

КЛОЧКОВА ЯРОСЛАВА

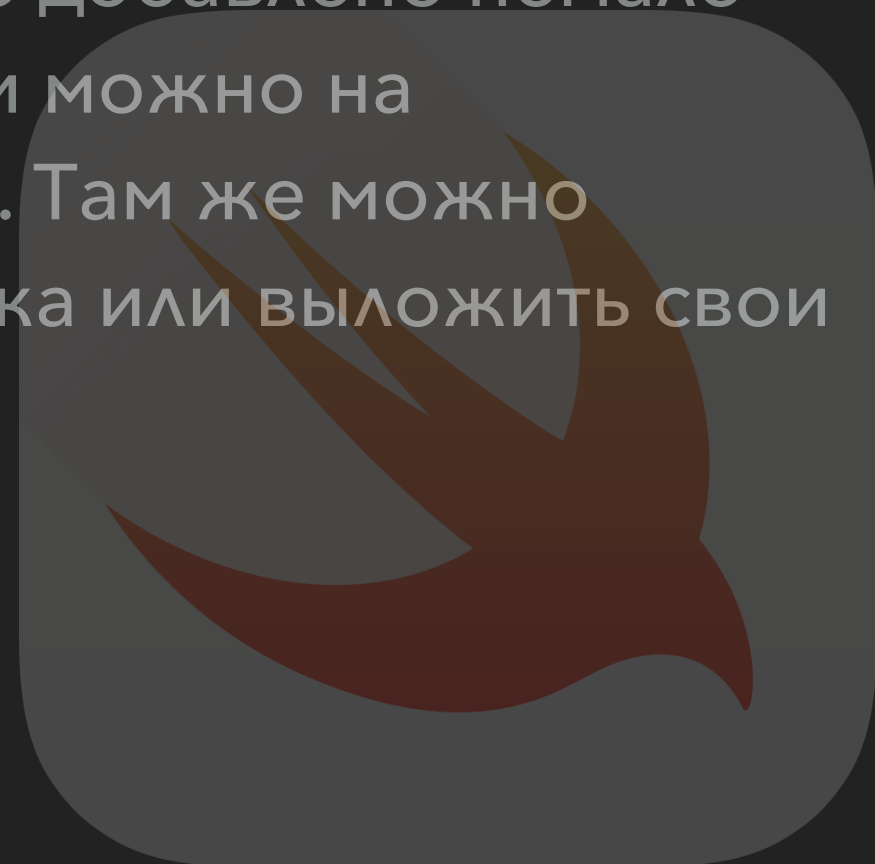
КМБО-02-15

УСТРОЙСТВО КОМПИЛЯТОРА SWIFT

- ▶ Swift - открытый мультипарадигмальный компилируемый язык программирования общего назначения. Создан компанией Apple в первую очередь для разработчиков iOS и macOS. Swift работает с фреймворками Cocoa и Cocoa Touch и совместим с основной кодовой базой Apple, написанной на Objective-C.
- ▶ Swift уже более двух лет является проектом с открытым исходным кодом. За это время в него было добавлено немало улучшений от комьюнити. Следить за ними можно на специальном **сайте***, а так же на **форуме****. Там же можно обсудить предложения по улучшению языка или **выложить свои идеи**.

* <https://apple.github.io/swift-evolution/>

** <https://forums.swift.org>



ПЛЮСЫ SWIFT (В СРАВНЕНИИ С OBJECTIVE-C)

- ▶ Проще в изучении, что способствует ускорению цикла разработки приложений;
- ▶ Автоматическое управление памятью;
- ▶ В 2.6 раз быстрее Objective-C; в 8 раз быстрее Python 2.7;
- ▶ Безопаснее;

SWIFT-КОД НА ПРИМЕРЕ ИГРЫ FLAPPY BIRD

```
class GameViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        if let scene = GameScene.unarchiveFromFile("GameScene") as? GameScene {
            // Configure the view.
            let skView = self.view as! SKView
            skView.showsFPS = true
            skView.showsNodeCount = true

            /* Sprite Kit applies additional optimizations to improve rendering performance */
            skView.ignoresSiblingOrder = true

            /* Set the scale mode to scale to fit the window */
            scene.scaleMode = .aspectFill

            skView.presentScene(scene)
        }
    }

    override var shouldAutorotate : Bool {
        return true
    }

    override var supportedInterfaceOrientations : UIInterfaceOrientationMask {
```

SWIFT STANDARD LIBRARY

Помимо компилятора и стандартной библиотеки в открытом доступе находится множество других подпроектов. Некоторые из них перечислены ниже.

- ▶ SourceKit - фреймворк для поддержки IDE: индексация, подсветка синтаксиса, автодополнение кода и так далее.
- ▶ SourceKit LSP - реализация LSP для Swift, созданная на основе SourceKit.
- ▶ Swift Package Manager - пакетный менеджер для проектов на Swift.
- ▶ Foundation - порт библиотеки Foundation, которая является одной из основных для ОС от Apple под сторонние платформы.
- ▶ Libdispatch (GCD) - GCD для сторонних платформ.
- ▶ Playground Support - в проект входят два фреймворка – PlaygroundSupport и PlaygroundLogger. Они обеспечивают взаимодействие с Xcode и красивое отображение данных соответственно.
- ▶ libcxx - реализация стандартной библиотеки C++.

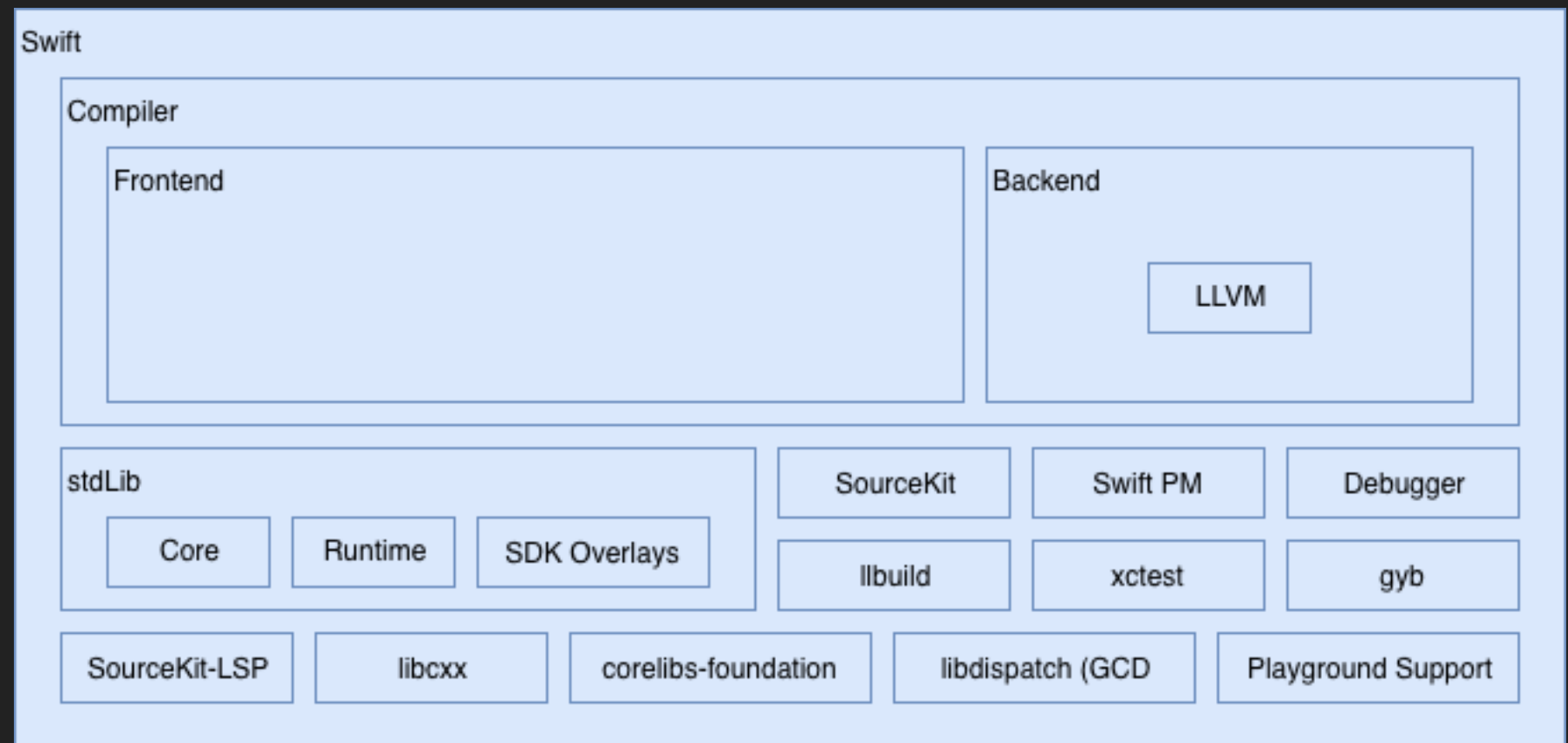
КОМПИЛЯТОР SWIFT

- ▶ frontend - преобразование кода в промежуточное представление, с которым удобно работать компилятору;
- ▶ middlend - выполняется оптимизация;
- ▶ backend - генерация кода в машинный код;

КОМПИЛЯТОР SWIFT

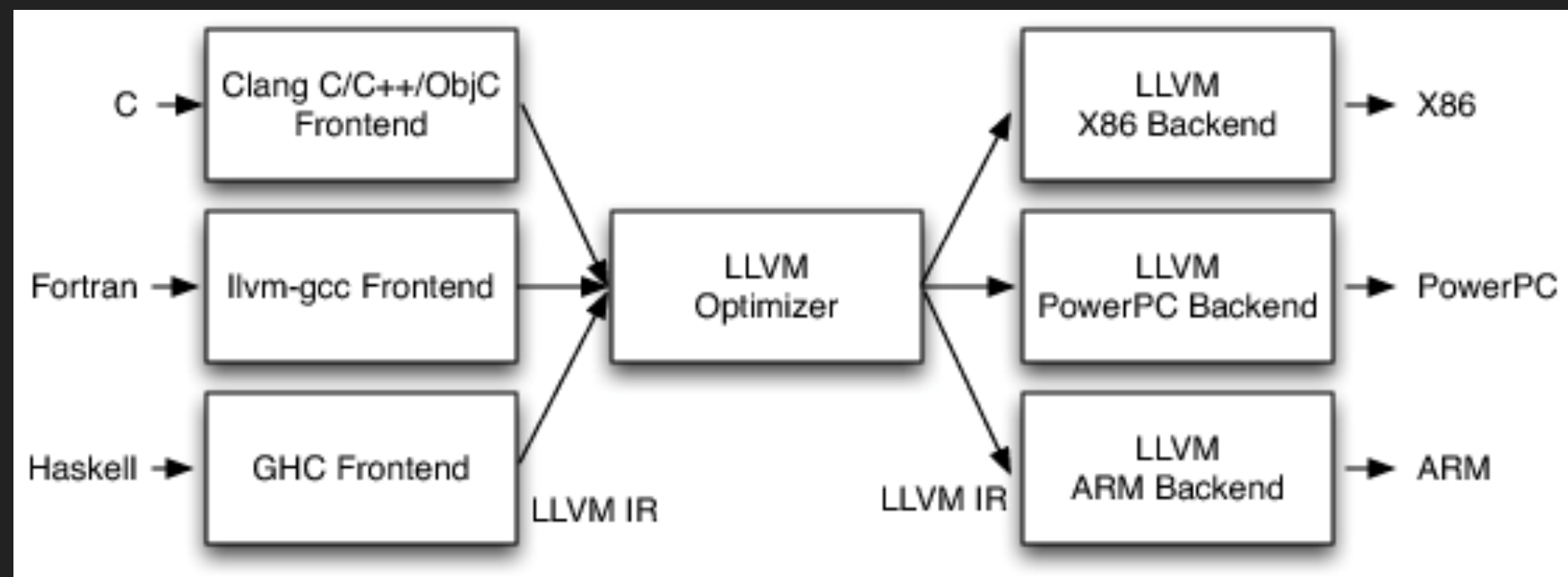
LLVM

В качестве бекенда в компиляторе Swift используется LLVM. LLVM – это большой проект, включающий в себя множество технологий. В его основе лежит intermediate representation (IR). Это универсальное промежуточное представление кода, которое можно преобразовать в исполняемый код на любой платформе, поддерживаемой LLVM.



КОМПИЛЯТОР SWIFT

Если появится новая архитектура, то достаточно будет добавить в LLVM генерацию машинного кода из IR под эту платформу. После этого все языки, для которых есть компилятор с генерацией IR, будут поддерживать эту архитектуру.



КОМПИЛЯТОР SWIFT

IR имеет три вида отображения:

- ▶ Дерево объектов в памяти.
- ▶ Текстовое представление.
- ▶ Сериализованный битовый формат «биткод».

FRONTEND

Задача фронтенда – сгенерировать из исходного кода промежуточное представление и передать его в LLVM. Этот процесс можно разделить на 8 шагов. Результат работы почти каждого из них можно вывести, передав в компилятор специальный параметр.

Реализация реализации компилятора для примитивного «языка программирования», который содержит только «области видимости» и число. Единственная его функция – вывод числа (если оно есть) в консоль. Например, в результате выполнения этого «кода» будет выведено число 678:

```
{{{678}}}
```

FRONTEND – ПРИМЕРЫ КОДА

Каждая область видимости состоит из открывающей скобки, содержимого и закрывающей скобки. Это можно записать так:

```
scope ::= open_brace x close_brace
open_brace ::= "{"
close_brace ::= «»
```

Содержимым может являться такая же область видимости, число, либо ничего, обозначенное тут:

```
scope ::= open_brace x close_brace
x ::= scope | number | <empty>
open_brace ::= "{"
close_brace ::= «»
```

Символ | означает «или». Число состоит из одной или более цифр:

```
s ::= open_brace x close_brace
```

```
x ::= scope | number | <empty>
```

```
open_brace ::= "{"
```

```
close_brace ::= «»
```

```
number ::= digit | number digit
```

```
digit ::= "0" | "1" | "2" | "3" |  
"4" | "5" | "6" | "7" | "8" | "9"
```

FRONTEND

Такую запись называют формой Бэкуса-Наура (БНФ), а совокупность всех правил – грамматикой языка или синтаксисом.

Фигурные скобки обозначают повторения. Такая запись означает, что A либо пусто, либо равно любому количеству B :

$$A ::= \{B\}$$

Квадратные скобки используются для условного вхождения. Такая запись означает, что A либо пусто, либо равно B :

$$A ::= [B]$$

Используя РБНФ грамматику компилятора скобок, можно преобразовать в такой вид:

```
scope ::= open_brace [scope | number] close_brace
```

```
open_brace ::= "{"
```

```
close_brace ::= «}"
```

```
number ::= digit {digit}
```

```
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

LEXER

С точки зрения компилятора файл с исходным кодом – это поток случайных символов, а может и какого-то мусора. Поэтому первый шаг – преобразование этих случайных символов в слова, которые допустимо использовать в языке программирования.

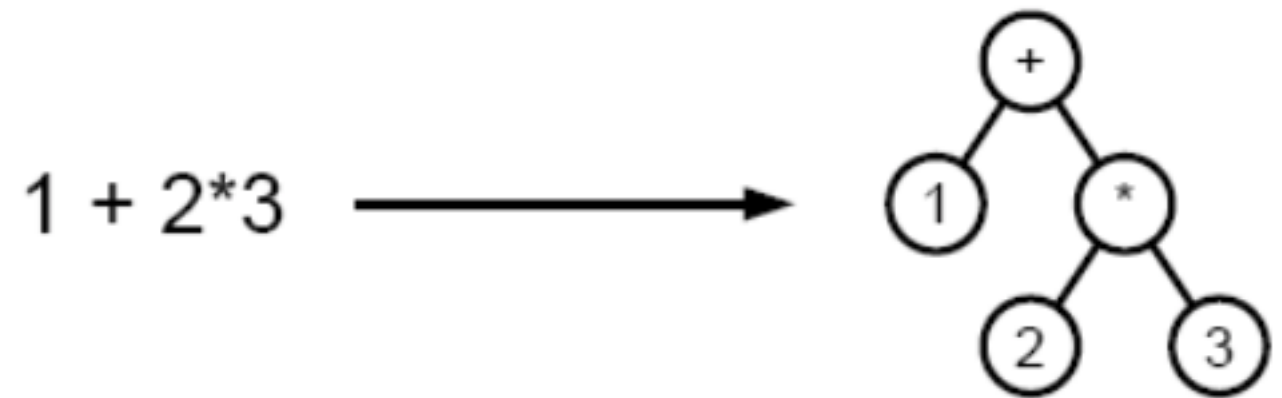
Этим занимается лексический анализатор (лексер/токенизатор). Слова, которые он ищет, называются лексемами или токенами. Поэтому этот процесс также называют токенизацией.

Лексер производит анализ в соответствии с синтаксисом языка. Сканируя символ за символом, он находит соответствие исходного кода и правой части записи, а затем генерирует соответствующий токен из левой части.

PARSER

Парсер осуществляет синтаксический анализ. На вход ему передается последовательность токенов, а результатом работы является AST.

AST – это граф, в котором вершинами являются операторы, а листьями – их операнды. Например, выражение $1 + (2 * 3)$ состоит из двух операторов: сложения и умножения. Первым операндом сложения является 1, а вторым – результат произведения 2 и 3. Группирующие скобки в AST не используются, так как в них нет необходимости. Сам граф определяет вложенность операций.



Во время формирования дерева парсер проверяет грамматику языка: корректная ли последовательность составлена из «слов». Например, строка "{}" будет успешно разобрана лексером, однако является неверной, так как отсутствует вторая закрывающая скобка.

PARSER

```
[(base) MacBook-Air-Aroslava:hello Yaroslava$ swiftc -dump-parse main.swift
(source_file
  (import_decl 'Foundation')
  (top_level_code_decl
    (brace_stmt
      (pattern_binding_decl
        (pattern_named 'name')
        (string_literal_expr type='<null>' encoding=utf8 value="LOL" builtin_initializer=**NULL** initializer=**NULL**)))
    ))
  (var_decl "name" type='<null type>' storage_kind=stored)
  (top_level_code_decl
    (brace_stmt
      (pattern_binding_decl
        (pattern_typed
          (pattern_named 'age')
          (type_ident
            (component id='Int' bind=none))))
        (integer_literal_expr type='<null>' value=32)
        (pattern_typed
          (pattern_named 'kek')
          (type_ident
            (component id='Int' bind=none))))
        (integer_literal_expr type='<null>' value=43))
      ))
    (var_decl "age" type='<null type>' storage_kind=stored)
    (var_decl "kek" type='<null type>' storage_kind=stored)
    (top_level_code_decl
      (brace_stmt
        (call_expr type='<null>' arg_labels=_:
          (unresolved_decl_ref_expr type='<null>' name=print function_ref=unapplied)
          (paren_expr type='<null>'
            (string_literal_expr type='<null>' encoding=utf8 value="Hello, World!" builtin_initializer=**NULL** initializer=**NULL**))))))
      ))
    ))
  (base) MacBook-Air-Aroslava:hello Yaroslava$
```

Пример парсинга маленькой программы, где объявлены три переменные, одна из которых строковая и вывод текста «Hello, World!»

SEMA И РАЗНИЦА МЕЖДУ PARSER И SEMA

Если мы попытаемся присвоить переменной типа Bool число 16, и передадим в парсер, тот не обнаружит никаких отклонений.

```
(source_file
  (import_decl 'Foundation')
  (top_level_code_decl
    (brace_stmt
      (pattern_binding_decl
        (pattern_typed
          (pattern_named 'x')
          (type_ident
            (component id='Bool' bind=None)))
        (integer_literal_expr type='<null>' value=16)))
    ))
  (var_decl "x" type='<null type>' let
    storage_kind=stored))
```

```
import Foundation
```

```
let x: Bool = 16
```

А вот семантический анализатор укажет, что не так с переданным ему кодом:

```
main.swift:11:15: error: cannot convert value of
type 'Int' to specified type 'Bool'
let x: Bool = 16
```

SEMA - проходит по всему выражению и присваивает типы выражениям, проверяет поддержку протоколов, синтезирует для структур анализаторы по умолчанию и др.

SILGEN

SIL имеет SSA форму. Static Single Assignment (SSA) – представление кода, в котором каждой переменной значение присваивается только один раз. Оно создаётся из обычного кода добавлением дополнительных переменных.

a = 1	a1 = 1
a = 2	a2 = 2
b = a	b1 = a2

SIL позволяет применять к коду Swift специфичные оптимизации и проверки, которые было бы сложно или невозможно осуществить на этапе AST.

```
%6 = integer_literal $Builtin.Int2048, 1          // user: %8
// function_ref Int.init(_builtinIntegerLiteral:)
%7 = function_ref @$Si22_builtinIntegerLiteralSiBi2048__tcfC : @$convention(method) (Builtin.Int2048, @thin Int.Type)
-> Int // user: %8
%8 = apply %7(%6, %5) : @$convention(method) (Builtin.Int2048, @thin Int.Type) -> Int // user: %10
```

SILGEN

- ▶ `sil` - объявление функции;
- ▶ `@, calling convention` - параметры, тип возвращаемого значения, код функции;
- ▶ `$` - начало указания типа;
- ▶ `S4main3AAAV3fffSiyF` - для одинаковых имен в разных модулях (4 - число символов в модуле, 3 - число символов в названии класса, Si - Swift.int);

```
struct AAA {  
    func fff() -> Int {  
        return 8  
    }  
}
```

```
struct BBB {  
    func fff() -> Int {  
        return 8  
    }  
}
```

```
sil hidden @$S4main3BBBV3fffSiyF : @$convention(method) (BBB) -> Int {
```

SILGEN

```
struct AAA {  
    func fff(iii internalName: Int) -> Int {  
        return 8  
    }  
  
    func fff(ddd internalName: Double) -> Int {  
        return 8  
    }  
}
```

Для данного когда в названии добавляются имена и типы аргументов функций. Это позволяет использовать перегрузку.

Для первой функции:

```
sil hidden @$S4main3AAAV3fff3iiiS2i_tF : @$convention(method) (Int, AAA) -> Int {
```

Для второй функции:

```
sil hidden @$S4main3AAAV3fff3dddSiSd_tF : @$convention(method) (Double, AAA) -> Int {
```

SILGEN

Базовый блок - последовательность инструкций с одной точкой входа, одной точкой выхода, которая не содержит инструкций ветвления или условий для раннего выхода.

```
bb0(%0 : $Int32, %1 : $UnsafeMutablePointer<Optional<UnsafeMutablePointer<Int8>>>):
```

```
//before
if 2 > 5 {
    //true
} else {
    //false
}
//after
```

4 блока:

- КОД ДО ВЕТВЛЕНИЯ;
- случай, когда выражение верно;
- случай, когда выражение ложно;
- КОД ПОСЛЕ ВЕТВЛЕНИЯ;

cond_br - инструкция для условного перехода

```
cond_br %14, bb1, bb2 // id: %15
```

```
bb1: // Preds: bb0
br bb3 // id: %16
```

```
bb2: // Preds: bb0
br bb3 // id: %17
```

```
bb3: // Preds: bb2 bb1
%18 = integer_literal $Builtin.Int32, 0 // user: %19
%19 = struct $Int32 (%18 : $Builtin.Int32) // user: %20
return %19 : $Int32
```

ГЕНЕРАЦИЯ КАНОНИЧНОГО SIL

```
let x = 16 + 8
```

В его сыром SIL можно найти сложение этих литералов:

```
%13 = function_ref @$Ssi1poiys2i_SitFZ : $@convention(method) (Int, Int, @thin Int.Type) -> Int //  
user: %14  
%14 = apply %13(%8, %12, %4) : $@convention(method) (Int, Int, @thin Int.Type) -> Int // user: %15
```

В каноничном его уже нет. Вместо этого используется константное значение 24:

```
%4 = integer_literal $Builtin.Int64, 24 // user: %5
```

Далее генератор LLVM IR преобразует SIL в промежуточное представление LLVM. Оно передаётся в бекенд для дальнейшей оптимизации и генерации машинного кода.

ПРИМЕР ГЕНЕРАЦИИ АССЕМБЛЕРНОГО КОДА

```
.section    __TEXT,__text,regular,pure_instructions
.macosx_version_min 10, 13
.globl _main
.p2align    4, 0x90
_main:
.cfi_startproc
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq    %rsp, %rbp
.cfi_def_cfa_register %rbp
movq    $16, _$S4main1xSivp(%rip)
movq    _$S4main1xSivp(%rip), %rax
addq    $7, %rax
seto    %cl
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movq    %rax, -24(%rbp)
movb    %cl, -25(%rbp)
jo      LBB0_2
xorl    %eax, %eax
movq    -24(%rbp), %rcx
movq    %rcx, _$S4main1ySivp(%rip)
popq    %rbp
retq
```

```
swiftc -emit-assembly main.swift
```

ЗАКЛЮЧЕНИЕ

- ▶ Swift – хорошо структурированный компилятор, и разобраться в его общей архитектуре оказалось не сложно.
- ▶ Swift отлично подходит под написание приложений и игр под операционные системы Apple. И гораздо перспективнее, чем Objective-C.

ЛИТЕРАТУРА

- ▶ <https://habr.com/ru/company/e-Legion/blog/440078/>
- ▶ <https://habr.com/ru/company/e-Legion/blog/438696/>
- ▶ <https://github.com/fullstackio/FlappySwift>