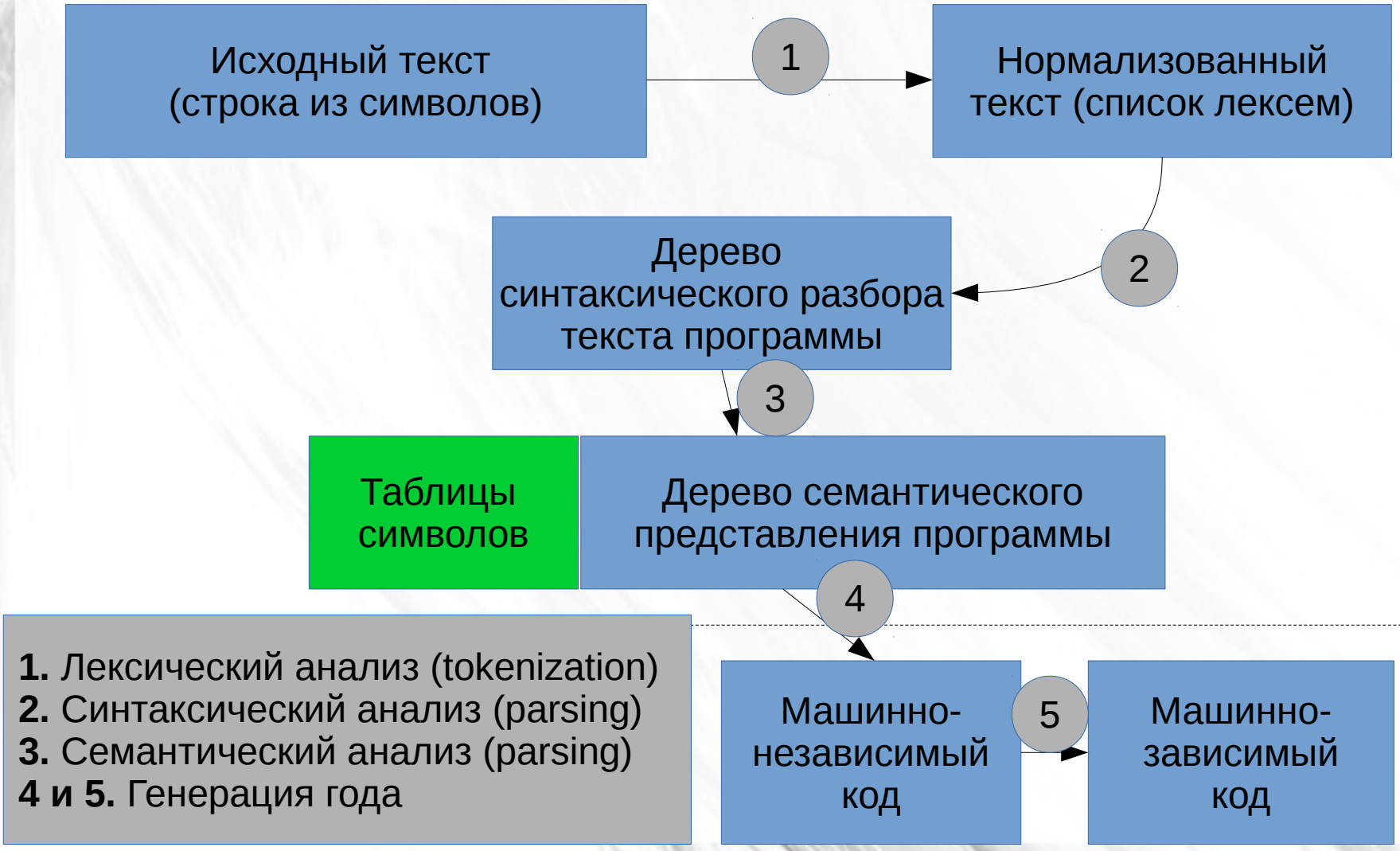
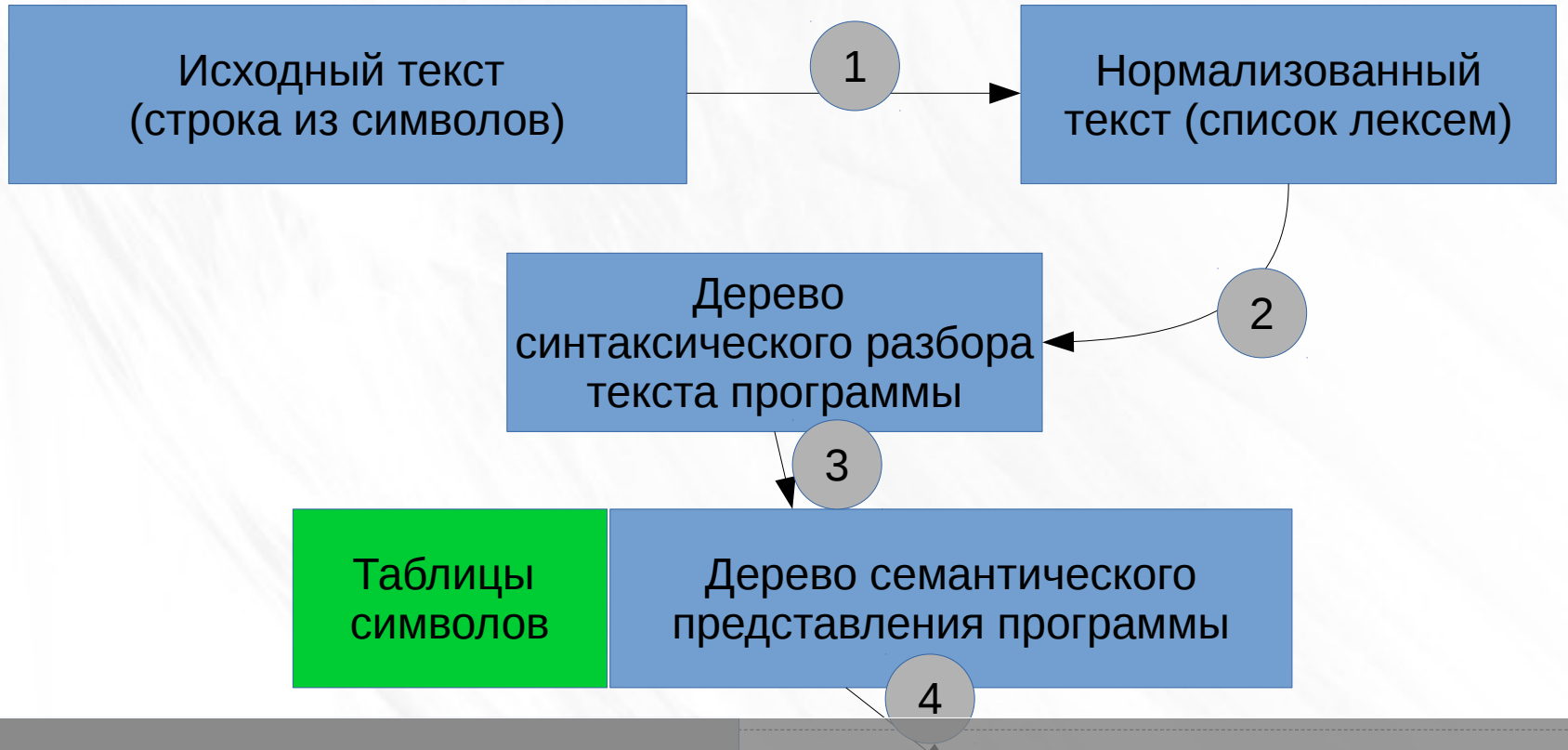
# Построение компиляторов и генерация кода LLVM

# Стадии компиляции

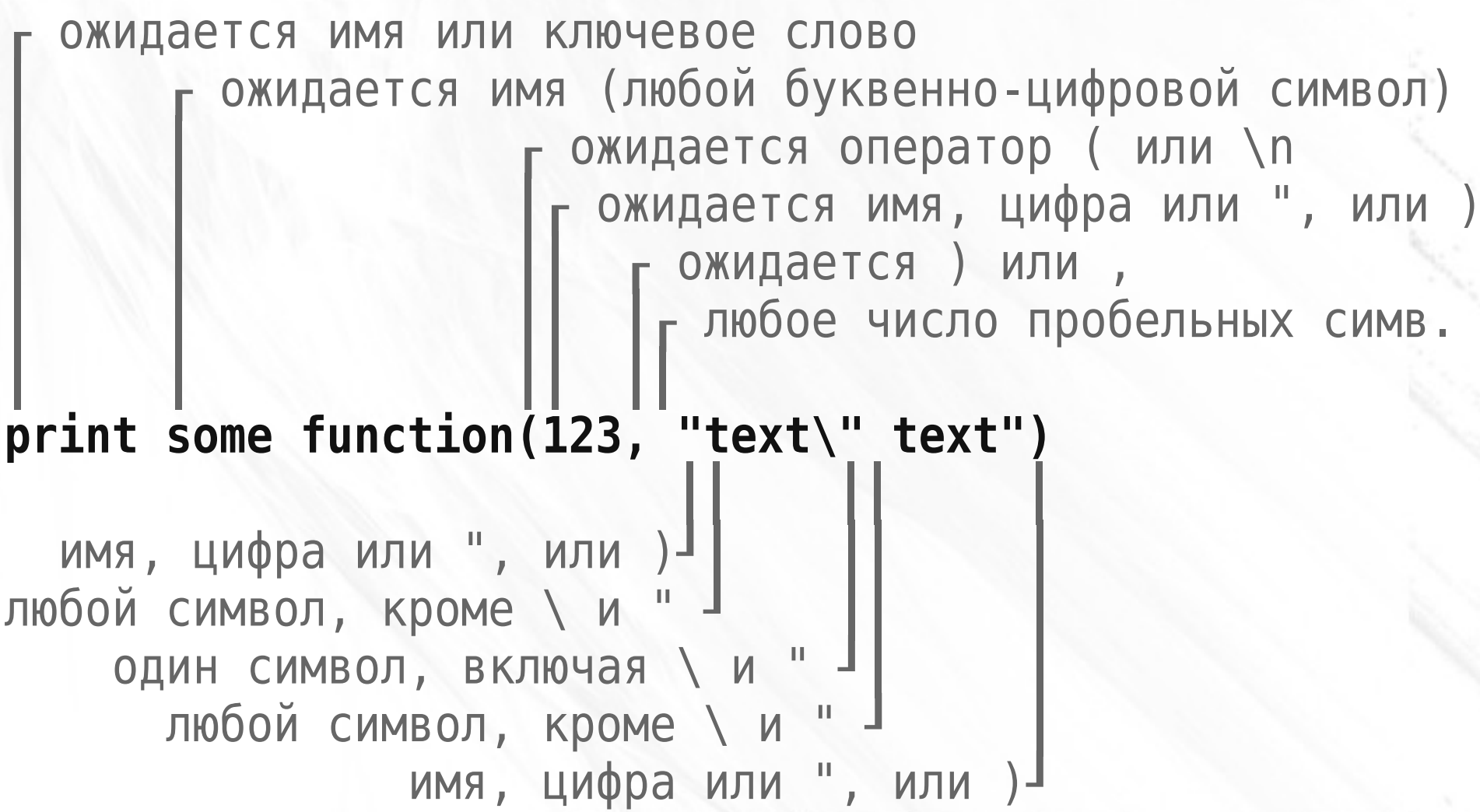



**1.** Лексический анализ (tokenization)
**2.** Синтаксический анализ (parsing)
**3.** Семантический анализ (parsing)
**4 и 5.** Генерация года

# Стадии компиляции



# Лексический анализ

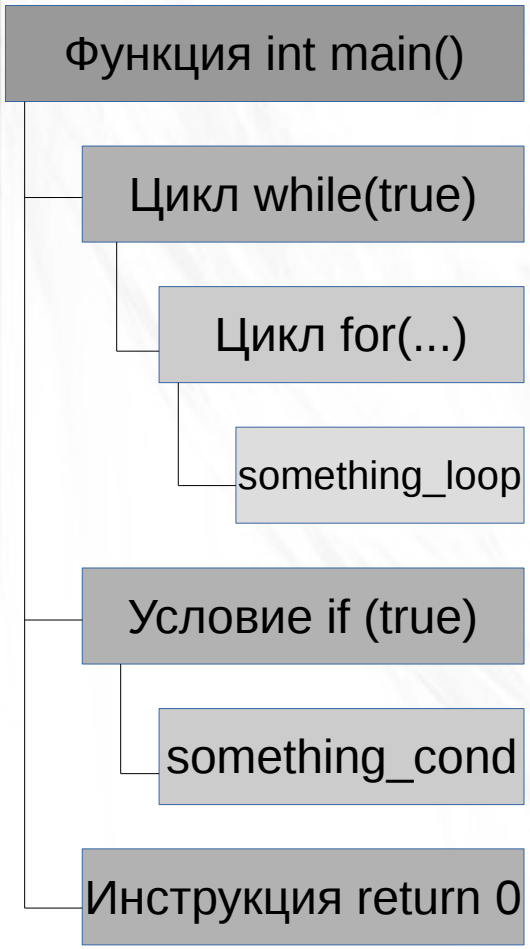


# Лексический анализ

```
print some function(123, "text\" text")
```

```
print           – идентификатор-ключевое слово
some function   – идентификатор
(               – оператор
123             – числовая константа
,               – оператор
"text\" text"   – литеральная константа
)               – оператор
```

# Синтаксический анализ текста



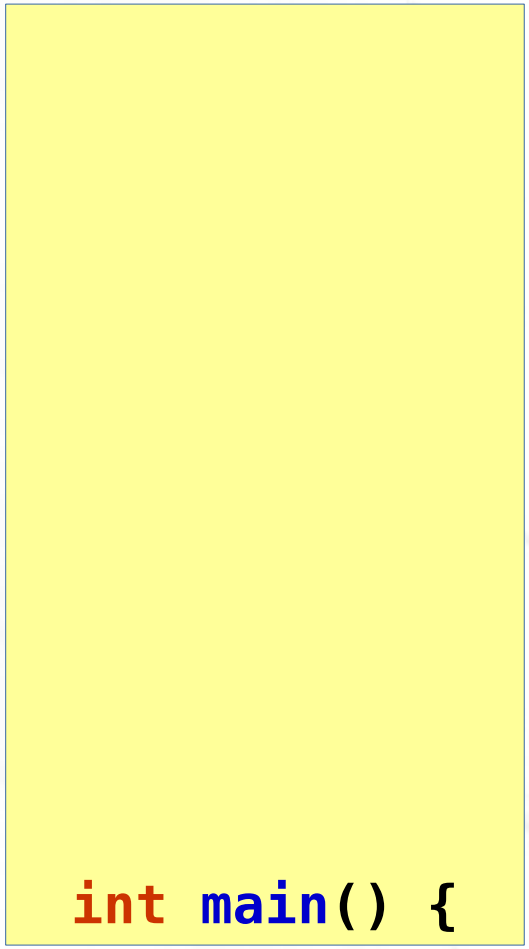

```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

# Синтаксический анализ текста

- Функция int main()
  - Цикл while(true)
    - Цикл for(...)
      - something_loop
  - Условие if (true)
    - something_cond
  - Инструкция return 0

```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**



# Синтаксический анализ текста

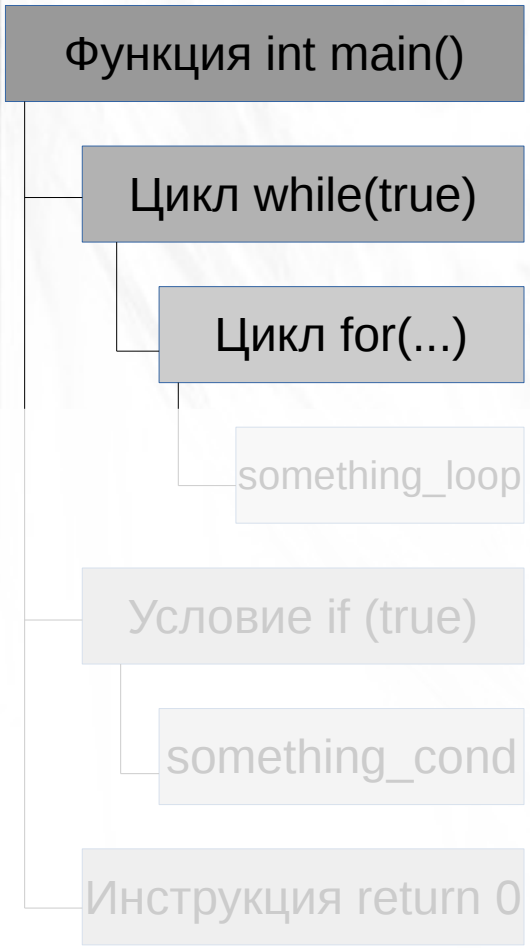
- Функция int main()
  - Цикл while(true)
    - Цикл for(...)
      - something_loop
    - Условие if (true)
      - something_cond
    - Инструкция return 0

```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

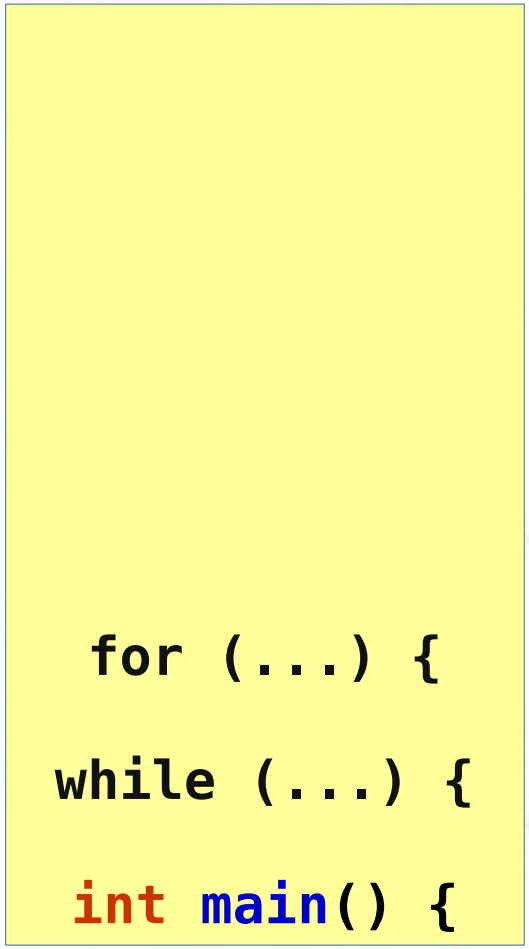
```
while (...) {

int main() {
```

# Синтаксический анализ текста

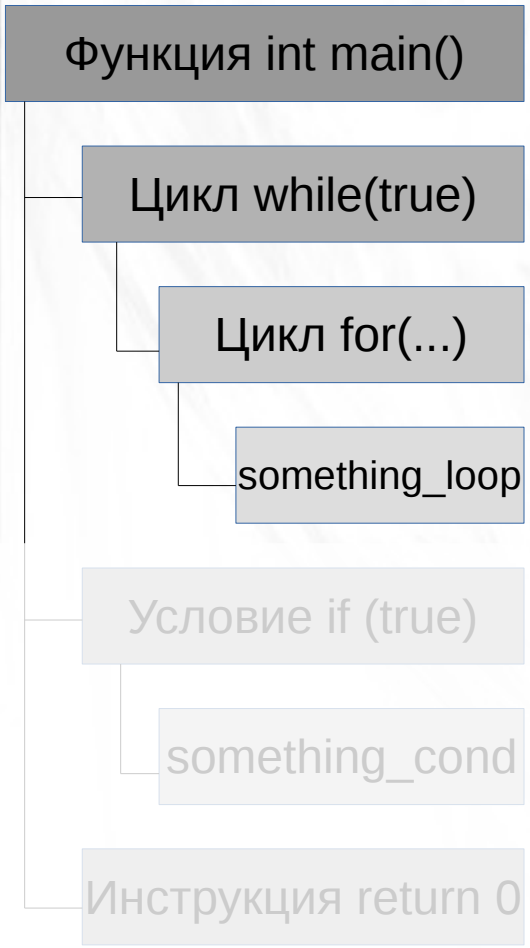



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

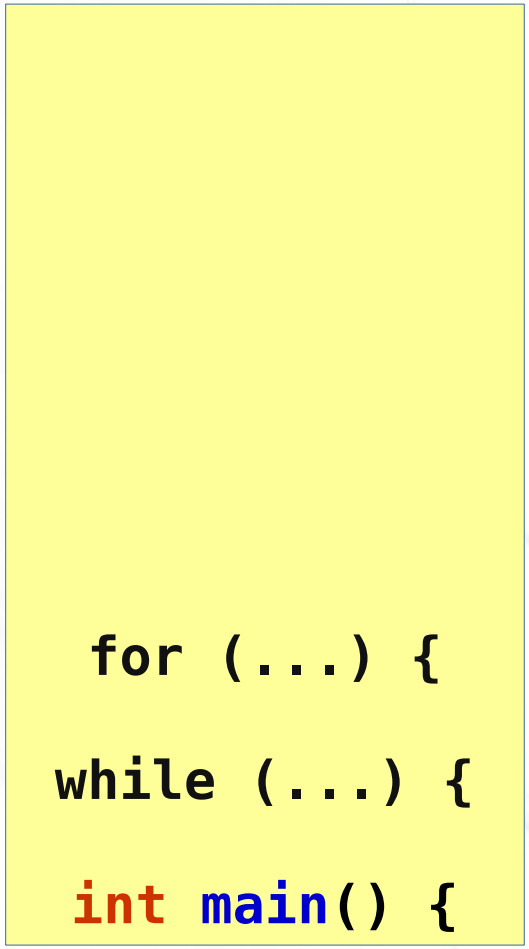


# Синтаксический анализ текста

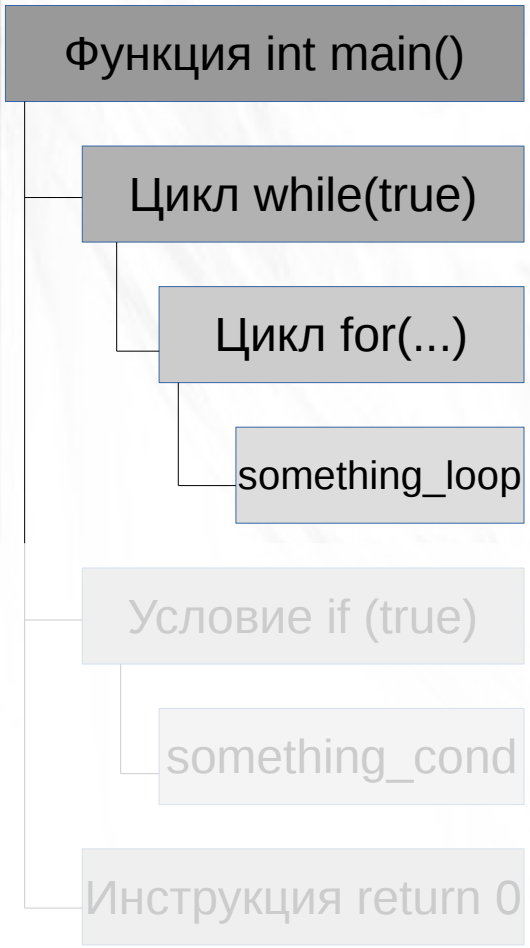



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

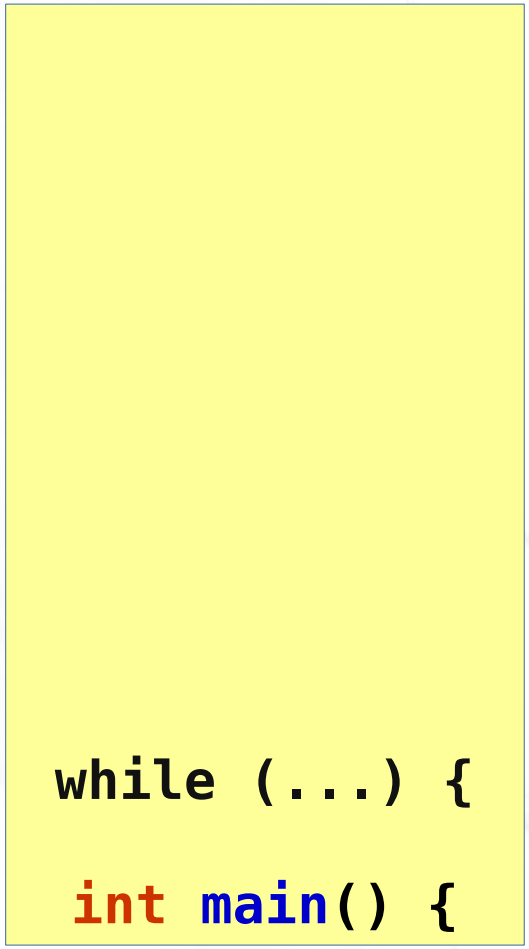


# Синтаксический анализ текста

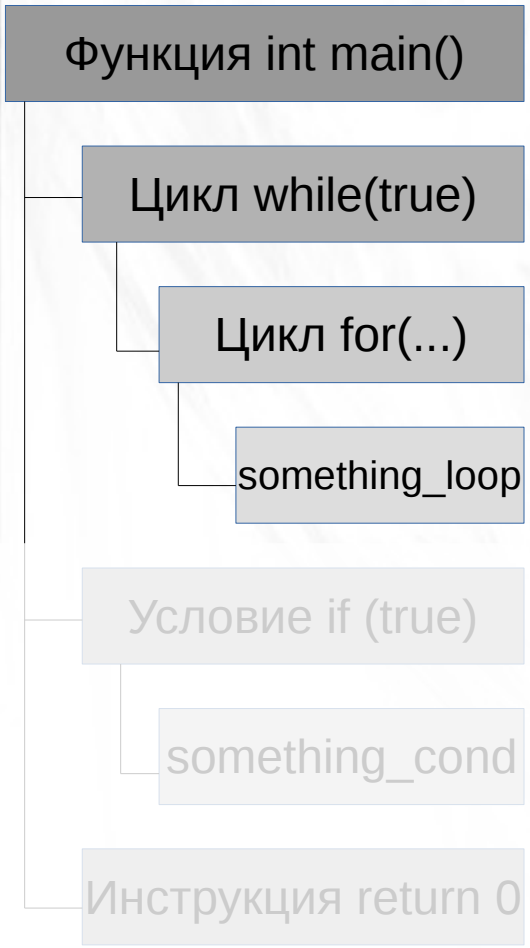



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

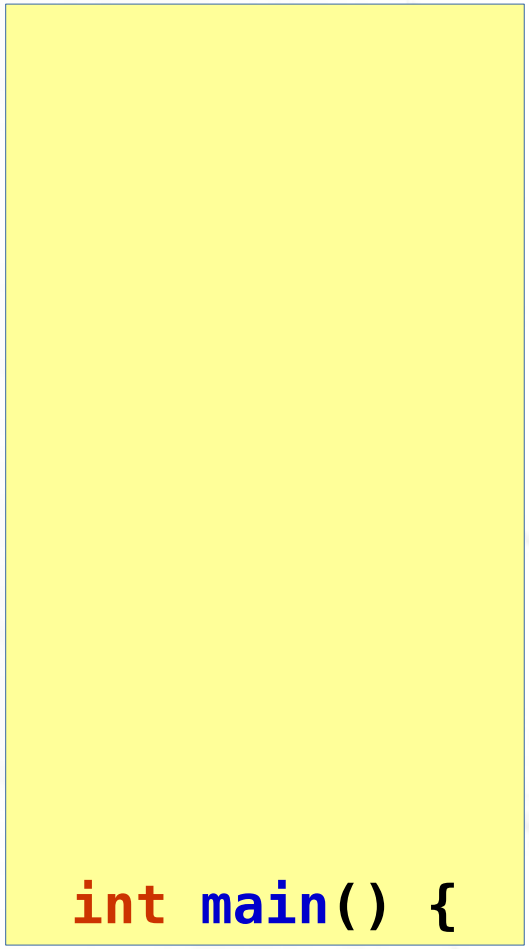


# Синтаксический анализ текста

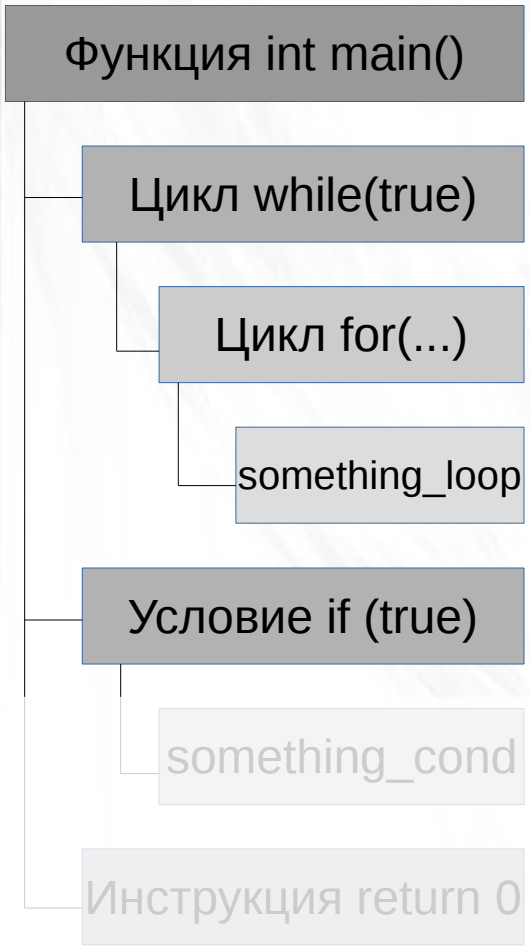



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

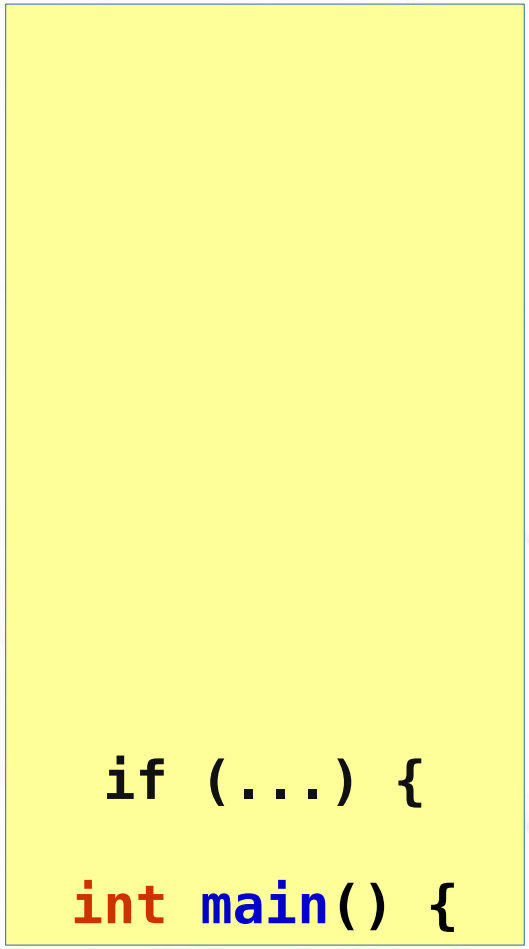


# Синтаксический анализ текста

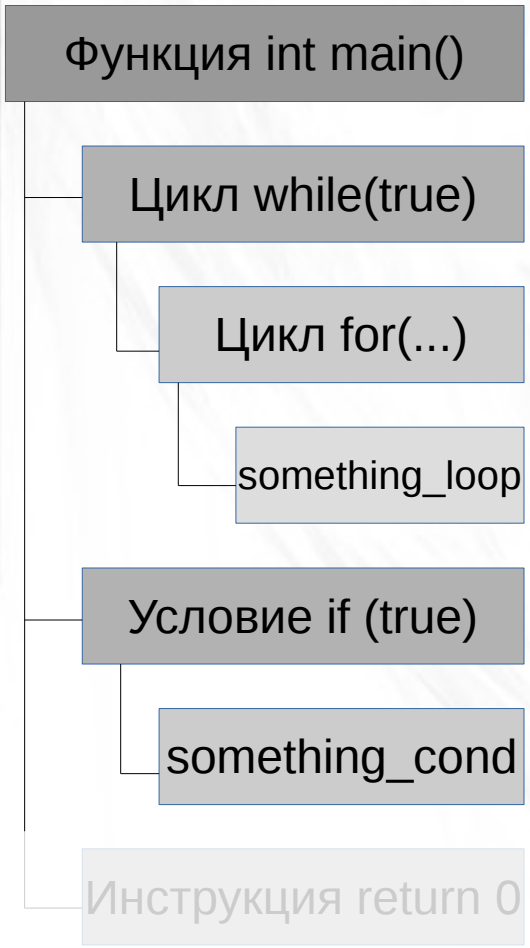



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

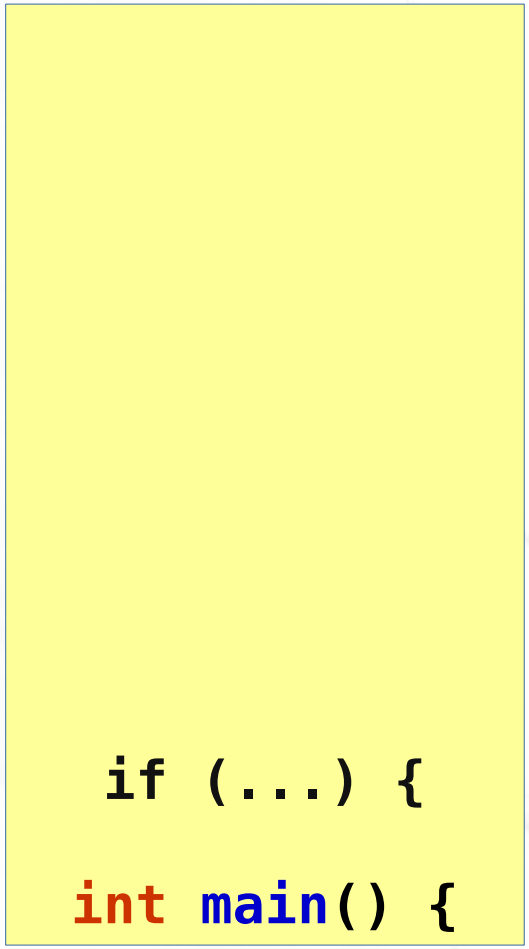


# Синтаксический анализ текста

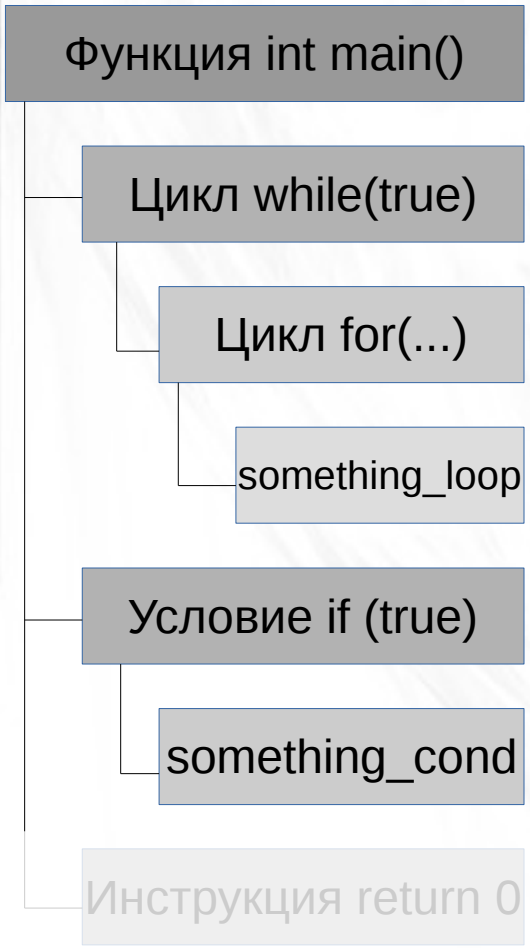



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

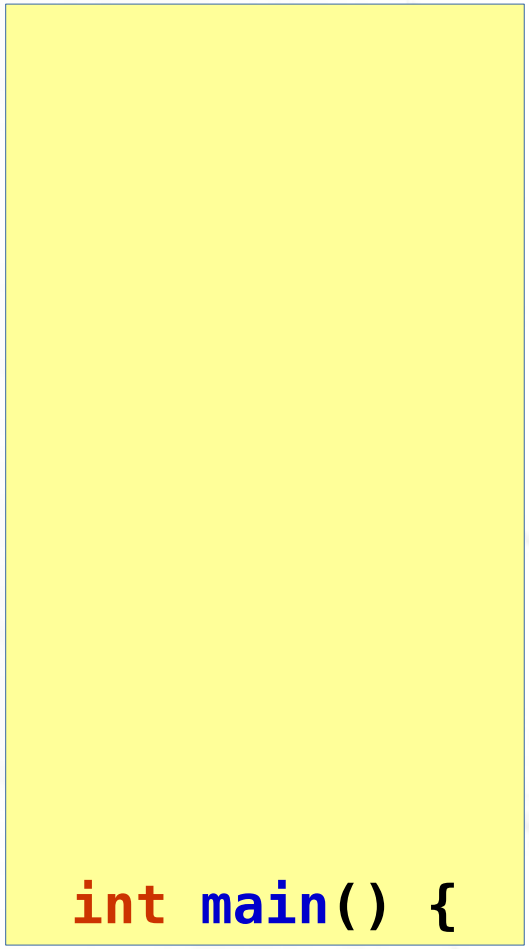


# Синтаксический анализ текста

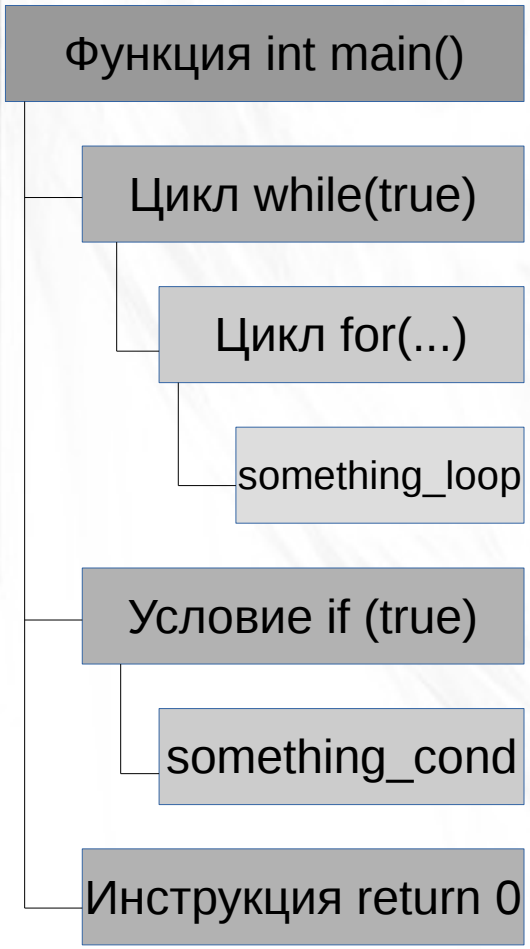



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

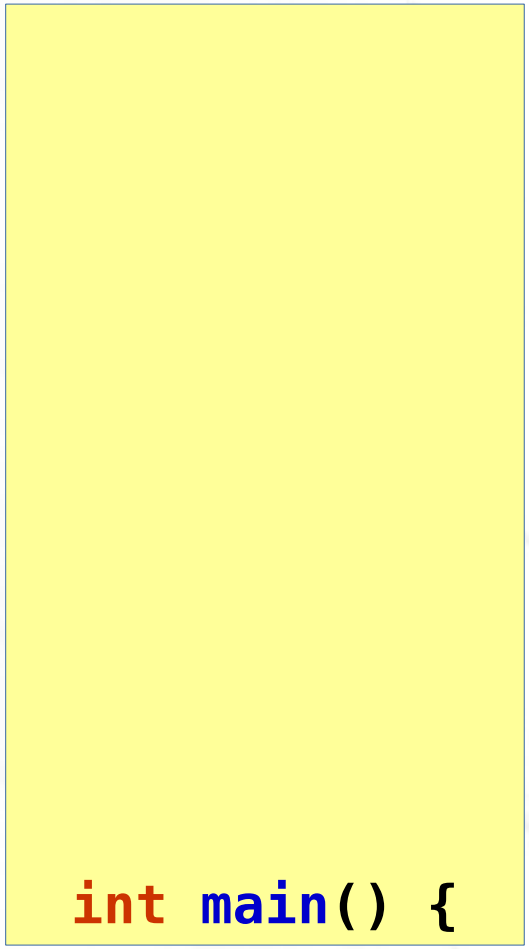


# Синтаксический анализ текста

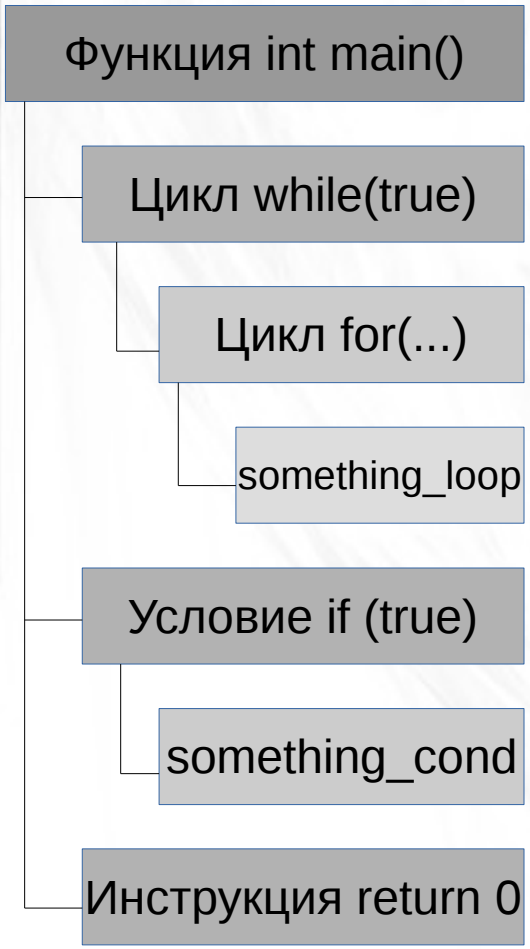



```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**



# Синтаксический анализ текста




```
int main()
{
  while (true)
  {
    for (int i=0;i<10;++i)
    {
      something_loop();
    }
  }
  if (true)
  {
    something_cond();
  }
  return 0;
}
```

**Стековый автомат:**

# Синтаксический анализ текста

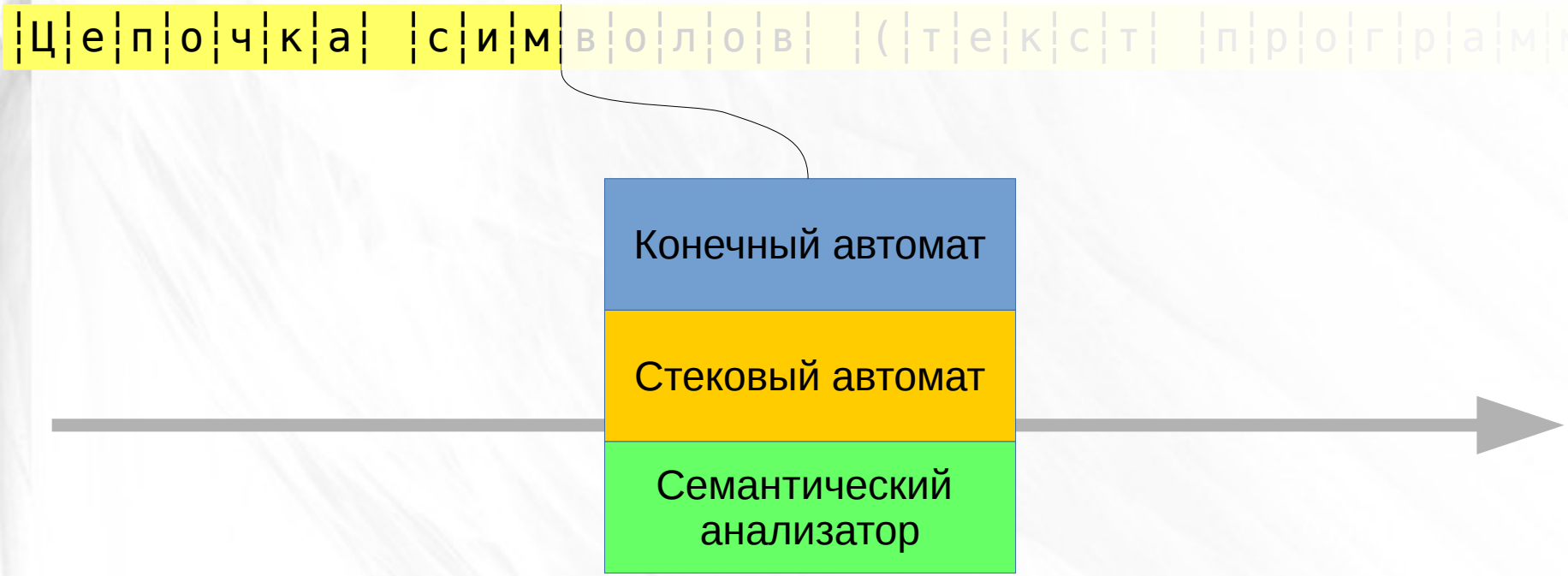
- По окончании синтаксического анализа стек должен быть пустым
- В процессе анализа из стека нельзя вытащить несуществующий элемент

Если это произошло — диагностируется ***синтаксическая ошибка***

# Семантический анализ

```
тип возвращаемого значения, из таблицы типов
   глобальное имя
      сигнатура функции
int main()
{  цикл, эквивалентный for (;;)
         значение uint8_t == 1
  while (true)
  {      перечислимый тип локальной переменной
            имя локальной переменной
               выражение типа bool
    for (int i=0;i<10;++i)
    {                  произвольная операция
      something_loop();
      вычисление rvalue выражения, пустое lvalue
```

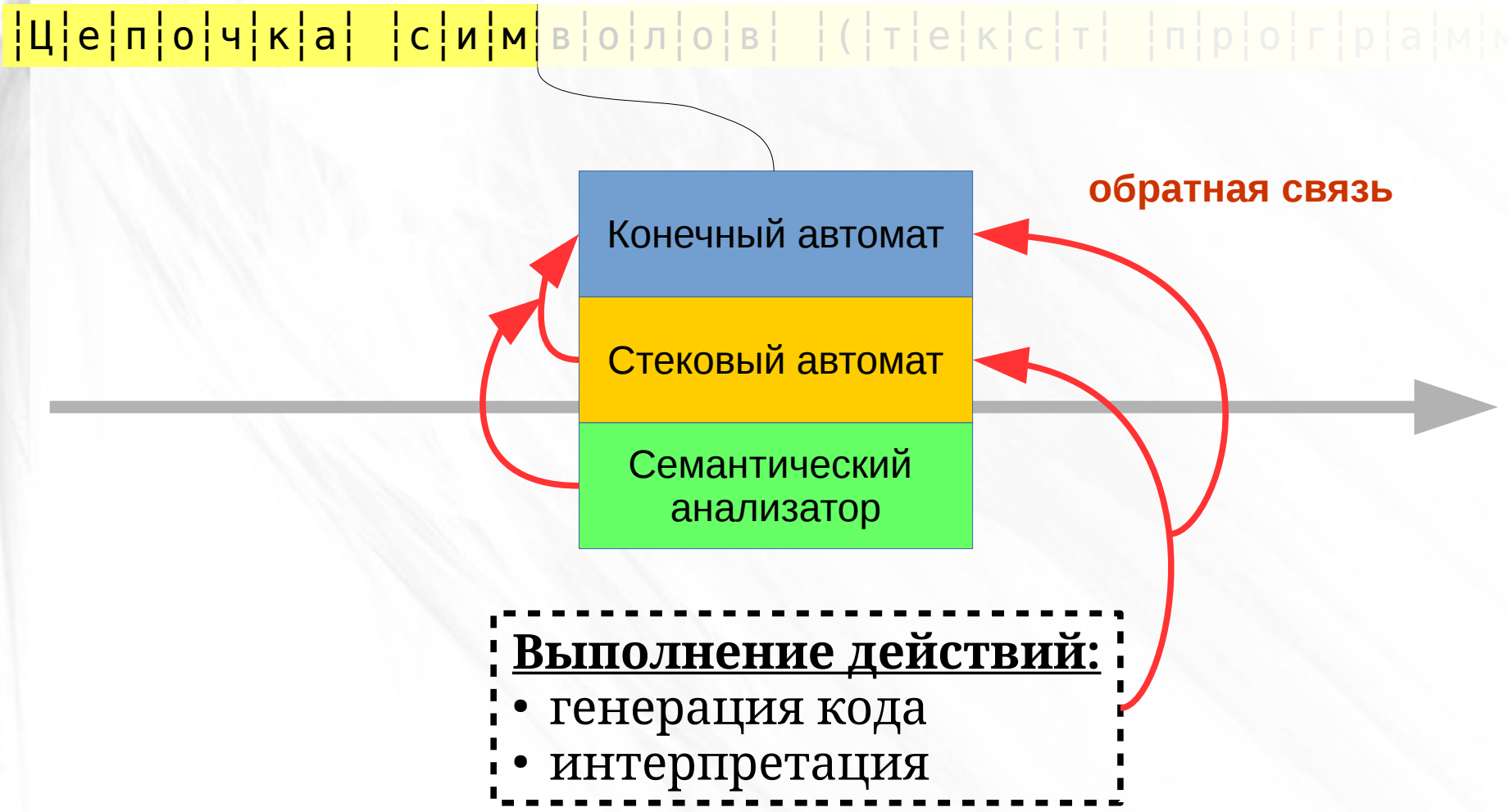
# Разбор текста на контекстно-свободном языке




**Выполнение действий:**

- генерация кода
- интерпретация

# Разбор текста на языке, который **не является** контекстно-свободным



# Perl

```
some_func / 25 ; # / ; die "Как это понимать???"
```

```
    time / 25 ; # / ; die "Это комментарий"
sin  / 25 ; # / ; die "Это сообщение о смерти"
```

---

- **time** — функция с 0 аргументами, следовательно следующий символ **/** — это опреатор
- **sin** — функция с 1 аргументом, следовательно следующий символ **/** — это НЕ опреатор, а часть составной лексемы

# Что дальше делать с деревом разбора программы?

- Обход от корня к вершинам, в процессе обхода вычислять выражения, и выполнять некоторые действия
  – ***интерпретация***
- Обход от корня к вершинам, в процессе обхода порождать новое представление
  – ***генерация кода (компиляция)***
- Обход произвольным образом, в процессе обхода выдавать диагностические сообщения
  – ***статический анализ***

# Машинный код

- Является эквивалентным представлением программы на ассемблере
- Может быть транслирован в текстовое представление (*дизассемблирован*)
- Является программой, которая распознается *конечным автоматом*

# Платформозависимость

- Длина команд (фиксированная – RISC, переменная – CISC)
- Номенклатура команд
- Поведение команд (пример: условные переходы x86 v.s. пропуск инструкций в ARM)
- Доступный набор регистров

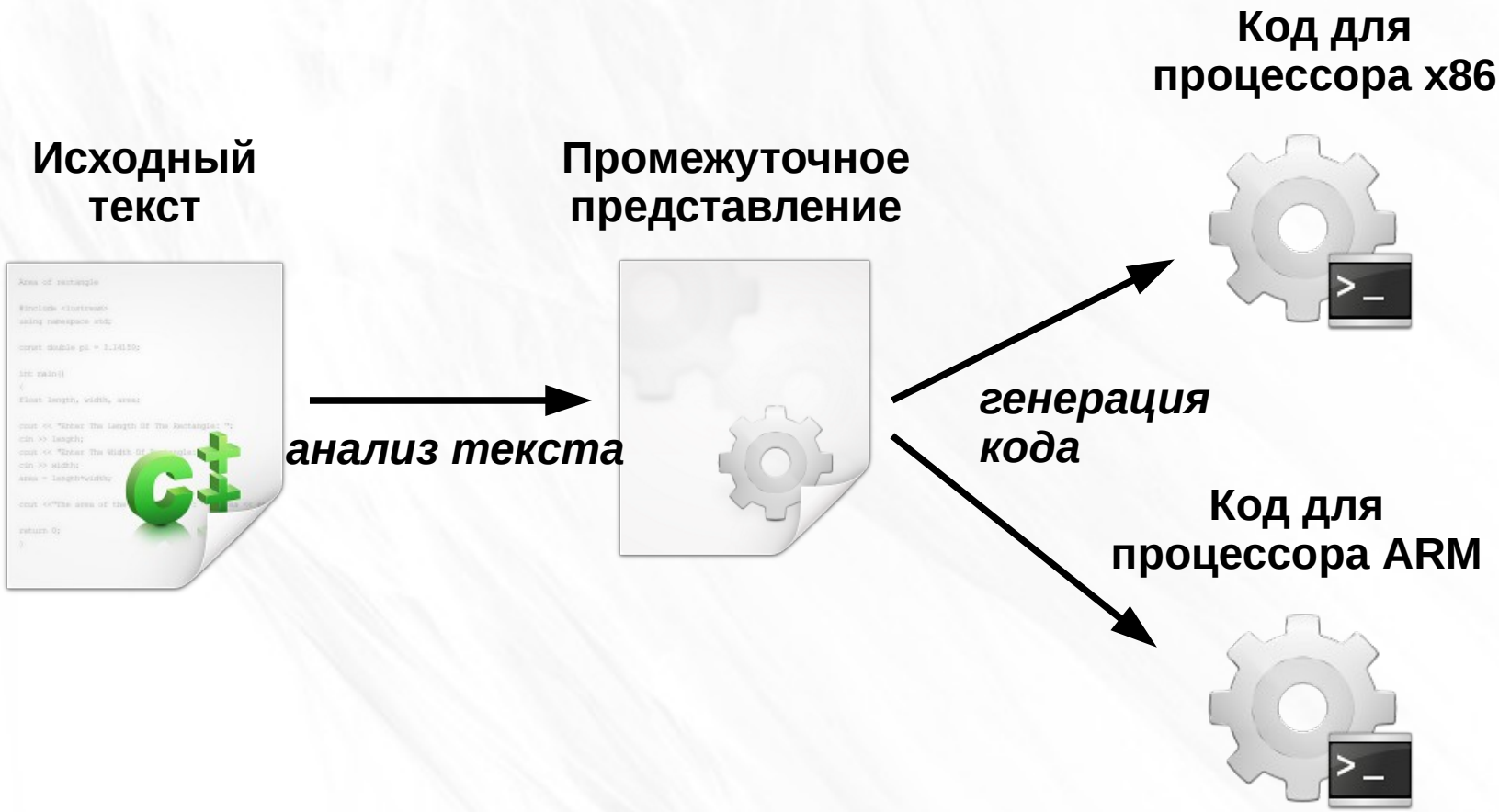
*На уровне процессора*

---

- Порядок работы со стеком
- Взаимодействие с окружающим миром

*На уровне ОС*

# Intermediate Language




***анализ текста*** **– Front-end компилятора**

***генерация кода*** **– Back-end компилятора**

# LLVM Intermediate Language

**L***ow* **L***evel* **V***irtual* **M***achine* – это низкоуровневый, не зависящий от процессора язык представления программ.

**http://www.llvm.org/**

# Команды для работы с LLVM

**`lli`** – интерпретатор LLVM-кода

**`llc`** – транслятор в машинный код (текст на ассемблере)

**`llvm-as`** – транслятор из текстового (`.ll`) в бинарное (`.bc`) представление

**`llvm-dis`** – транслятор из бинарного (`.bc`) в текстовое (`.ll`) представление

**`clang -c -emit-llvm ФАЙЛ.c`** – компиляция из C/C++ в LLVM, а не в машинный код

**`clang ФАЙЛ.ll`** – компиляция из LLVM в программу

**`rustc -c -emit llvm-ir ФАЙЛ.rs`** – компиляция из Rust в LLVM (текстовое представление)

# Пример: вычисление степени

```c
#include <stdio.h>
int my_power(int base, unsigned pow)
{
    int result = 1;
    for (unsigned i=0; i<pow; ++i) {
        result *= base;
    }
    return result;
}
int main()
{
    int result = my_power(2, 8);
    printf(
     "2 in power of 8: %d\n",
     result
    );
    return 0;
}
```

```
shell> clang -c -std=c99 \
   -emit-llvm my_power.c
# появился файл my_power.bc
shell> lli my_power.bc
2 in power of 8: 256

shell> llvm-dis my_power.bc
# появился файл my_power.ll
shell> lli my_power.ll
2 in power of 8: 256

shell> clang my_power.ll
# появился файл a.out
shell> ./a.out
2 in power of 8: 256
```

# Пример: вычисление степени

```
define i32 @my_power(i32 %base, i32 %pow) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %result = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %base, i32* %1, align 4
  store i32 %pow, i32* %2, align 4
  store i32 1, i32* %result, align 4
  store i32 0, i32* %i, align 4
  br label %3

; <label>:3        ; preds = %11, %0
  %4 = load i32* %i, align 4
  %5 = load i32* %2, align 4
  %6 = icmp ult i32 %4, %5
  br i1 %6, label %7, label %14
```

```
; <label>:7        ; preds = %3
  %8 = load i32* %1, align 4
  %9 = load i32* %result, align 4
  %10 = mul nsw i32 %9, %8
  store i32 %10, i32* %result, align 4
  br label %11

; <label>:11       ; preds = %7
  %12 = load i32* %i, align 4
  %13 = add i32 %12, 1
  store i32 %13, i32* %i, align 4
  br label %3

; <label>:14       ; preds = %3
  %15 = load i32* %result, align 4
  ret i32 %15
}
```

# Си v.s. C++ v.s. Rust

```cpp
#include <iostream>
int my_power(int base, unsigned pow)
{
  int result = 1;
  for (unsigned i=0; i<pow; ++i) {
    result *= base;
  }
  return result;
}

int main()
{
  int result = my_power(2, 8);
  std::cout
    << "2 in power of 8: "
    << result << std::endl;
  return 0;
}
```

```rust
fn my_power(base: i32, pow: u32) -> i32
{
    let mut result = 1;
    for _ in 0..pow {
        result *= base;
    }
    result
}

fn main()
{
    let result = my_power(2, 8);
    println!(
      "2 in power of 8: {}",
      result
    );
}
```

# Си v.s. C++ vs. Rust

C++

```
define i32 @_Z8my_powerij(i32 %base, i32 %pow) {
  . . . .
```

```
extern "C" int my_power(int base, unsigned pow)

define i32 @my_power(i32 %base, i32 %pow) {
  . . . .
```

Rust

```
define i32 @_ZN8my_power20h2d9a7b8a5ac225adeaaE(i32, i32) {
  . . . .
```

```
#[no_mangle]
fn my_power(base: i32, pow: i32) -> i32

define i32 @my_power(i32, i32) {
  . . . .
```

# Структура программы на низком уровне

LLVM

```
@.str = private unnamed_addr
  ↘ constant [15 x i8]
  ↘ c"Hello, World!\0A\00",
  ↘ align 1

define i32 @main() #0 {
  %1 = tail call i32 @puts(
    ↘ i8* getelementptr
    ↘ inbounds (
    ↘ [15 x i8]* @.str,
    ↘ i64 0, i64 0)) #2
  ret i32 %1
}
```

Ассемблер x86_64

```
        .section .rodata
.LC0:   .string "Hello, World!"
        .text
        .globl main
        .type  main, @function
main:   pushq  %rbp
        movq   %rsp, %rbp
        movl   $.LC0, %edi
        call   puts
        popq   %rbp
        ret
```

```
#include <stdio.h>
int main() { printf("Hello, World!\n"); return 0; }
```

# Использование регистров

```
int func(int a, int b) {
  int res = 10+(a+b)*(a*b)+(a*b);
  return res-a;
}
```

LLVM:

```
define i32 @func(i32 %a, i32 %b)
{
  %1 = add nsw i32 %b, %a
  %2 = mul nsw i32 %b, %a
  %3 = mul nsw i32 %1, %2
  %4 = sub i32 10, %a
  %5 = add i32 %4, %2
  %6 = add i32 %5, %3
  ret i32 %6
}
```

ARMv7:
gcc -Os

```
mul r3, r1, r0
add r1, r0, r1
mla r3, r3, r1, r3
add r3, r3, #10
rsb r0, r0, r3
bx  lr
```

x86
gcc -O0 -m32
-fomit-frame-pointer

```
subl    $16, %esp
movl    24(%esp), %eax
movl    20(%esp), %edx
addl    %eax, %edx
movl    20(%esp), %eax
imull   24(%esp), %eax
imull   %edx, %eax
leal    10(%eax), %edx
movl    20(%esp), %eax
imull   24(%esp), %eax
addl    %edx, %eax
movl    %eax, 12(%esp)
movl    20(%esp), %eax
movl    12(%esp), %edx
subl    %eax, %edx
movl    %edx, %eax
addl    $16, %esp
ret
```

# Явное выделение памяти в стеке

```c
#include <stdio.h>
main() {
    int val;
    scanf("%d", &val);
}
```

```
@.str = private unnamed_addr
  ↘ constant [3 x i8] c"%d\00",
  ↘ align 1

define i32 @main() {
  %val = alloca i32, align 4
  %1 = call i32 (i8*, ...)*
  ↘ @scanf(i8* getelementptr
  ↘ inbounds ([3 x i8]* @.str,
  ↘ i64 0, i64 0), i32* %val)
  ret i32 0
}
```

```
.LC0:   .string    "%d"
main:   subq   $24, %rsp
        movl   $.LC0, %edi
        xorl   %eax, %eax
        leaq   12(%rsp), %rsi
        call   scanf
        addq   $24, %rsp
        ret
```

```
shell> man 3 alloca

NAME
   alloca - allocate memory
   that is automatically freed

SYNOPSIS
   #include <alloca.h>
   void *alloca(size_t size);
```

# Явное выделение памяти в стеке: C99/C2011

```c
#include <stdio.h>          /* clang -std=c99 source.c */
int main() {
    int N;    scanf("%d", &N);    int arr[N];
    for (int i=0; i<N; ++i) arr[i] = i*2;
    for (int i=0; i<N; ++i) printf("%d ", arr[i]);
    return 0; }
```

*Это легальная конструкция в Си, но не C++*

```
define i32 @main() {
  %N = alloca i32, align 4
  %1 = call i32 (i8*, ...)*
   ↘ @__isoc99_scanf(i8*
   ↘ getelementptr inbounds
   ↘ ([3 x i8]* @.str, i64 0,
   ↘   i64 0), i32* %N)
  %2 = load i32* %N, align 4
  %3 = zext i32 %2 to i64
  %4 = alloca i32,
       ↘ i64 %3, align 16
```

```
. . . . .
; gcc -S -std=c99 -m32 -Os
call   __isoc99_scanf
movl   -28(%ebp), %edx
addl   $16, %esp
leal   18(,%edx,4), %eax
andl   $-16, %eax
subl   %eax, %esp
. . . . .
```

# Условные блоки и циклы

**LLVM**

```
define функция() {
[Метка 1]:
  . . . .
  . . . .
  терминатор
Метка 2:
  . . . .
  . . . .
  терминатор
}
```

**Ассемблер**

```
функция:
[Метка 1]:
  . . . .
  . . . .
  . . . .
Метка 2:
  . . . .
  . . . .
  . . . .
```

# Условные блоки и циклы

## Основные терминальные инструкции

- `ret [<type> <value>]`
- `br <label>`
- `br i1 <cond>, label <iftrue>, label <iffalse>`
- `switch <type> <value>, label <default>,`
  `[<type1> <value1>, label <label1> . . .]`

- Каждый блок начинается с метки (начало функции - опционально)
- Каждый блок **обязан** завершаться терминальной инструкцией

# Цикл while (true)

```
while (true)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
Loop_Start:
  <statement 1>
  . . . .
  <statement N>
  br %Loop_Start
```

# Цикл while (true)

```
while (true)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
Loop_Start:
  <statement 1>
  . . . .
  <statement N>
  br %Loop_Start
```

# Цикл while (условие)

```
while (<cond>)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
Loop_Start:
  %cond = . . .
  br i1 %cond,
   ↘ label %Loop_Body,
   ↘ label %Loop_End
Loop_Body:
  <statement 1>
  . . . .
  <statement N>
  br %Loop_Start
Loop_End:
  . . . .
```

# Цикл for ( ; ; )

```
for (<init>;<cond>;<incr>)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
    <init statements>
Loop_Start:
    %cond = . . .
    br i1 %cond,
      ↘ label %Loop_Body,
      ↘ label %Loop_End
Loop_Body:
    <statement 1>
    . . . .
    <statement N>
    <increment statement>
    br %Loop_Start
Loop_End:
    <cleanup statements>
    . . . .
```

# Дополнительные источники

- Обзорная статья на Хабре про LLVM [https://habrahabr.ru/post/277717/]
- LLVM Language Reference Manual [http://llvm.org/docs/LangRef.html]
- Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции (в 2-х томах). М.:Мир, 1978