SSA-форма в LLVM IR

Лекция №9

Теория языков программирования

Вопросы лекции

- Где используется SSA-форма?
- Зачем она нужна?
- Как она влияет на LLVM IR?
- Каковы границы возможностей LLVM IR?

Где используется SSA

Single Static Assignment

SSA-форма — это форма IR, в которой есть только одно присваивание каждого значения

Single Static Assignment

SSA-форма — это форма IR, в которой есть только одно присваивание каждого значения

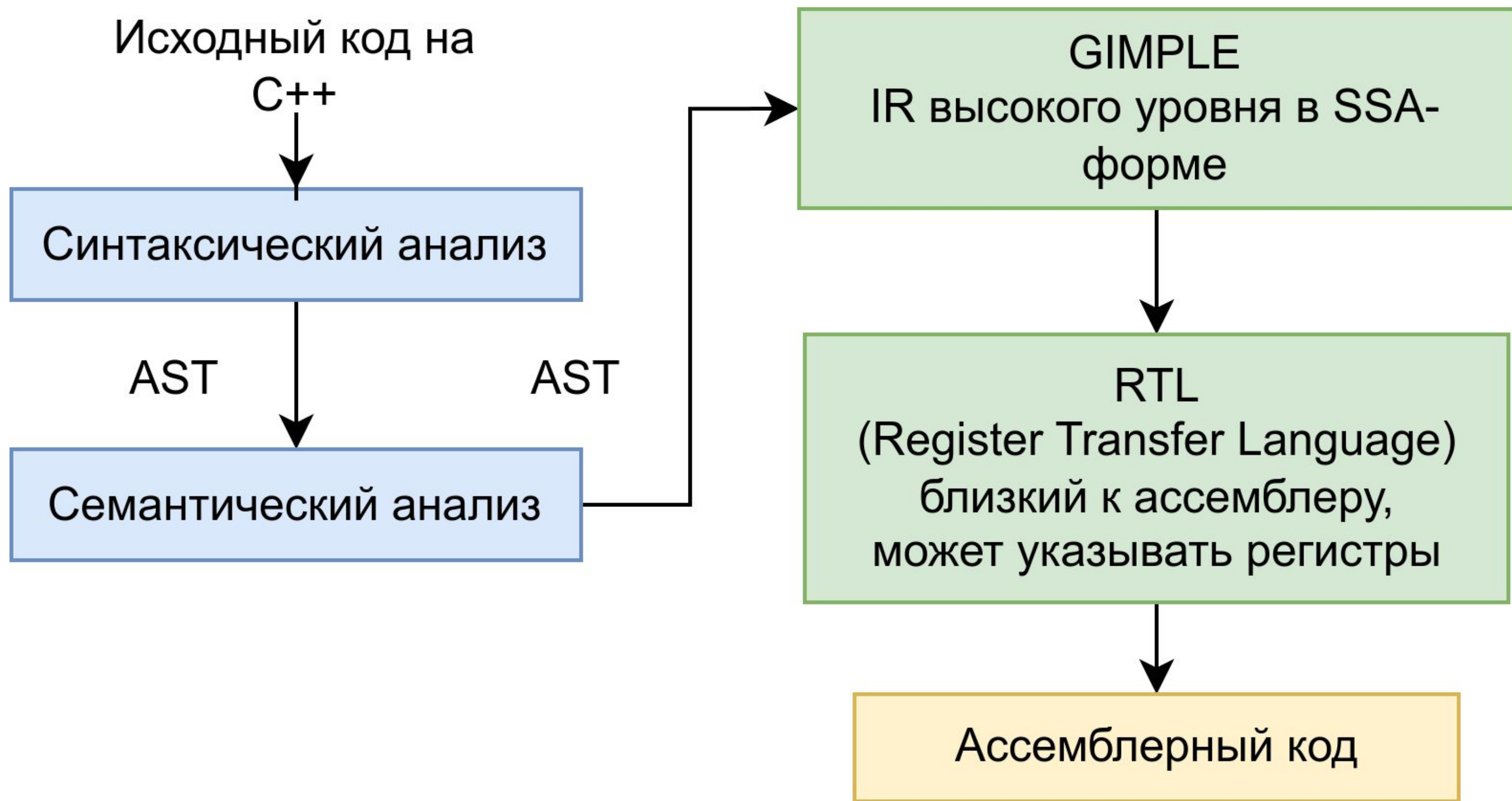
```
; Нельзя присвоить %1 дважды:
%1 = add i32 %arg.x, %arg.y
%1 = imul i32 %1, %1

; Можно создать новое значение %2:
%1 = add i32 %arg.x, %arg.y
%2 = imul i32 %1, %1
```

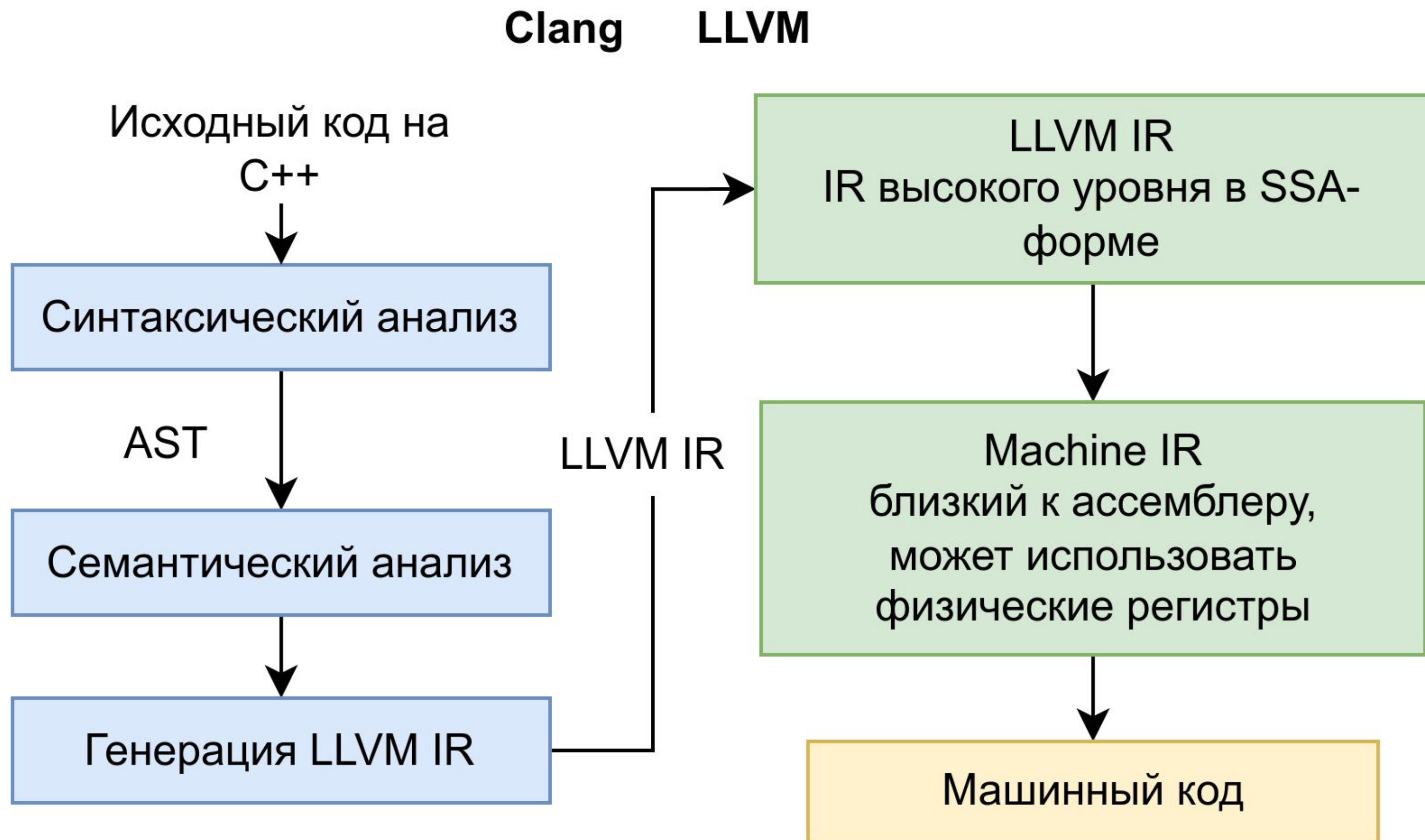
Где используется SSA?

1. Компиляторы на LLVM — Clang, Swift, Rust и т.д.
2. Компиляторы GCC
3. Компилятор Go
4. JIT-компилятор .NET — RuyJit

Архитектура GCC

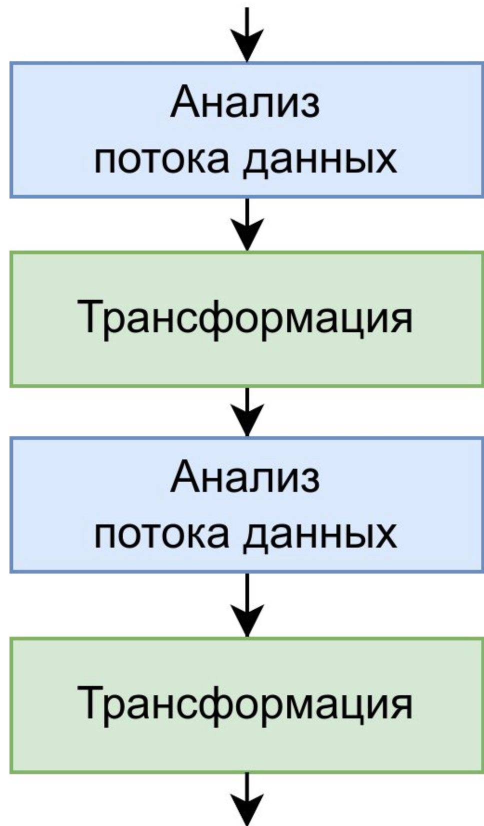


Архитектура Clang



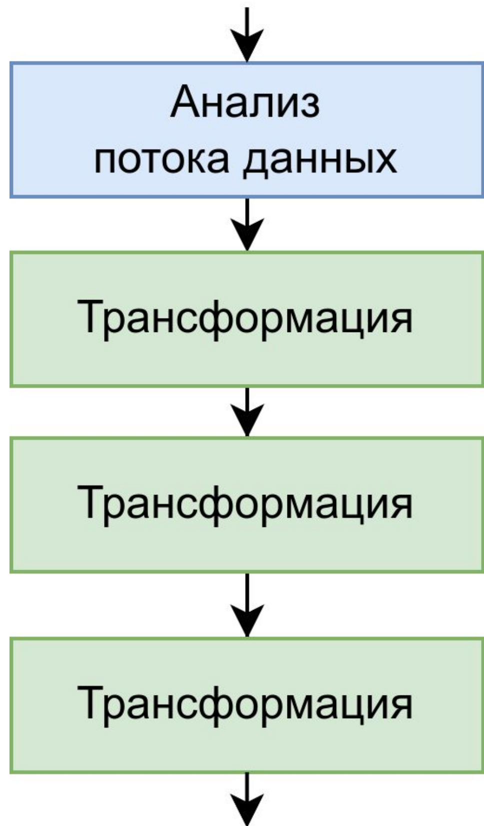
Зачем нужен SSA

Оптимизации компилятора без SSA-формы



1. Трансформация аннулирует анализ
   - Из-за этого надо повторять анализ
2. Межпроцедурные оптимизации слишком дорогие

Оптимизации компилятора с SSA-формой



1. Трансформация не аннулирует анализ
2. Межпроцедурные оптимизации приемлемы
3. Ряд алгоритмов проще и эффективнее

LLVM IR

Три формы LLVM IR

1. Структуры в памяти
2. Текстовый файл
3. Бинарный файл

Три формы LLVM IR

1. Структуры в памяти
2. Текстовый файл
3. Бинарный файл

```
define i32 @main() {
entry:
 %var.curr = alloca i32, align 4
 %var.next = alloca i32, align 4
 %var.i = alloca i32, align 4
 %var.tmp = alloca i32, align 4
 store i32 0, ptr %var.curr, align 4
 store i32 1, ptr %var.next, align 4
 store i32 0, ptr %var.i, align 4
 br label %loop
```

Особенности LLVM IR

1. Трёхадресный код
2. SSA-форма
3. RISC

Особенности LLVM IR

1. Трёхадресный код
2. SSA-форма
3. RISC

`call` принимает любое число аргументов

остальные — до 2 операндов и 1 результат:

`%3 = add %1, %2`

Особенности LLVM IR

1. Трёхадресный код
2. SSA-форма
3. RISC

SSA-форма для виртуальных регистров:

```
%3 = add %1, %2
```

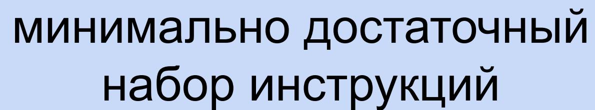
Памяти это не касается:

```
%4 = alloca i32, align 4
store i32 100, %4
store i32 200, %4
```

Есть оптимизация mem2reg

Особенности LLVM IR

1. Трёхадресный код
2. SSA-форма
3. RISC



Базовые блоки

1. Функция состоит из BasicBlock
    - все ветвления и циклы — это переходы между BasicBlock

Базовые блоки

1. Функция состоит из BasicBlock
   - все ветвления и циклы — это переходы между BasicBlock
2. BasicBlock состоит из инструкций
   - все **φ-узлы** должны быть в начале блока
   - завершается терминальной инструкцией — Br, CondBr, Ret, …
   - ровно одна терминальная инструкция

Переходы и возвраты

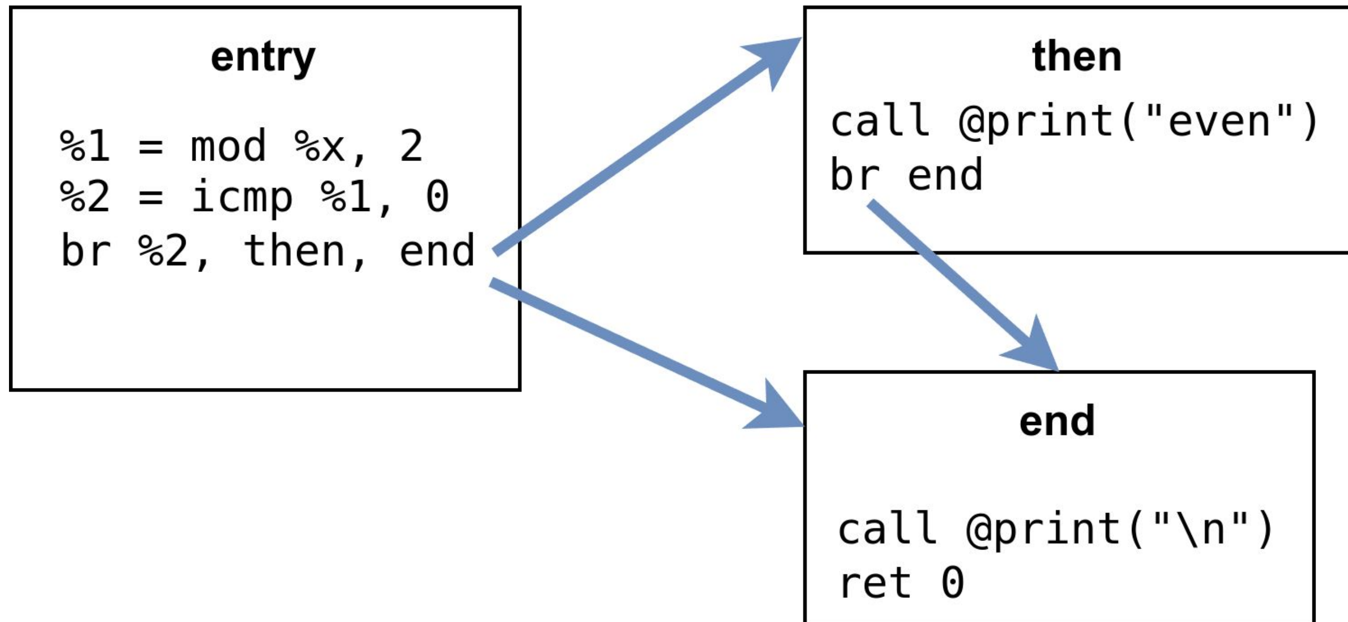
1. Безусловный переход
    - **`br`** `label %merge_then_else`
2. Условный переход
    - **`br`** `i1 %if_cond, label %then, label %else`
3. Возврат из функции
    - **`ret`** `i32 0`
    - **`ret`** `ptr %then_else_value`

Трансляция ветвления в переходы

```
if x % 2 = 0
then
    print("even")
print("\n")
```

Трансляция ветвления в переходы

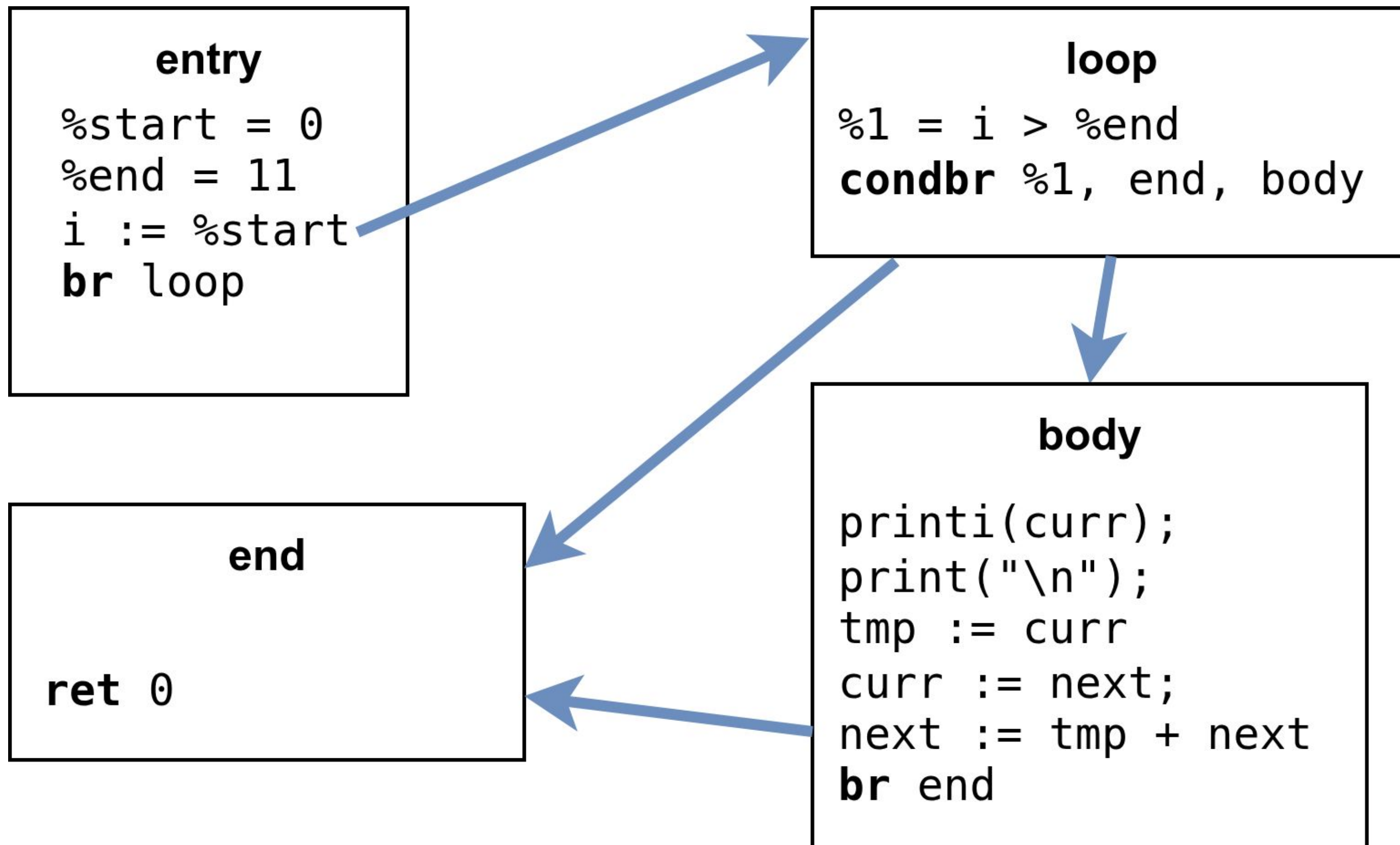
```
if x % 2 = 0
then
    print("even")
print("\n")
```



Трансляция цикла for в переходы

```
for i := 0 to 11 do (
    printi(curr);
    print("\n");
    tmp := curr
    curr := next;
    next := tmp + next
)
```

Трансляция цикла for в переходы

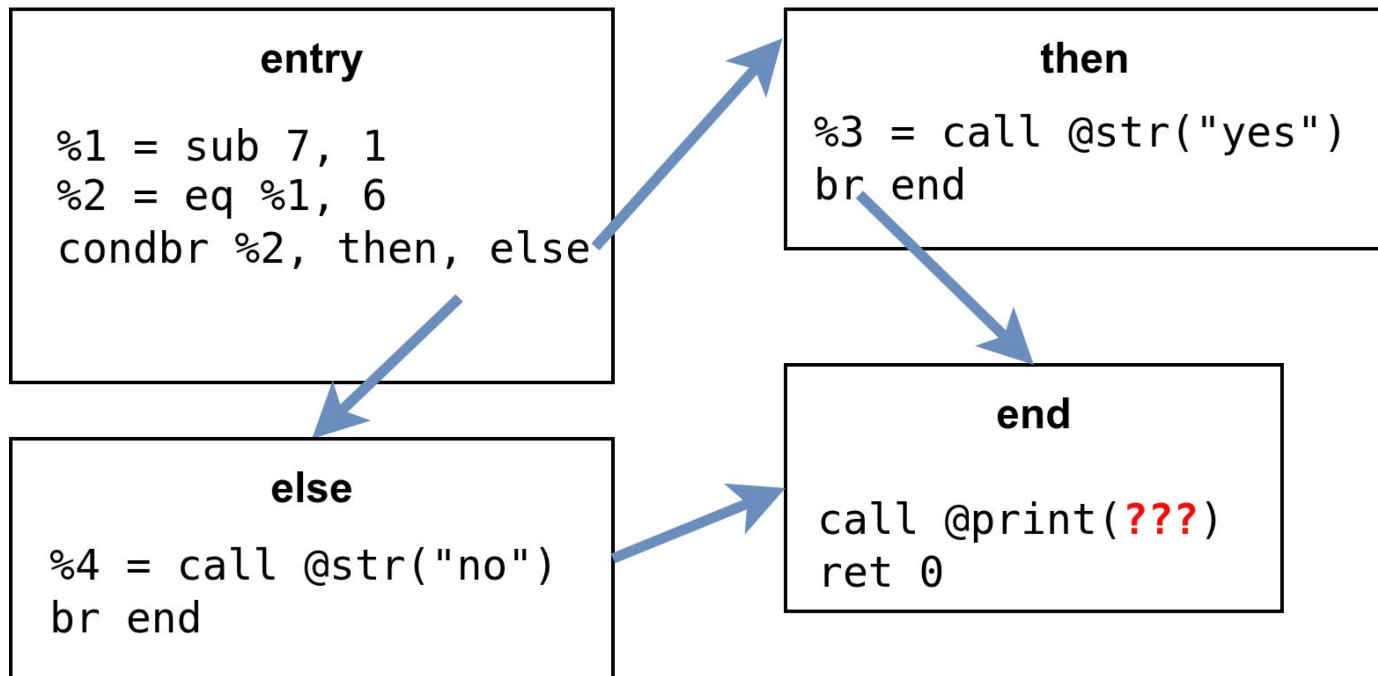


Тернарная операция if-then-else

```
print(
    if 7 - 1 = 6
    then
        "yes"
    else
        "no"
)
```

Тернарная операция if-then-else

```
print(
    if 7 - 1 = 6
    then
        "yes"
    else
        "no"
)
```



Инструкция φ (phi)

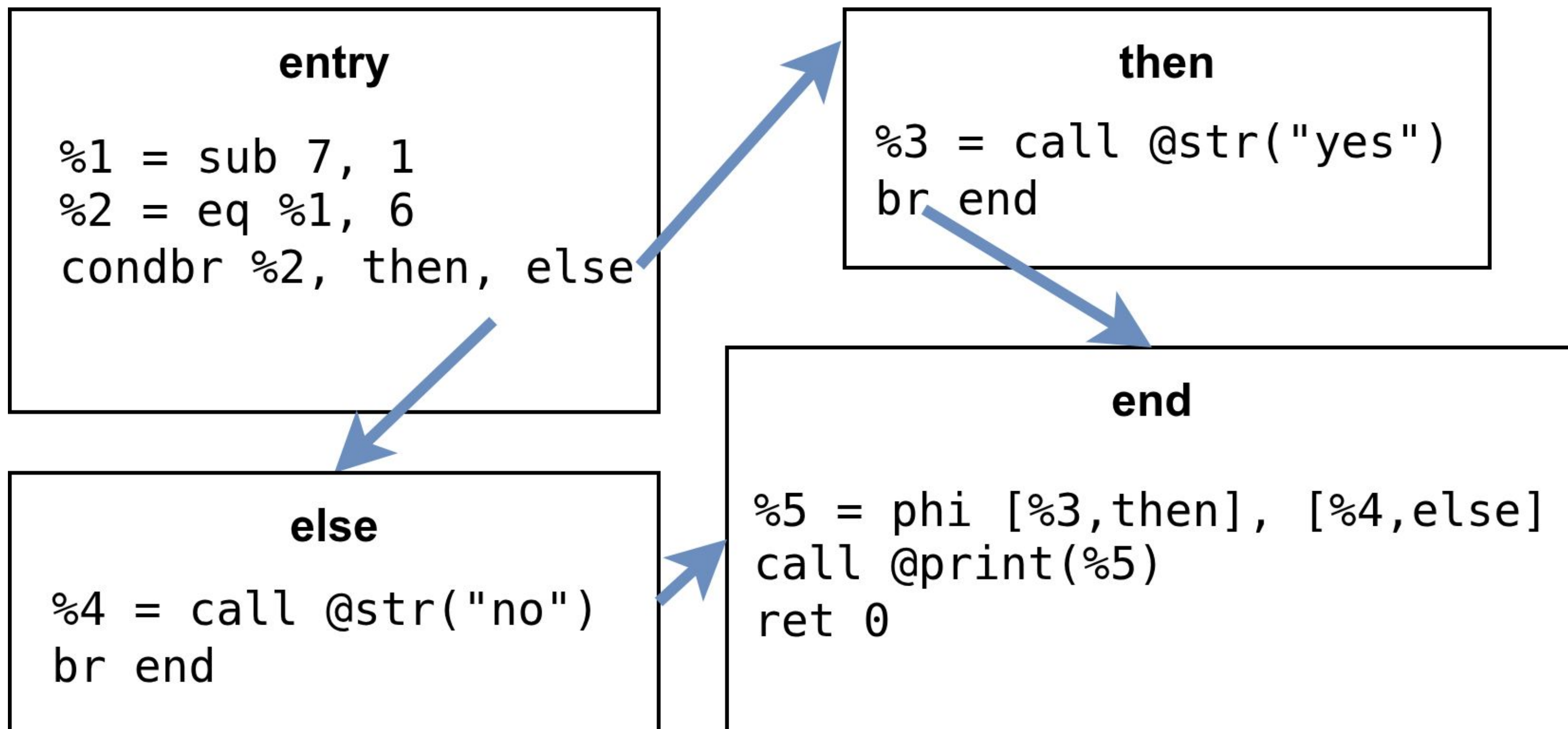
1. φ выбирает значение в зависимости от *предыдущего* блока
2. все φ должны быть в начале блока

Инструкция φ (phi)

1. φ выбирает значение в зависимости от *предыдущего* блока
2. все φ должны быть в начале блока

```
merge_then_else: ; preds = %else, %then
 %5 = phi ptr [ %3,%then ], [ %4,%else ]
 call void @tiger_print(ptr %5)
 call void @tiger_remove_reference(ptr %5)
 ret i32 0
```

Инструкция φ (phi)



Одного LLVM IR недостаточно

Одного LLVM IR недостаточно

В LLVM IR нет информации об особенностях языка

Одного LLVM IR недостаточно

В LLVM IR нет информации об особенностях языка

1. Лишнее создание, копирование, удаление объектов
   - в т.ч. строки, массивы, словари
2. Лишние изменения счётчика ссылок
   - для сильных и слабых ссылок

Пример потери информации в LLVM IR

```
print("\n");
print("isOdd(11) = ");
print(isOdd(11))
```

Пример потери информации в LLVM IR

```
print("\n");
print("isOdd(11) = ");
print(isOdd(11))
```

```
 %str.1 = call ptr @tiger_create_string(ptr @str.3)
call void @tiger_print(ptr %str.1)
%str.2 = call ptr @tiger_create_string(ptr @str.4)
call void @tiger_print(ptr %str.2)
%.result3 = call ptr @isOdd(i32 11)
call void @tiger_print(ptr %.result3)
call void @tiger_remove_reference(ptr %str.2)
call void @tiger_remove_reference(ptr %.result3)
call void @tiger_remove_reference(ptr %str.)
```

Пример потери информации в LLVM IR

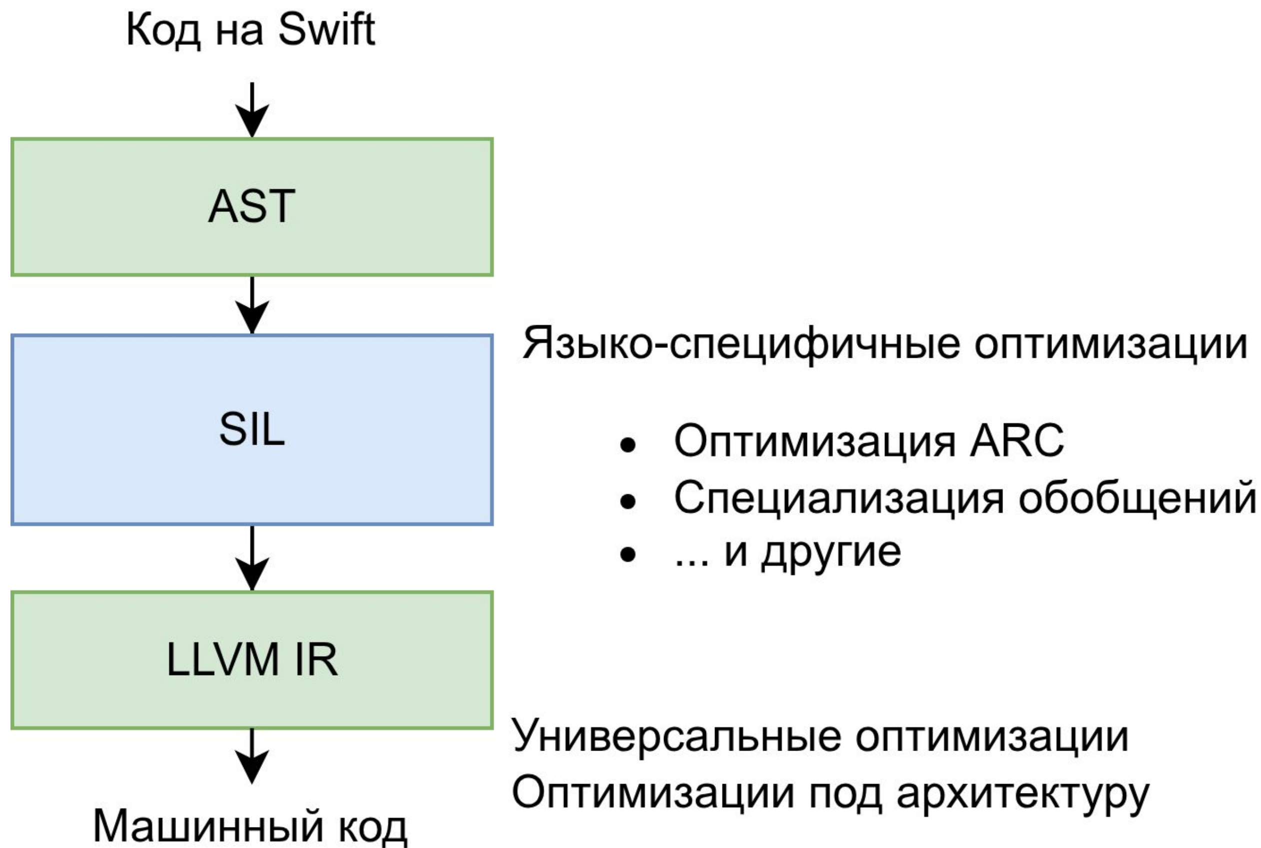
```
print("\n");
print("isOdd(11) = ");
print(isOdd(11))
```

LLVM не “знает”, что делает tiger_create_string

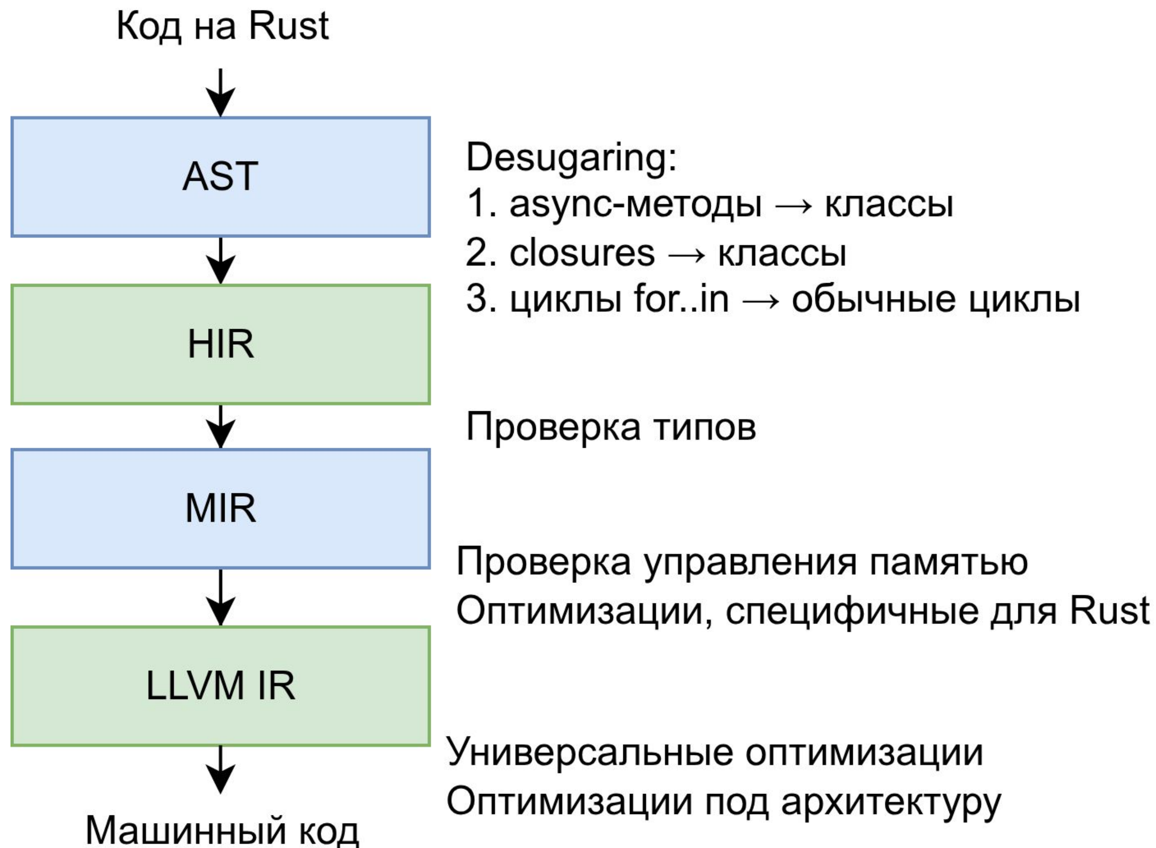
```
 %str.1 = call ptr @tiger_create_string(ptr @str.3)
call void @tiger_print(ptr %str.1)
%str.2 = call ptr @tiger_create_string(ptr @str.4)
call void @tiger_print(ptr %str.2)
%.result3 = call ptr @isOdd(i32 11)
call void @tiger_print(ptr %.result3)
call void @tiger_remove_reference(ptr %str.2)
call void @tiger_remove_reference(ptr %.result3)
call void @tiger_remove_reference(ptr %str.)
```

Уровни IR

Уровни IR в Swift



Уровни IR в Rust



Уровни IR в LLVM

AST или другой IR

Код на LLVM IR

Machine IR

Машинный код

Универсальные оптимизации:

1. Встраивание функций (inlining)
2. Устрание общих подвыражений
3. Вынесение инвариантов циклов
4. Превращение переменных в регистров

1. Выбор инструкций
2. Планирование инструкций
3. Распределение физических регистров

Уровни IR

1. High-level IR
    - Rust HIR, Rust MIR
    - Swift SIR
    - Clang CIR

Уровни IR

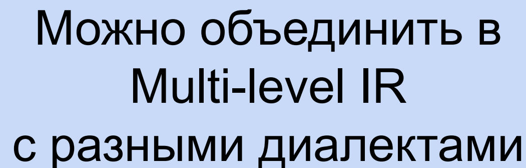
1. High-level IR
   - Rust HIR, Rust MIR
   - Swift SIR
   - Clang CIR
2. Middle-level IR
   - LLVM IR в LLVM
   - GIMPLE в GCC

Уровни IR

1. High-level IR
   - Rust HIR, Rust MIR
   - Swift SIR
   - Clang CIR
2. Middle-level IR
   - LLVM IR в LLVM
   - GIMPLE в GCC
3. Machine-level IR
   - MachineIR в LLVM
   - RTL в GCC

Уровни IR

1. High-level IR
   - Rust HIR, Rust MIR
   - Swift SIR
   - Clang CIR
2. Middle-level IR
   - LLVM IR в LLVM
   - GIMPLE в GCC
3. Machine-level IR
   - MachineIR в LLVM
   - RTL в GCC



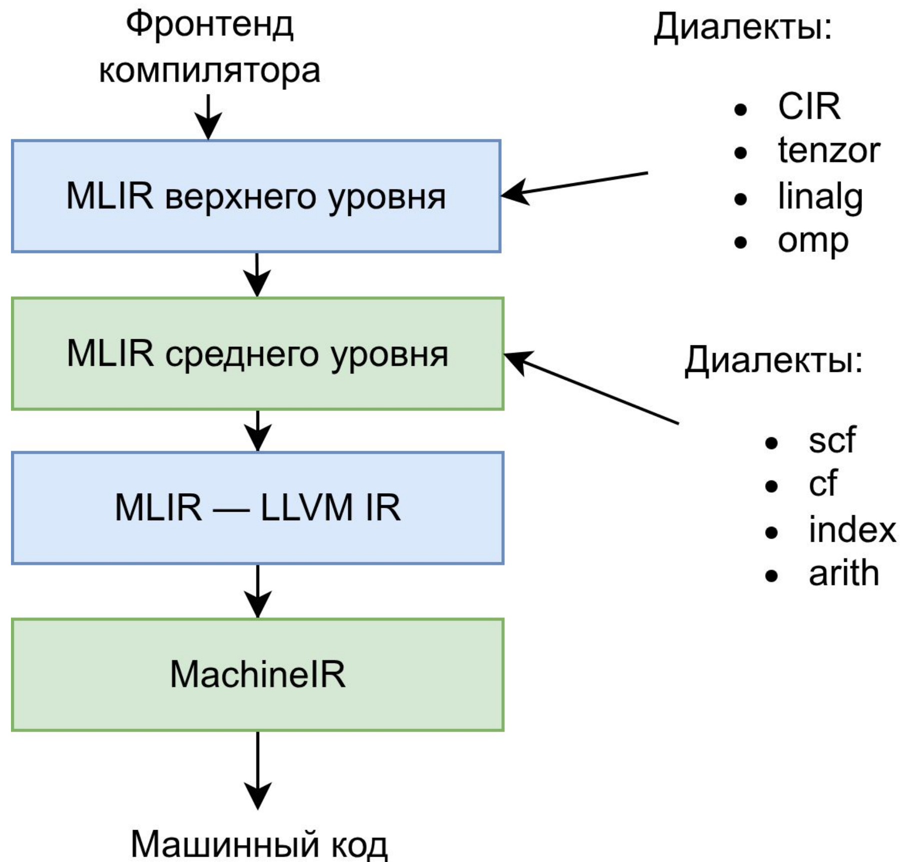
MLIR в LLVM

Что такое MLIR

MLIR — Multi-Level Intermediate Representation

- Один IR, много уровней представления
- Каждый уровень — один диалект

Трансформация диалектов MLIR



Диалекты:

- CIR
- tenzor
- linalg
- omp

Диалекты:

- scf
- cf
- index
- arith

MLIR — диалект scf

Диалект с поддержкой структурного программирования

1. Ветвления
2. Циклы while, for

MLIR — диалект scf

Диалект с поддержкой структурного программирования

1. Ветвления
2. Циклы while, for

```
scf.if %b  {
  // true region
} else {
  // false region
}
```

MLIR — диалект scf

Диалект с поддержкой структурного программирования

1. Ветвления
2. Циклы while, for

```
scf.if %b  {

  // true region

} else {

  // false region

}
```

```
%lb = index.constant 0
%ub = index.constant 10
%step = index.constant 1

scf.for %iv = %lb to %ub step %step {
  // loop region
}
```

Подытожим

Single Static Assignment

SSA-форма — это форма IR, в которой есть только одно присваивание каждого значения

```
; Нельзя присвоить %1 дважды:
%1 = add i32 %arg.x, %arg.y
%1 = imul i32 %1, %1

; Можно создать новое значение %2:
%1 = add i32 %arg.x, %arg.y
%2 = imul i32 %1, %1
```

Ограничения базовых блоков

1. Функция состоит из BasicBlock
   - все ветвления и циклы — это переходы между BasicBlock
2. BasicBlock состоит из инструкций
   - все **φ-узлы** должны быть в начале блока
   - завершается терминальной инструкцией — Br, CondBr, Ret, …
   - ровно одна терминальная инструкция

Одного LLVM IR недостаточно

В LLVM IR нет информации об особенностях языка

1. Лишнее создание, копирование, удаление объектов
   - в т.ч. строки, массивы, словари
2. Лишние изменения счётчика ссылок
   - для сильных и слабых ссылок

Одного LLVM IR недостаточно

В LLVM IR нет информации об особенностях языка

1. Лишнее создание, копирование, удаление объектов
    - в т.ч. строки, массивы, словари
2. Лишние изменения счётчика ссылок
    - для сильных и слабых ссылок

Решения:

- Свой High-Level IR
- Диалекты LLVM MLIR

Конец!