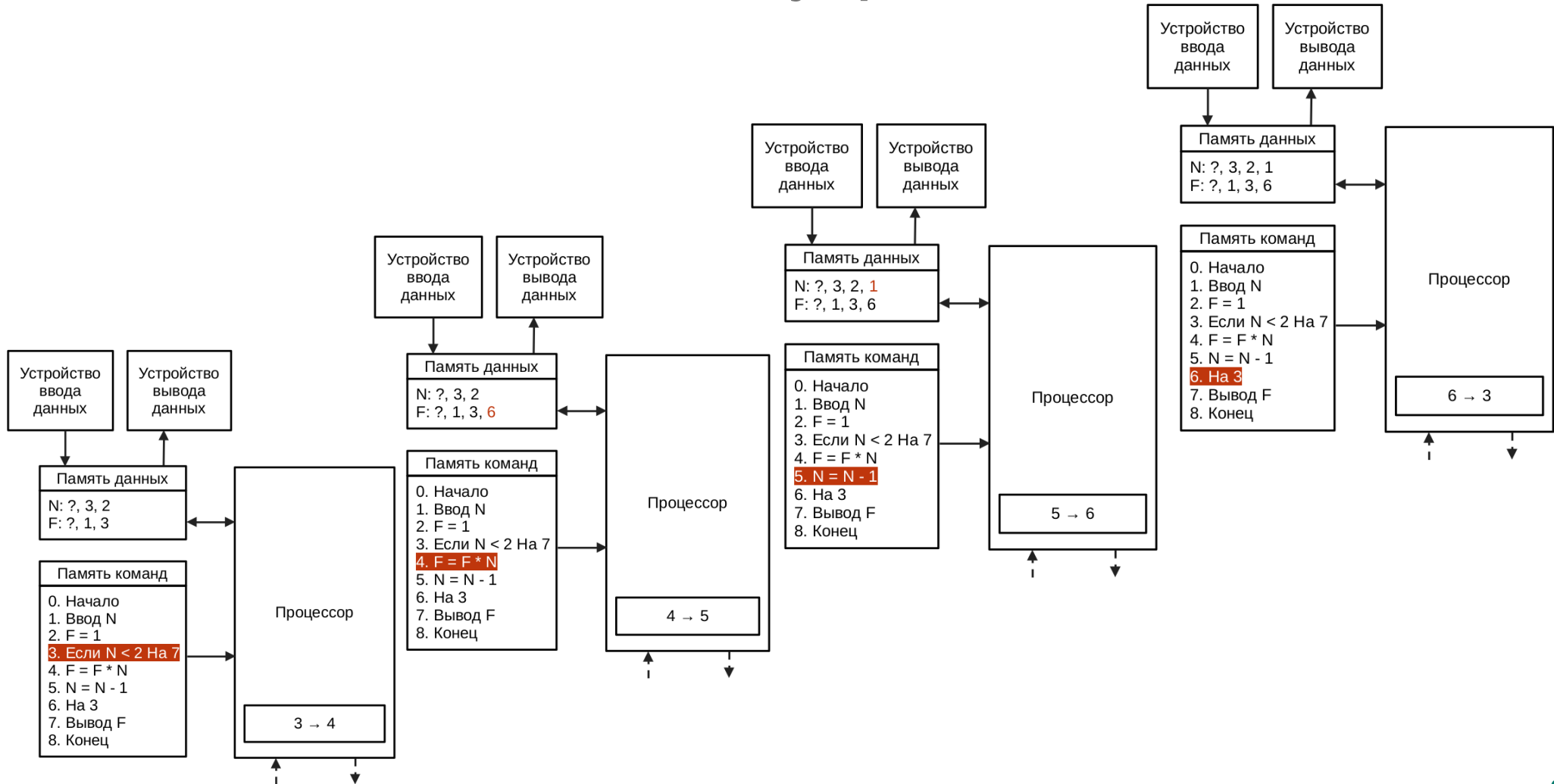
Что внутри?



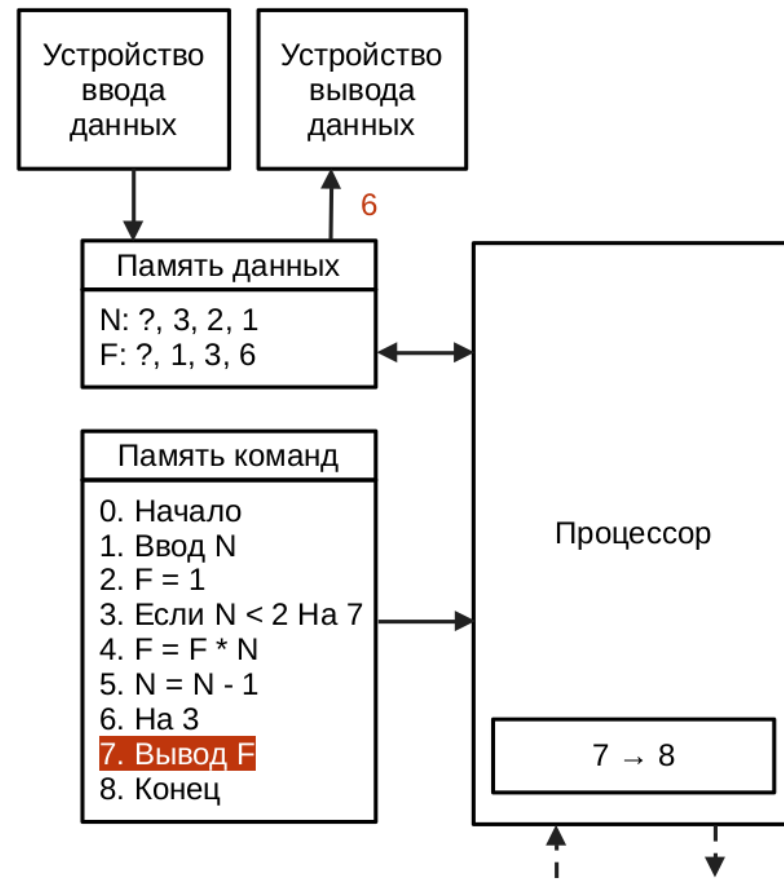
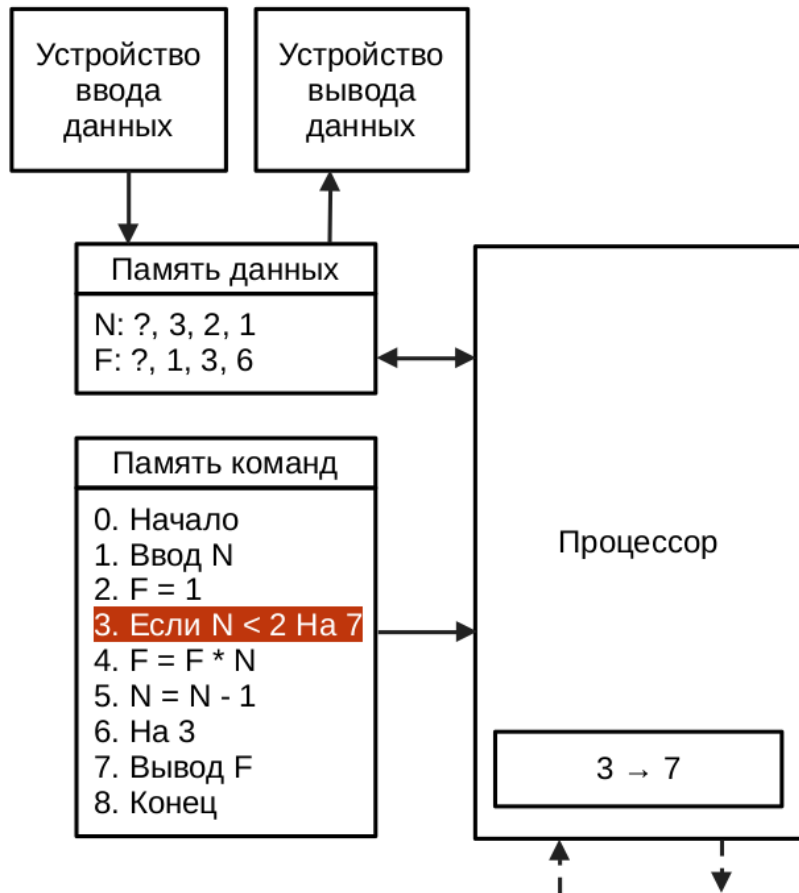
Что внутри?



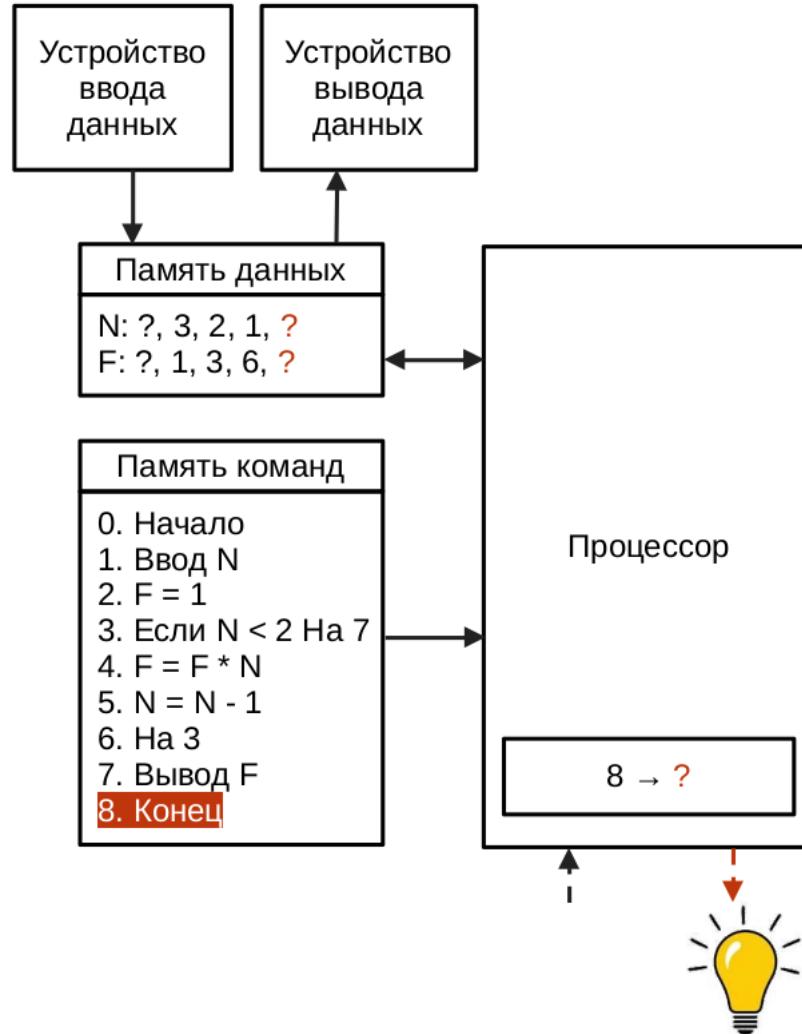
Что внутри?



Что внутри?



Что внутри?



А компьютер выполнит этот алгоритм?

А компьютер выполнит этот алгоритм?

- 1) Можно ли использовать действительные числа, символы, строки?
- 2) Как воспринимается ввод данных?
- 3) Как отображаются данные?
- 4) На какие устройства ввода-вывода можно использовать?
- 5) Какова семантика каждой выполняемой операции?
- 6) Какой тип памяти данных у N и F?

Факторы, влияющие на каждый из архитектурных уровней

1. Методы алгоритмизации (МА)

- императивное программирование
- функциональное программирование
- автоматное программирование
- декларативное программирование

2. Методы композиции (МК)

- абстрактные типы данных + функции
- классы = данные + методы
- модули
- пространства имен

3. Методы задания однозначности (МЗО)

- статическая типизация (однозначность на уровне описания типов данных)
- динамическая типизация (однозначность на уровне вычисляемых тегов)
- операционная однозначность (на уровне кодов операций компьютера)

4. Методы управления вычислениями (МУВ)

- последовательное программирование
- параллельное программирование (разнообразие вариантов)

5. Уровни абстракции (УА)

- Непосредственное отображение
- Абстракция типов
- Метапрограммирование

Способы задания однозначности

Однозначность определяет четкие правила выполнения операций реальными и виртуальными вычислительными системами, позволяя избегать или обходить ошибки программирования.

Различные методы задания однозначности операций позволяют **контролировать корректность программы** с разной степенью и на различных стадиях обработки

Выделяются:

- 1) Операционная однозначность (бестиповые системы)
- 2) Динамическая однозначность (системы с динамической типизацией)
- 3) Статическая однозначность (системы со статической типизацией)

Операционная однозначность

Однозначность операций формируется за счет четкого определения что и с какими типами данных делает каждая операция. Сами данные при этом не несут никакой дополнительной семантической идентификации и представляются в виде набора строк бит (байт), размещенных в памяти. Доступ к обезличенным данным осуществляется по адресам, задаваемым в операциях. Для таких архитектур характерны бестиповые языки.

Примеры подобных архитектур:

- 1) Современные архитектуры уровня системы команд и их языки ассемблера
- 2) Объектно-ориентированный язык программирования Eolang
- 3) Языки системного программирования

Программа для компьютера

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      printf("n? ");
8      scanf("%d", &n);
9      loop:
10     | if(n < 2) goto end;
11       f *= n;
12       n--;
13       goto loop;
14     end:
15     | printf("n! = %d\n", f);
16       return 0;
17     }
```


Бестиповое программирование в RARS (Assembler RISC-V)

```
1  # Нахождение факториала
2  .data
3      arg:      .asciz "n? "
4      result:   .asciz "n! = "
5      f:        .word 1
6  .text
7  start:
8      la a0, arg          # Подсказка для первого числа
9      li a7, 4            # Вызов №4. Вывод строки
10     ecall
11     li a7, 5            # Вызов №5. Ввод целого n
12     ecall
13     mv t1, a0           # t1 = n
14     la t2, f            # t2 = &f
15     lw t3, 0(t2)        # t3 = f
16     li t4, 2
```

Run I/O

```
n? 10
n! = 3628800
```

```
-- program is finished running (0) --
```

```
17  loop:
18      blt t1, t4, end     # if(n < 2) goto end
19      mul t3, t3, t1      # f *= n
20      li t5, -1
21      add t1, t1, t5      # n--
22      b loop
23  end:
24      la a0, result       # Подсказка для результата
25      li a7, 4            # Системный вызов №4
26      ecall
27      li a7, 1           # Вывести десятичное число
28      mv a0, t3
29      ecall
30      li a0, '\n'        # Перевод строки
31      li a7, 11          # Системный вызов №11
32      ecall
33      li a7, 10          # №10 – останов программы
34      ecall
```

Бестиповое программирование на GNU Assembler (Intel)

```
1 # asm-fact.s
2 .intel_syntax noprefix
3 # Константные данные
4 .section .rodata
5 question:
6 .string "n? "
7 .equ questionLength, .-question-1
8 formatIn:
9 .string "%d"
10 .equ formatInLength, .-formatIn-1
11 formatOut:
12 .string "n! = %d\n"
13 .equ formatOutLength, .-formatOut-1
14
15 # Статические переменные
16 .data
17 n: .long 0
18
19 # Текст программы
20 .text
21 .globl main
22 main:
23     push    rbp                # пролог
24     mov     rbp, rsp
25
26     # Ввод начального значения n
27     lea     rdi, question[rip] # адрес формата подсказки
28     mov     eax, 0             # не действительные числа
29
30
31     call    printf@plt         # печать подсказки
32
33     lea     rdi, formatIn[rip] # адрес формата числа
34     lea     rsi, n[rip]
35     mov     eax, 0             # не действительные числа
36     call    scanf@plt         # ввод целого
37
38     # Вычисление факториала
39     mov     eax, 1             # начальная установка f
40     mov     ebx, n[rip]        # перенос n в регистр
41
42     loop:
43         cmp     ebx, 2         # проверка на завершение
44         jl      end           # выход по меньше
45         mul     ebx            # f *= n
46         dec     ebx           # --n;
47         jmp     loop
48
49     end:
50
51     # Вывод результата вычислений
52     lea     rdi, formatOut[rip] # адрес формата результата
53     mov     rsi, rax           # значение результата
54     mov     eax, 0             # не действительные числа
55     call    printf@plt         # печать результата
56
57     mov     eax, 0             # return 0
58     pop     rbp               # эпилог
59     ret
```



Нисан Ноам, Шокен Шимон.

Архитектура компьютерных систем. Как собрать современный компьютер по всем правилам — Москва : Эксмо, 2023. — 496 с.

Сайт книги:

<https://www.nand2tetris.org/>

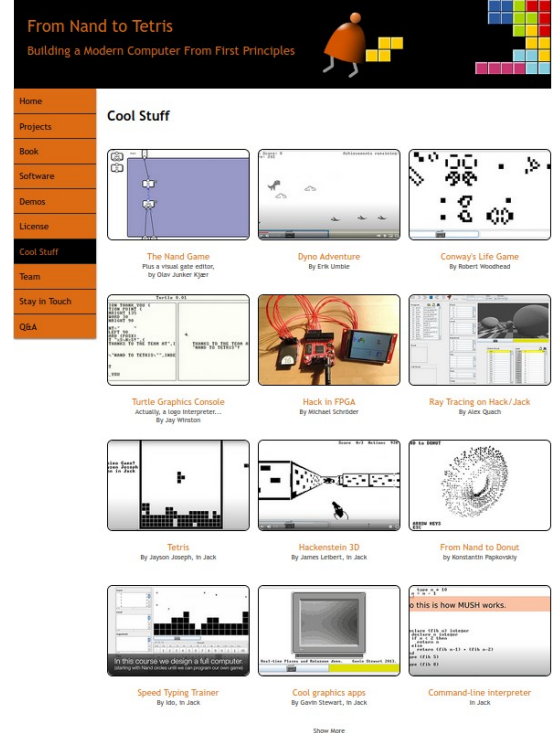
Создание современного компьютера с нуля: от Nand до Tetris (проектный курс)

<https://www.coursera.org/learn/build-a-computer>

Build a Modern Computer from First Principles: Nand to Tetris Part II (project-centered course)

<https://www.coursera.org/learn/nand2tetris2>

- Хотя программисты редко пишут программы непосредственно на машинном языке, изучение низкоуровневого программирования — необходимое условие для полного и глубокого понимания того, как работают компьютеры. Кроме того, глубокое понимание низкоуровневого программирования помогает программисту писать более качественные и эффективные программы высокого уровня. Наконец, довольно увлекательно наблюдать на практике за тем, как самые сложные программные системы оказываются, по сути, потомками простых инструкций, каждая из которых задумана выполнять побитовую операцию на аппаратном уровне.



Динамическая однозначность

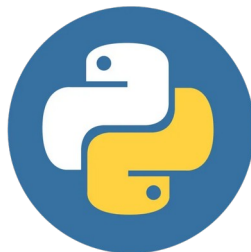
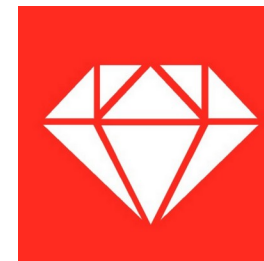
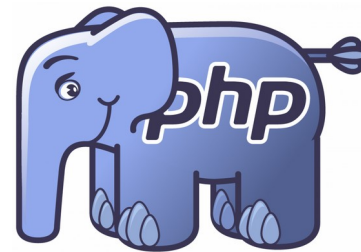
Динамическая однозначность операций формируется за счет того, что с каждым значением, формируемым в программе сопоставляется его тип. Любая операция над данным может проверить этот тип и выбрать в соответствии с этим нужные вычисления. То есть, одна и та же операция может обрабатывать различные типы данных. При этом идентификация типа осуществляется во время выполнения программы. Одни и те же переменные могут хранить данные различного типа. В любой момент программа может проверить тип переменной. Данный подход широко используется в языках программирования, ориентированных на интерпретацию.

Динамическая типизация — приём, используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

Свойства языков с динамической типизацией





Примеры языков с динамической типизацией:

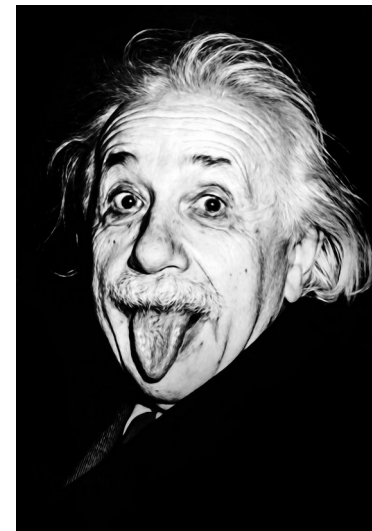
- Smalltalk
- Python
- Objective-C
- Ruby
- PHP
- Perl
- JavaScript
- Лисп



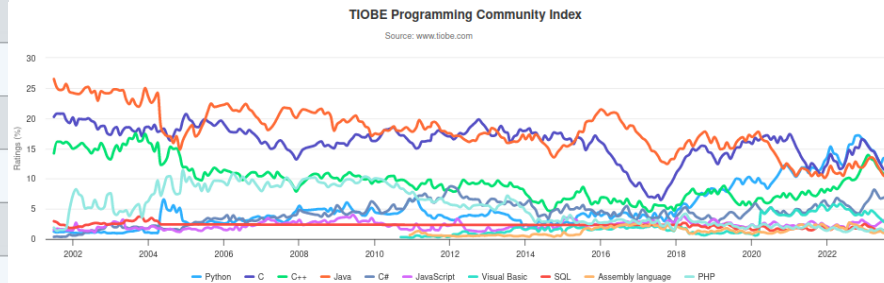
Динамическая типизация упрощает написание программ для работы с меняющимся окружением, при работе с данными переменных типов; при этом отсутствие информации о типе на этапе компиляции повышает вероятность ошибок в исполняемых модулях.

Позиции языков с динамической типизацией

Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		 Python	13.33%	-2.30%
2	2		 C	11.41%	-3.35%
3	4	▲	 C++	10.63%	+0.49%
4	3	▼	 Java	10.33%	-2.14%
5	5		 C#	7.04%	+1.64%
6	8	▲	 JavaScript	3.29%	+0.89%
7	6	▼	 Visual Basic	2.63%	-2.26%
8	9	▲	 SQL	1.53%	-0.14%
9	7	▼	 Assembly language	1.34%	-1.41%
10	10		 PHP	1.27%	-0.09%
11	21	▲	 Scratch	1.22%	+0.63%
12	15	▲	 Go	1.16%	+0.20%
13	17	▲	 MATLAB	1.05%	+0.17%
14	18	▲	 Fortran	1.03%	+0.24%
15	31	▲	 COBOL	0.96%	+0.59%
16	16		 R	0.92%	+0.01%
17	19	▲	 Ruby	0.91%	+0.18%
18	11	▼	 Swift	0.90%	-0.35%
19	22	▲	 Rust	0.89%	+0.32%
20	28	▲	 Julia	0.85%	+0.41%



<https://www.tiobe.com/tiobe-index/>



Организация значения при динамической типизации

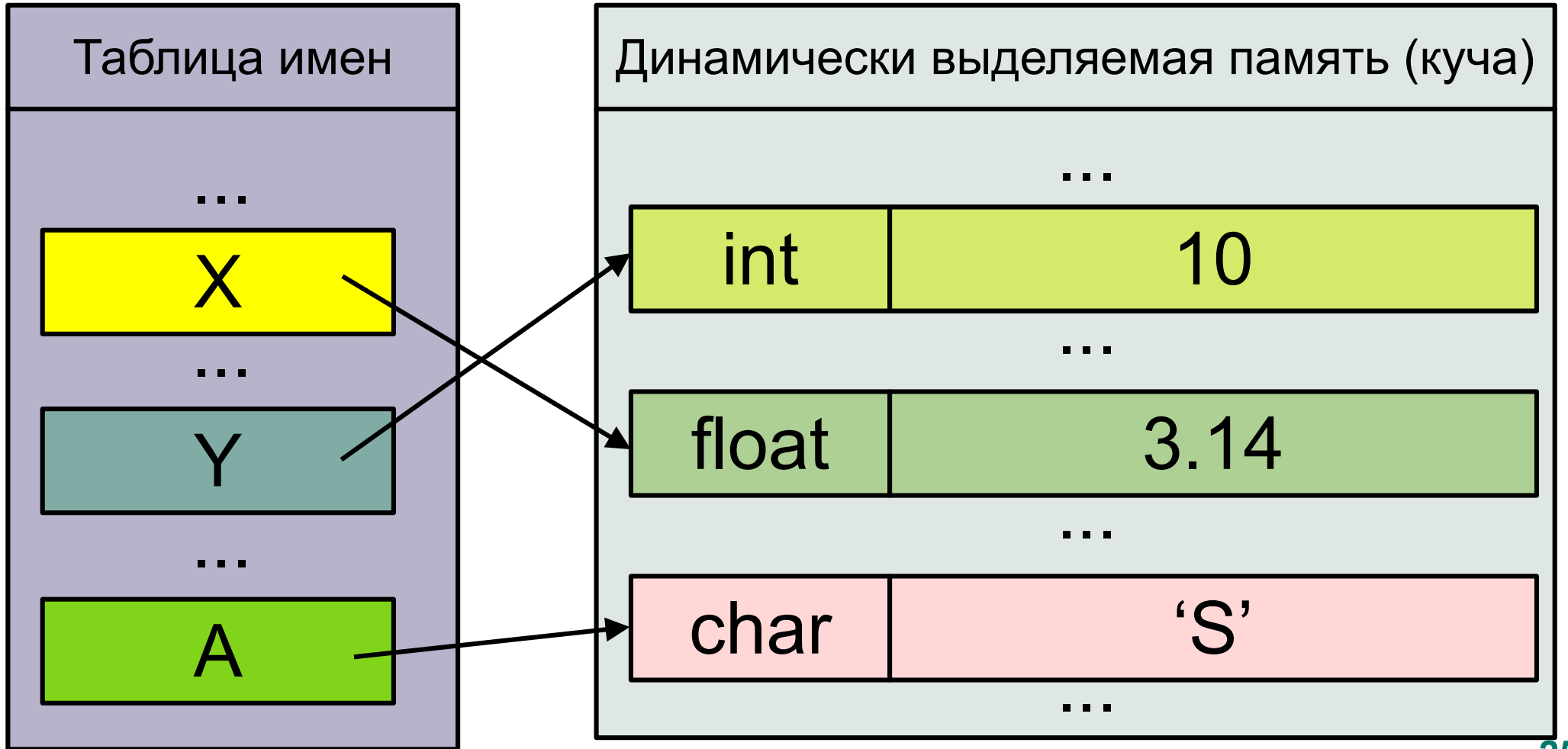
Тип	Величина
-----	----------

int	10
-----	----

float	3.14
-------	------

char	'S'
------	-----

Обращение к величинам через указатели



Python. Использование REPL (Read-Execute-Print Loop) для демонстрации изменения типа переменной

```
>>> value = 10
>>> value
10
>>> type(value)
<class 'int'>
>>> value = 3.14
>>> value
3.14
>>> type(value)
<class 'float'>
>>> value = "Hello!"
>>> value
'Hello!'
>>> type(value)
<class 'str'>
```

Python. Изменение и проверка типа в программе

```
1 import random
2
3 ▼ for i in range(10):
4     key = random.randint(1,2)
5     ▼ if key == 1:
6         value = random.uniform(1.0, 10.0)
7     ▼ else:
8         value = random.randint(100, 200)
9
10 print('key = {0}; value = {1}; type = {2}'.format(key, value, type(value)))
```

```
key = 2; value = 155; type = <class 'int'>
key = 2; value = 130; type = <class 'int'>
key = 1; value = 6.131331242406195; type = <class 'float'>
key = 1; value = 8.280520967840578; type = <class 'float'>
key = 1; value = 5.030964057739875; type = <class 'float'>
key = 2; value = 134; type = <class 'int'>
key = 1; value = 6.393939330816693; type = <class 'float'>
key = 2; value = 101; type = <class 'int'>
key = 1; value = 7.203902995902304; type = <class 'float'>
key = 2; value = 155; type = <class 'int'>
```

Python. Использование динамической однозначности

```
1 n = int(input("n? "))
2 f = 1
3 while n > 1:
4     f *= n
5     n -= 1
6 print("n! = {0}".format(f))
```

```
n? 5
n! = 120
```

```
1 n = float(input("n? "))
2 f = 1
3 while n > 1:
4     f *= n
5     n -= 1
6 print("n! = {0}".format(f))
```

```
n? 5
n! = 120.0
```

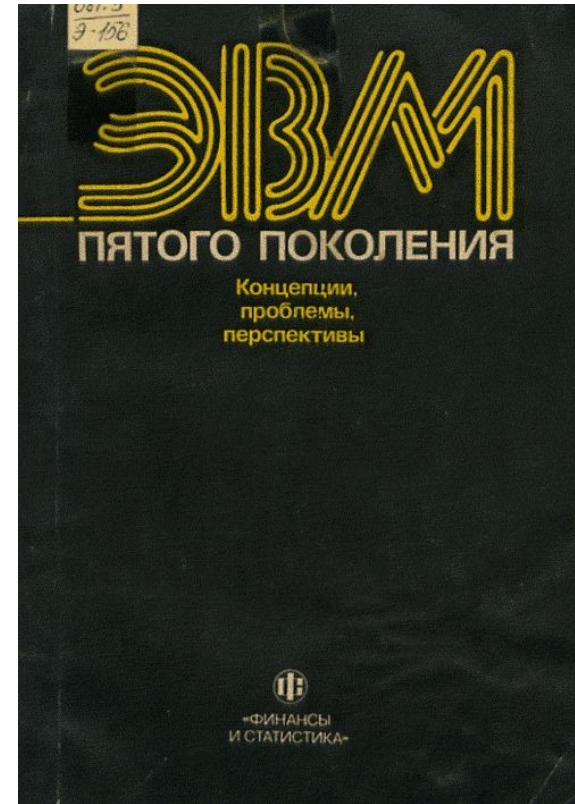
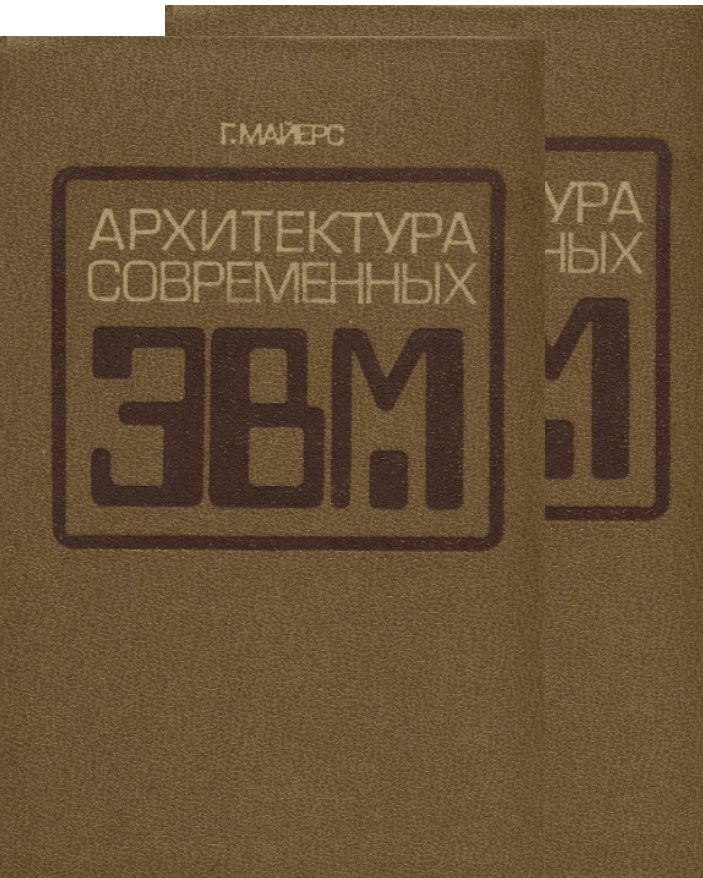
```
n? 4.8
n! = 91.929599999999998
```

Аппаратные решения с динамической однозначностью

Майерс, Г.

Архитектура современных ЭВМ: в 2 кн.

– М.: Мир, 1985.

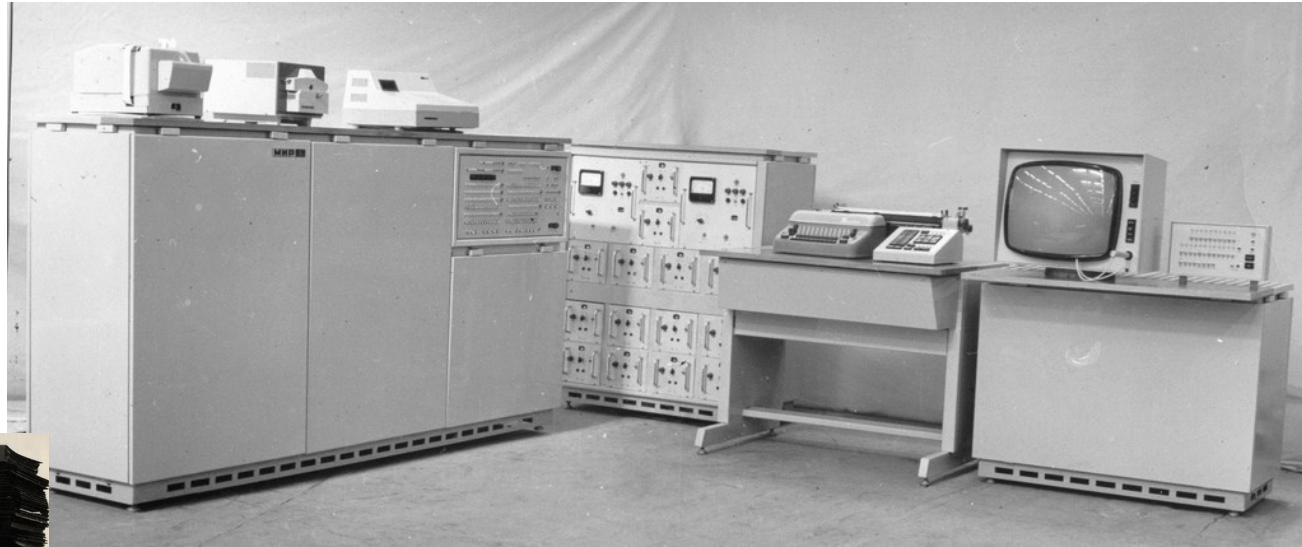


ЭВМ пятого поколения: Концепции, проблемы, перспективы / Под ред. .Мото-ока
– М.: Финансы и статистика, 1984, 110 с.

ЭВМ семейства МИР ***(Машина для Инженерных Расчетов)***

Мир-1

Язык
программирования:
Алмир-65



Мир-2

Язык
программирования:
Аналитик

Архитектура и статическая типизация



Статическая однозначность

Статическая однозначность операций формируется за счет того, что с каждым значением в программе сопоставляется его тип. Этот тип задается при описании переменных и может быть проверен во время компиляции. Для всех временных и промежуточных значений тип может быть также выведен во время компиляции. Поэтому его не имеет смысла проверять во время выполнения. Одна и та же операция может быть задана с разными типами, но все вопросы по ее конкретному выполнению решаются во время компиляции (статический полиморфизм). С каждой переменной сопоставляется только один тип. Допускает эффективную трансформацию в бестиповые архитектуры уровня системы команд. Используется в языках компилируемого типа.

Примеры:

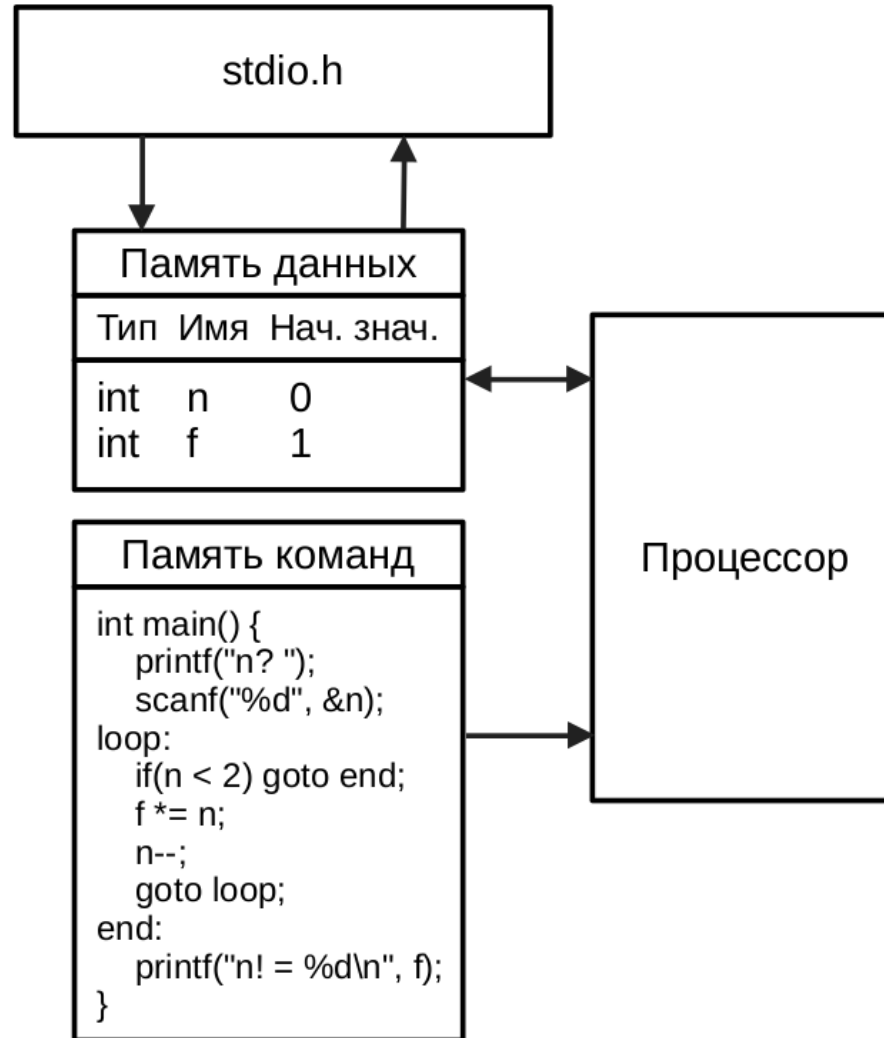
1) Императивные языки программирования: C, C++, Pascal, Oberon family, Java, C#, Rust, Go...

2) Языки функционального программирования: ML, Haskell...

Программа для компьютера

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      printf("n? ");
8      scanf("%d", &n);
9      loop:
10     | if(n < 2) goto end;
11       f *= n;
12       n--;
13       goto loop;
14     end:
15     | printf("n! = %d\n", f);
16       return 0;
17     }
```

Отображение программы на структуру



Lilith

Старт проекта: **1977**
Премия Тьюринга: **1984**
Основа: **Modula-2**

https://en.wikipedia.org/wiki/Lilith_%28computer%29

Руслан Богатырев
Язык как основа архитектуры. Проект Lilith
<https://www.computer-museum.ru/frgnhist/lilith.htm>



Кронос (1984-1988)

Кронос

Ориентация на поддержку:
Си, Модула-2, Паскаль, Оккам и т.п.

<http://kronos.ru/>



Процессор Кронос П2.2



Процессор Кронос П2.5



Процессор Кронос П2.6

Формирование однозначности операций

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      // Во время выполнения:
8      printf("n? ");          // calc(char*)
9      scanf("%d", &n);        // calc(char*); if(%d)-> use n as int
10 loop:
11     // Во время компиляции:
12     if(n < 2) goto end;      // <(int, int) -> bool
13     f *= n;                  // *(int, int) -> int; =(int) -> int
14     n--;                     // --(int) -> int
15     goto loop;
16 end:
17     // Во время выполнения:
18     printf("n! = %d\n", f);  // calc(char*); if(%d)-> use n as int
19     return 0;
20 }
```

```
[ fact01]$ c++ fact.cpp
[ fact01]$ ./a.out
n? 5
n! = 120
```

Формирование однозначности операций

```
1  #include <stdio.h>
2
3  static int n;
4  static int f = 1;
5
6  int main() {
7      // Во время выполнения:
8      printf("n? ");          // calc(char*)
9      scanf("%c", &n);        // calc(char*); if(%c)-> use n as char
10 loop:
11     // Во время компиляции:
12     if(n < 2) goto end;      // <(int, int) -> bool
13     f *= n;                  // *(int, int) -> int; =(int) -> int
14     n--;                     // --(int) -> int
15     goto loop;
16 end:
17     // Во время выполнения:
18     printf("n! = %s\n", f);  // calc(char*); if(%s)-> use n as char*
19     return 0;
20 }
```

```
[fact01]$ c++ bad_fact.cpp
[fact01]$ ./a.out
n? 5
n! = (null)
```

Избыточность статической типизации

```
1 #include <stdio>
2
3 int n;
4 int f = 1;
5
6 ▼ int main() {
7     printf("n? ");
8     scanf("%d", &n);
9     loop:
10     if(n < 2) goto end;
11     f *= n;
12     n--;
13     goto loop;
14 end:
15     printf("n! = %d\n", f);
16 }
```

man 3 printf

man 3 scanf

int printf(char *, ...) →

int scanf(char *, ...) →

< (int, int) → bool

*(int, int) → int; = (int) → int

--(int) → int

**Трансформация к операционной
однозначности:**

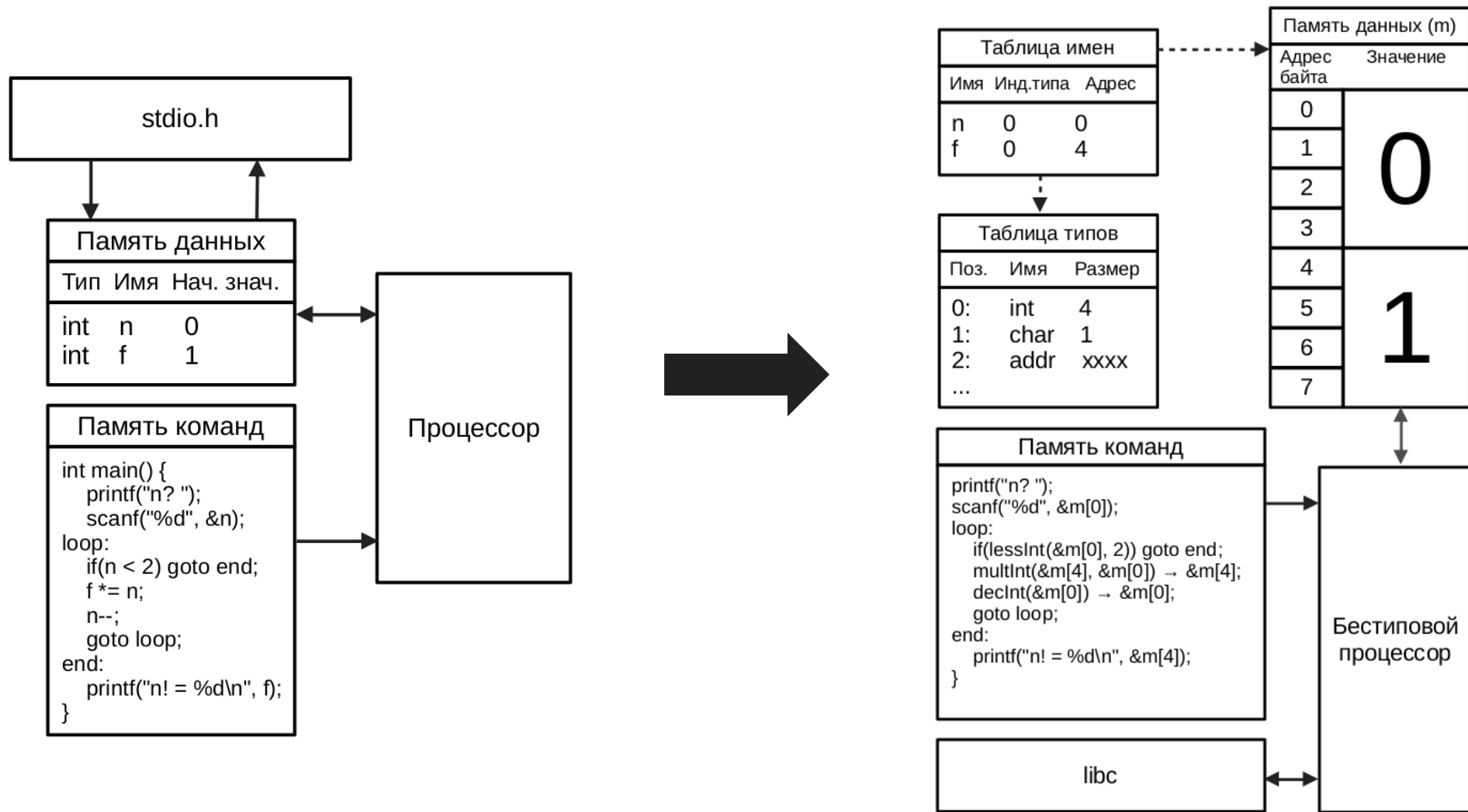
lessInt(void*, void*) → void*

multInt(void*, void*) → void*

movInt(void*) → void*

decInt(void*) → void*

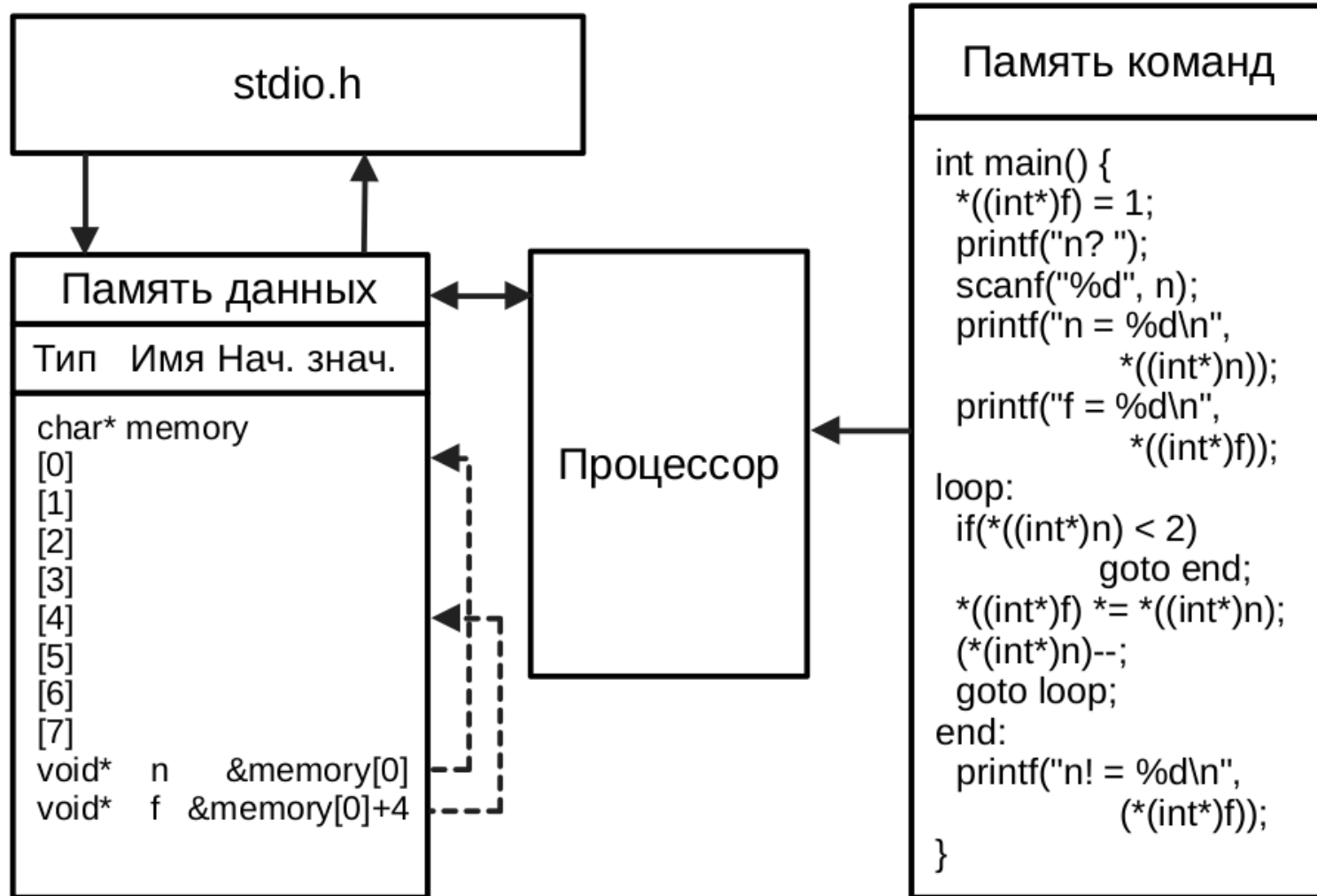
Трансформация статической типизации



Бестиповое программирование на C

```
1 #include <stdio.h>
2
3 static char memory[2*sizeof(int)]; // Память для n и f
4 static void* n = memory; // Адрес на область для n
5 static void* f = memory + sizeof(int); // Адрес на область для f
6
7 int main() {
8     *((int*)f) = 1;
9     printf("n? ");
10    scanf("%d", &n);
11    printf("n = %d\n", *((int*)n));
12    printf("f = %d\n", *((int*)f));
13 loop:
14     if(*((int*)n) < 2) goto end;
15     *((int*)f) *= *((int*)n);
16     (*(int*)n)--;
17     goto loop;
18 end:
19     printf("n! = %d\n", (*(int*)f));
20     return 0;
21 }
```

Отображение бестиповой программы на структуру



Статическая типизация и локальные данные

```
1 #include <stdio.h>
2
3 int factorial(int n) {
4     if(n < 2) {
5         return 1;
6     }
7     int f = 1;
8     for(int i = 2; i <= n; i++) {
9         f *= i;
10    }
11    return f;
12 }
13
14 int main() {
15     int n;
16     printf("n? ");
17     scanf("%d", &n);
18     printf("n! = %d\n", factorial(n));
19 }
```

Глобальная память		
Тип	Имя	Нач. знач.

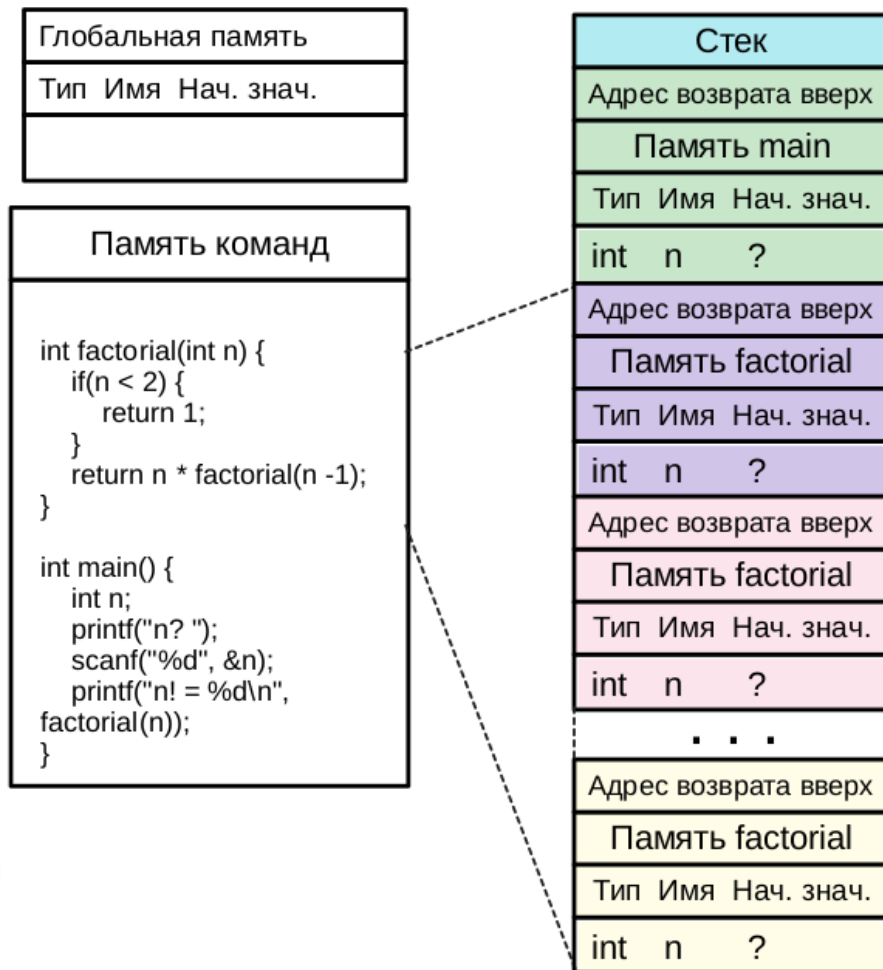
Память команд		
int factorial(int n) { if(n < 2) { return 1; } int f = 1; for(int i = 2; i <= n; i++) { f *= i; } return f; }		
int main() { int n; printf("n? "); scanf("%d", &n); printf("n! = %d\n", factorial(n)); }		

Память factorial		
Тип	Имя	Нач. знач.
int	n	?
int	f	1
int	i	?

Память main		
Тип	Имя	Нач. знач.
int	n	?

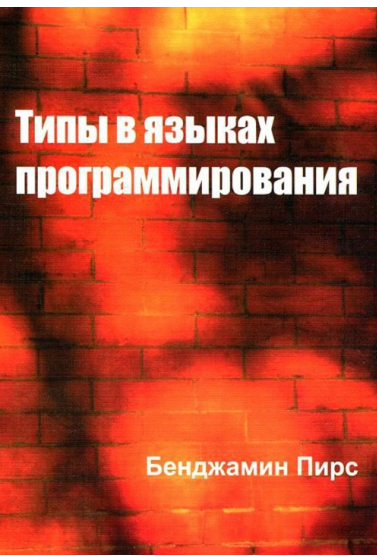
Статическая типизация и рекурсия

```
1 #include <stdio.h>
2
3 int factorial(int n) {
4     if(n < 2) {
5         return 1;
6     }
7     return n * factorial(n - 1);
8 }
9
10 int main() {
11     int n;
12     printf("n? ");
13     scanf("%d", &n);
14     printf("n! = %d\n", factorial(n));
15 }
```



Список источников информации по данной теме

1. [Википедия] Динамическая типизация
https://ru.wikipedia.org/wiki/Динамическая_типизация
2. [Википедия] Статическая типизация
https://ru.wikipedia.org/wiki/Статическая_типизация
3. Статическая и динамическая типизация
<https://habr.com/ru/post/308484/>
4. Пирс Бенджамин. Типы в языках программирования. — 2010.



Вопросы для обсуждения

1. Основная идея однозначности выполнения операций. Способы достижения однозначности.
2. Достоинства и недостатки операционной однозначности.
3. Достоинства и недостатки динамической однозначности.
4. Достоинства и недостатки статической однозначности.
5. Связь между однозначностью и методами типизации
6. Нужна ли динамическая проверка типов данных в статически типизированных языках?
7. Для чего в статически типизированных языках могут применяться бестиповые решения?
8. Когда в статически типизированных языках появляется необходимость динамической проверки типов?

