



LLVM Internals

МФТИ
Весна 2025

LLVM – повторение



- Front end

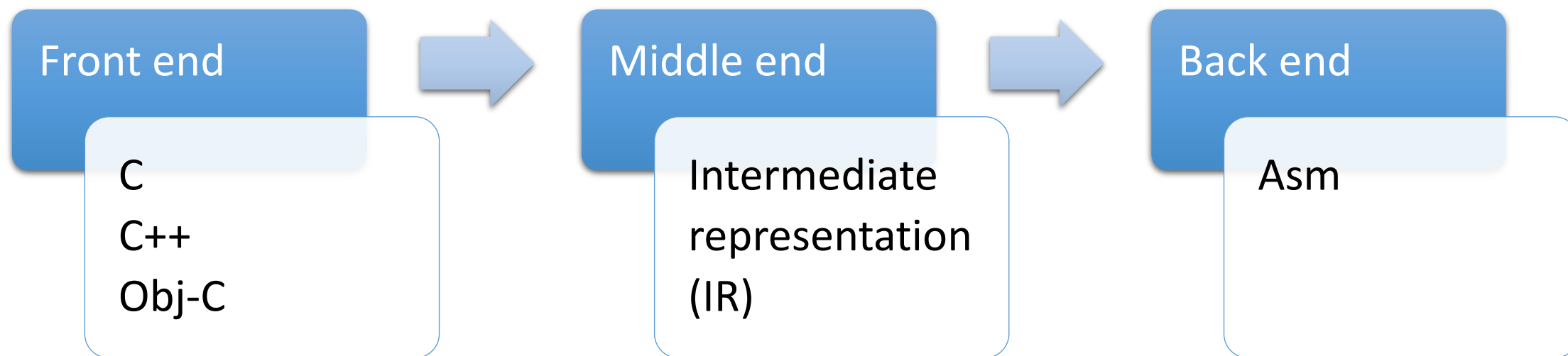
- Lexer
- Parser
- AST
- Semantic analyzer
- AST to IR

- Middle end

- IR
- IR analysis
- Optimizations over IR

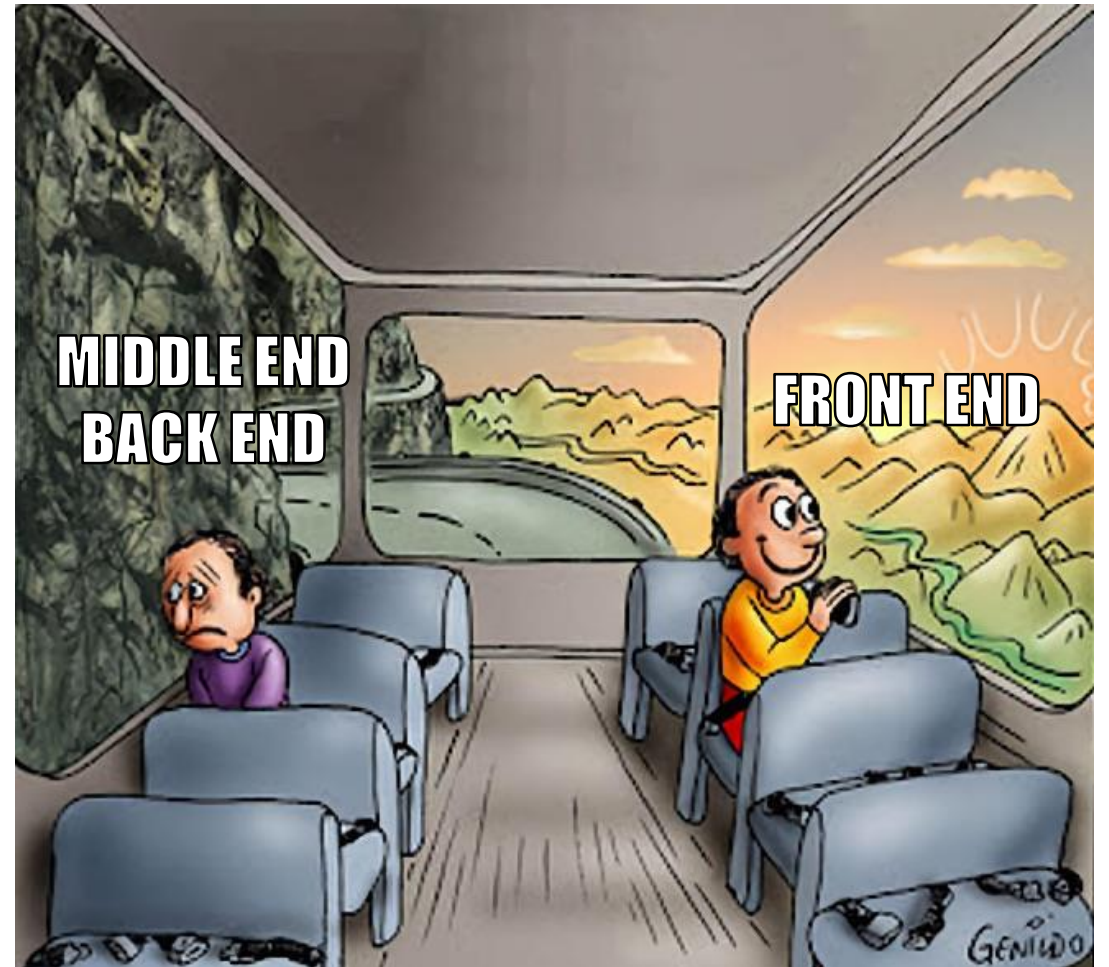
- Back end

- Codegen



Погружаемся в детали

- Как мы выяснили на прошлом занятии, для Middle end и Back end нужно спроектировать:
 - IR
 - Организованная работа с IR
 - Описание поддерживаемых целевых архитектур
 - Оптимальное преобразование IR в команды целевой архитектуры

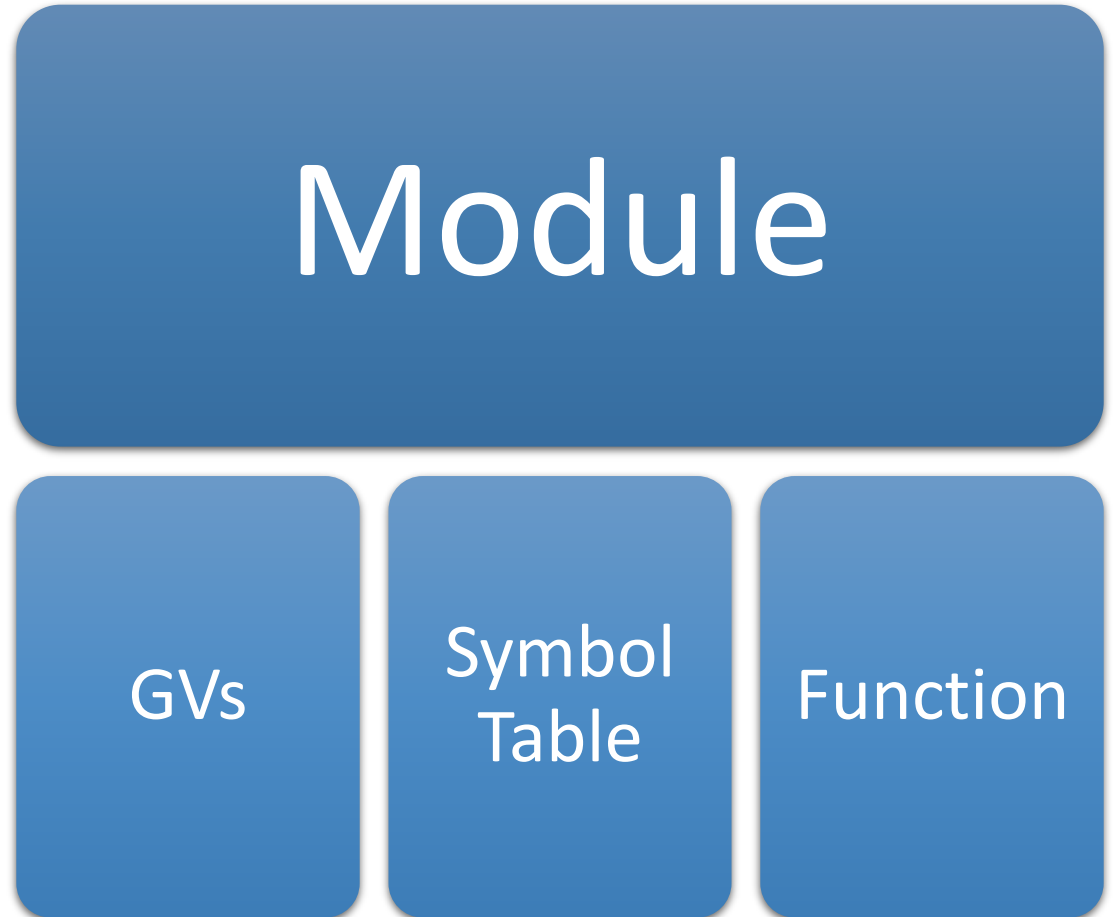


Погружаемся в детали: IR

- Рассмотрим требования:
 - Универсальный набор команд подходящий как для трансляции языков высокого уровня в него, так и для трансляции в ассемблер
 - Удобен для анализа
 - Удобен для трансформаций
 - Быстр для анализа и трансформаций

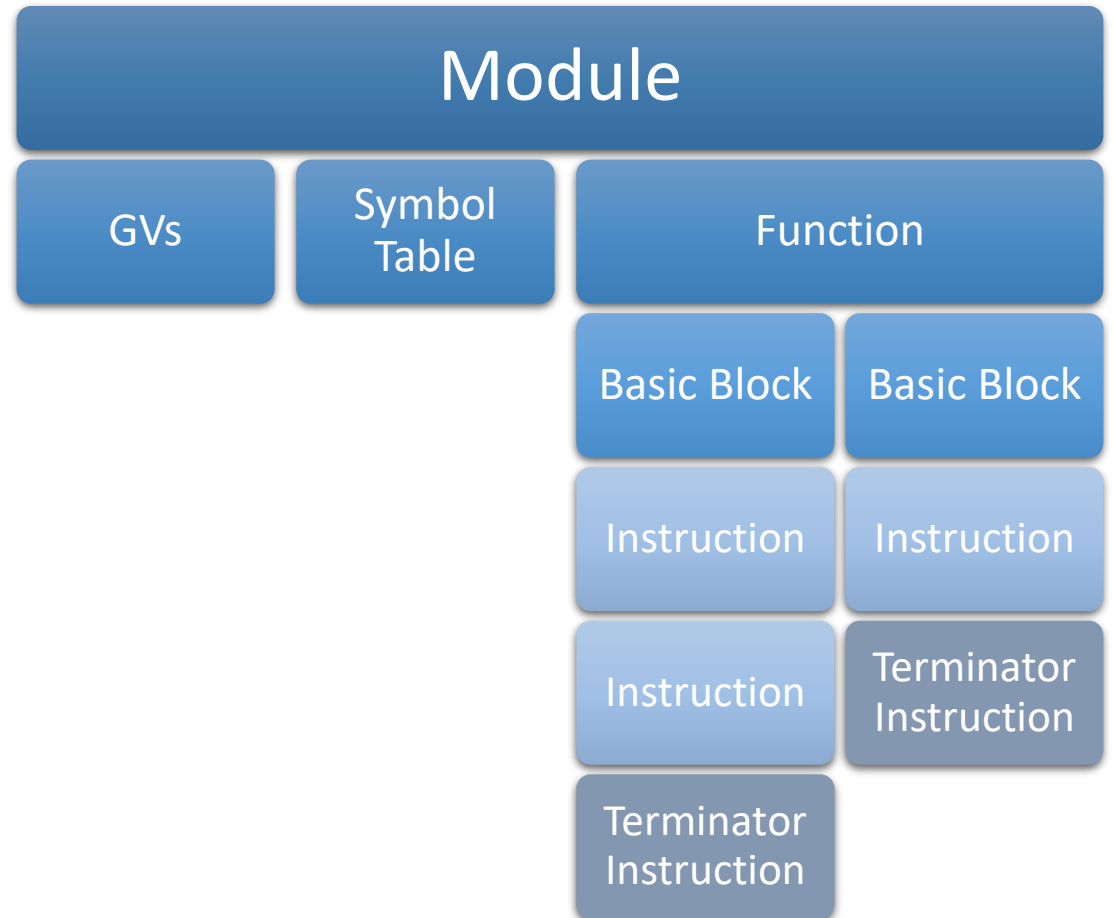
LLVM IR – общая структура

- В LLVM программы состоят из модулей, которые соответствуют единице трансляции исходного языка
- Модуль состоит из
 - Глобальных переменных (Global Variables – GV's)
 - Таблицы символов
 - Функций (объявлений и определений)



LLVM IR – общая структура

- Функция состоит из списка базовых блоков
- Базовые блоки состоят из списка инструкций
- Базовый блок обязан заканчиваться специальной инструкцией – терминатором – определяющей, какой базовый блок должен исполняться следующим



LLVM IR – пример

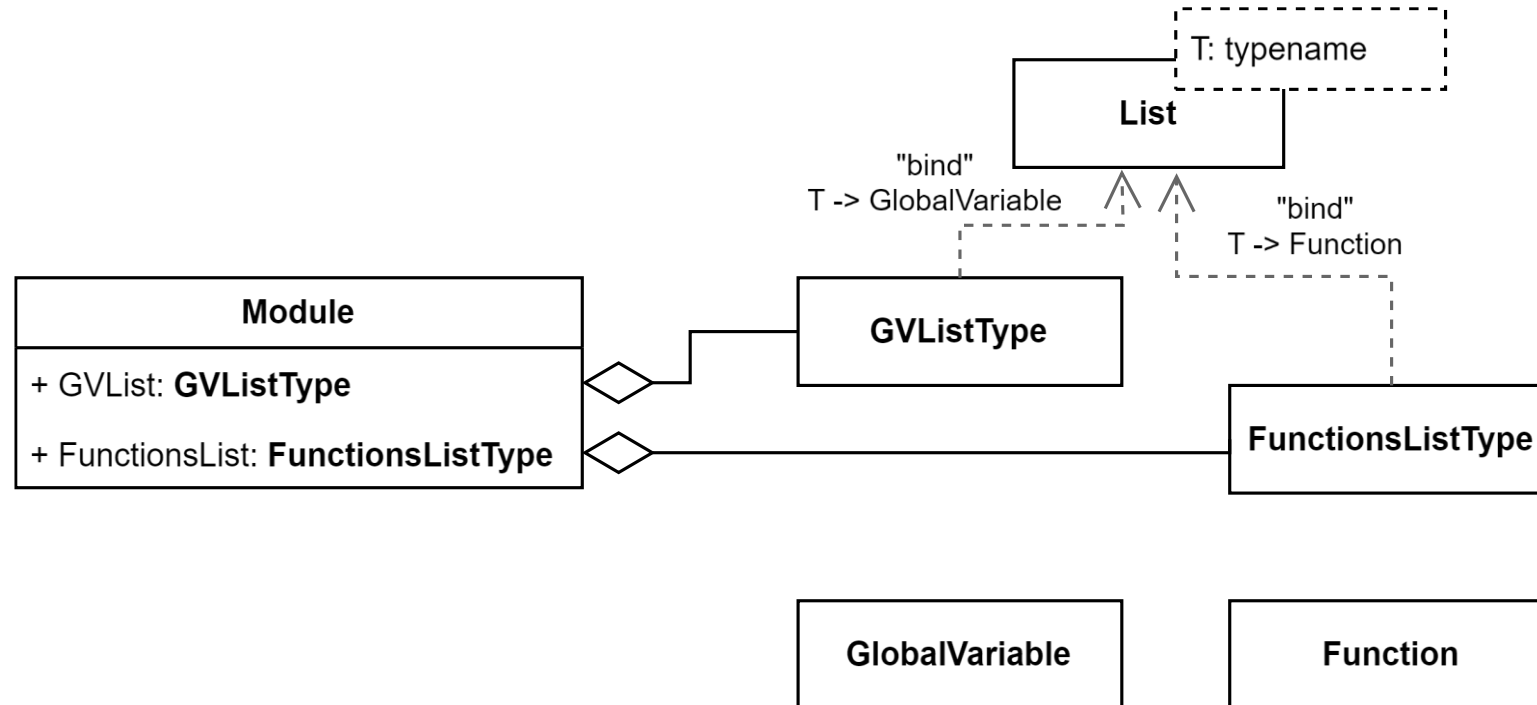
```
int mul(int x, int y)
{
    return x * y;
}
```

```
define i32 @mul(i32 %0, i32 %1):
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, ptr %3, align 4
    store i32 %1, ptr %4, align 4
    %5 = load i32, ptr %3, align 4
    %6 = load i32, ptr %4, align 4
    %7 = mul i32 %5, %6
    ret i32 %7
```

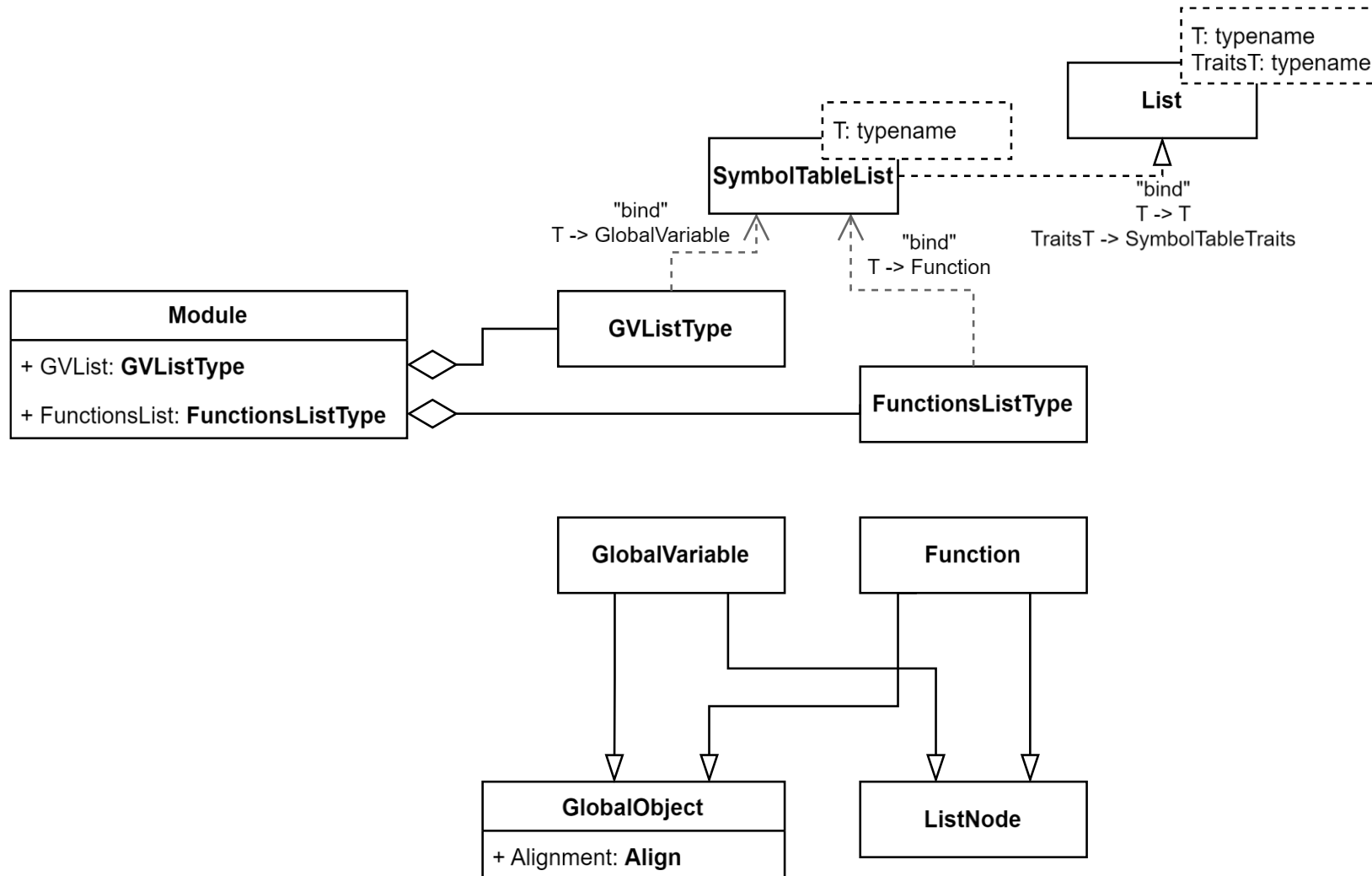
LLVM IR – проектируем объектную модель

| Module |
|-----------------------------------|
| + GVList: < <i>type?</i> > |
| + FunctionsList: < <i>type?</i> > |

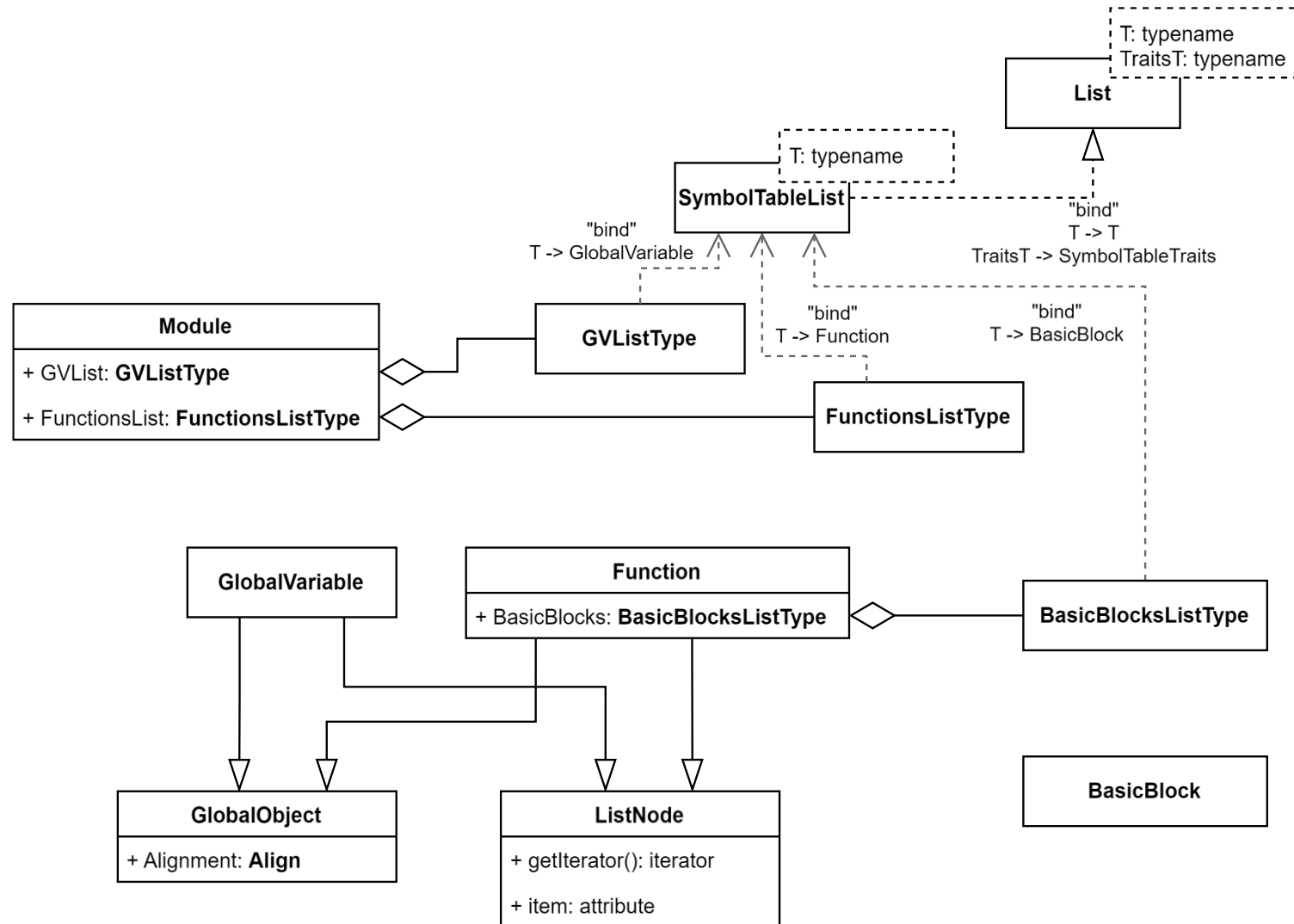
LLVM IR – проектируем объектную модель



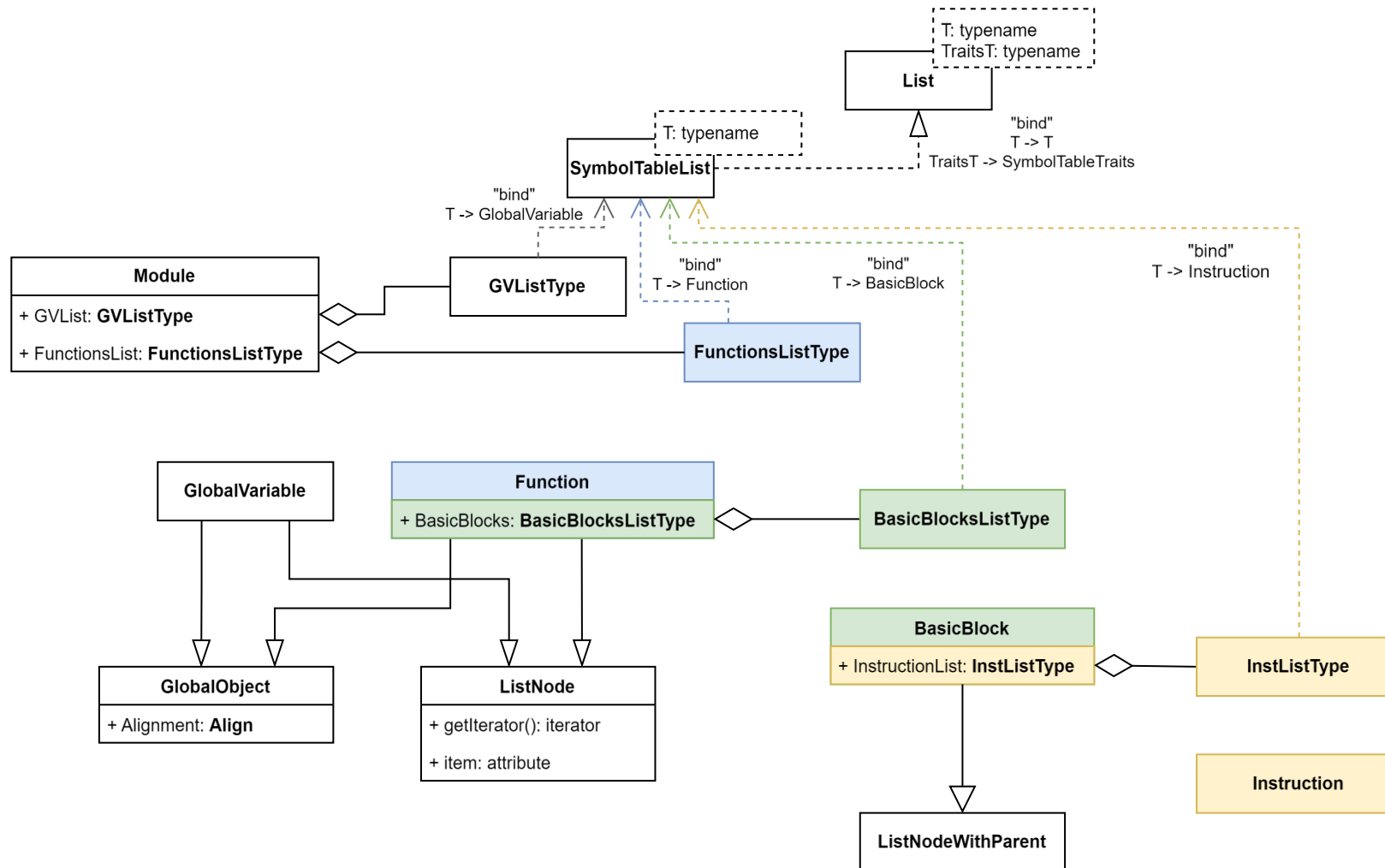
LLVM IR – проектируем объектную модель

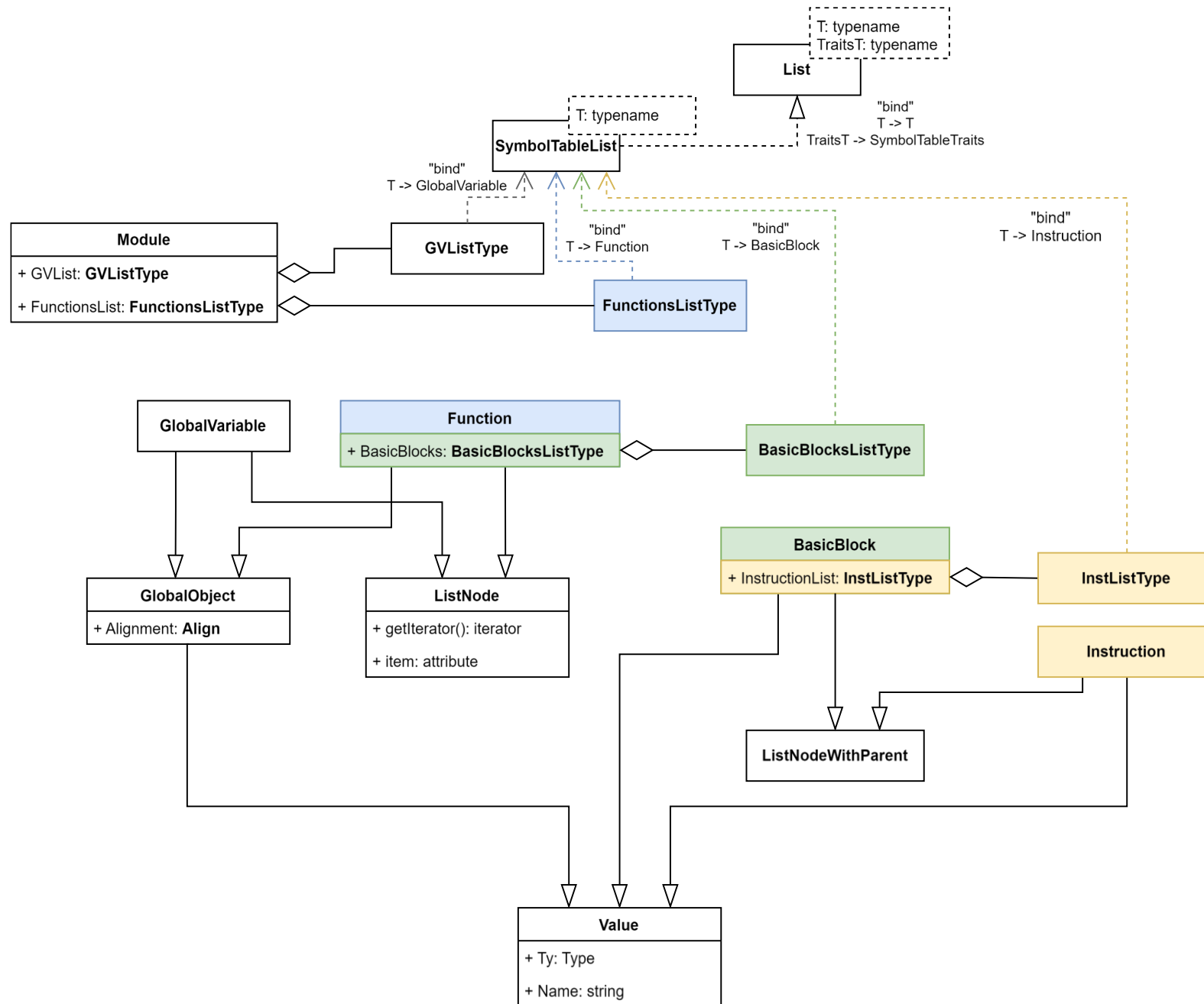


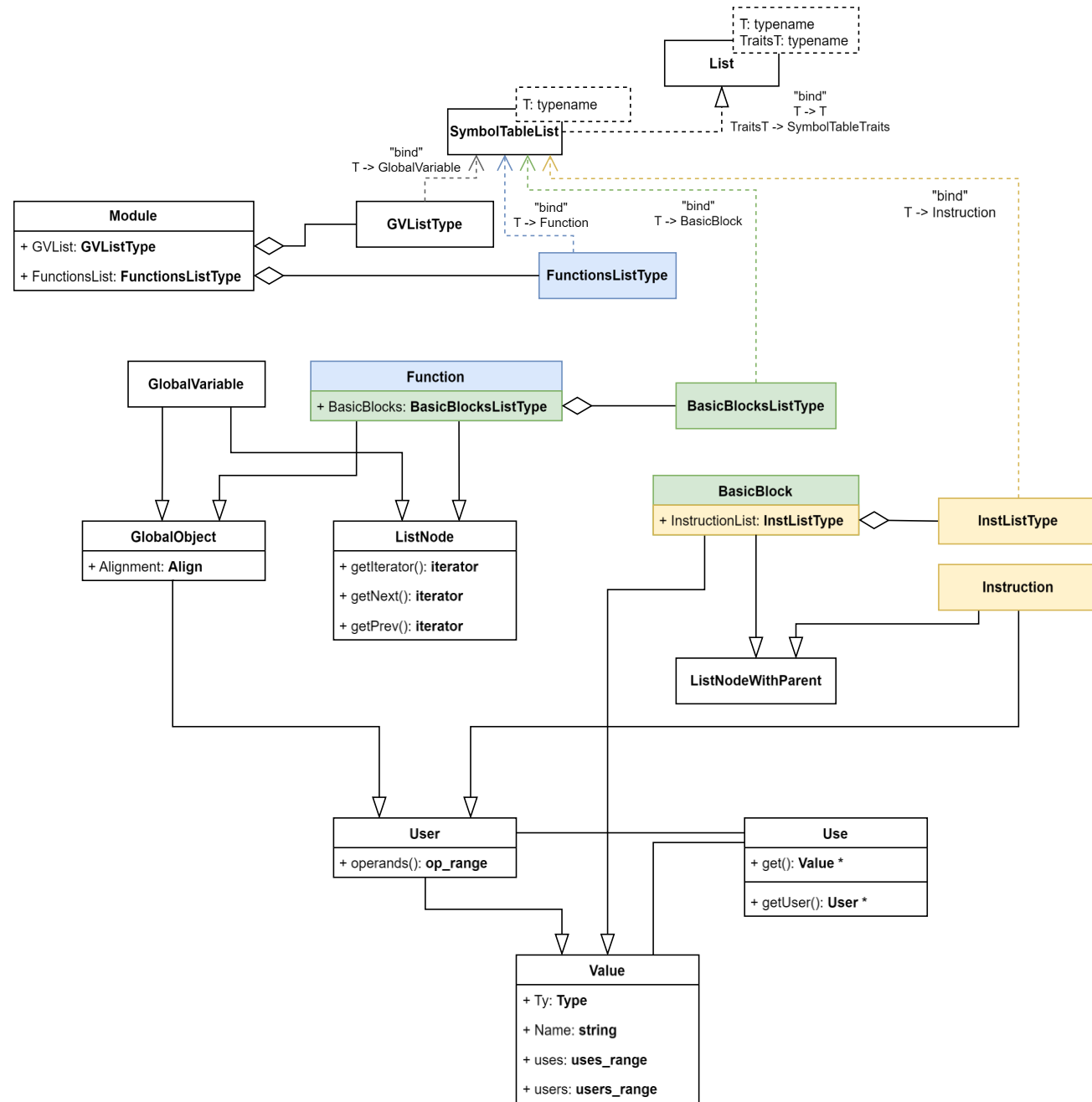
LLVM IR – проектируем объектную модель



LLVM IR – проектируем объектную модель







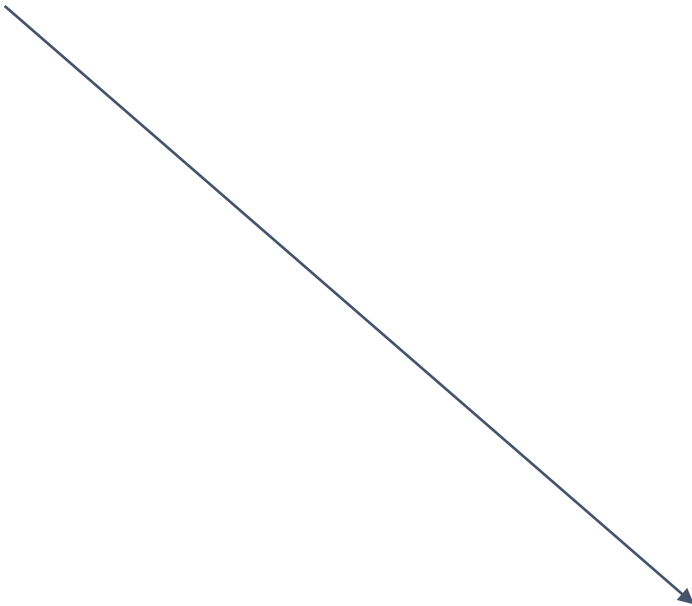
LLVM IR – набор команд

- Все инструкции в LLVM делятся на несколько категорий:
 - Терминаторы
 - Унарные операции
 - Бинарные операции
 - Побитовые операции
 - Векторные операции
 - Операции с агрегированными типами
 - Операции с памятью
 - Операции преобразований типов
 - Другие инструкции

LLVM IR Базовый пример

```
int mul(int x, int y) {  
    return x * y;  
}
```

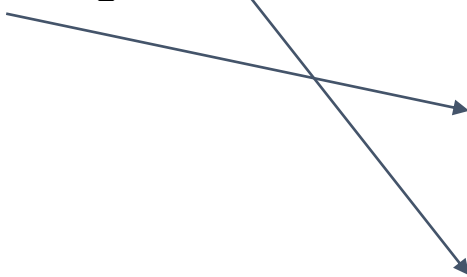
```
define i32 @mul(i32 %0, i32 %1):  
    %3 = alloca i32, align 4  
    %4 = alloca i32, align 4  
    store i32 %0, ptr %3, align 4  
    store i32 %1, ptr %4, align 4  
    %5 = load i32, ptr %3, align 4  
    %6 = load i32, ptr %4, align 4  
    %7 = mul i32 %5, %6  
    ret i32 %7
```




LLVM IR Пример с getelementptr (массив)

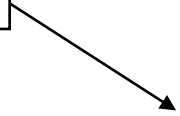
```
int mul(int *x,  
        int *y) {  
    return x[0] * y[0];  
}
```

```
define i32 @mul(ptr %0, ptr %1):  
    %3 = alloca ptr, align 8  
    %4 = alloca ptr, align 8  
    store ptr %0, ptr %3, align 8  
    store ptr %1, ptr %4, align 8  
    %5 = load ptr, ptr %3, align 8  
    %6 = getelementptr inbounds i32, ptr %5, i64 0  
    %7 = load i32, ptr %6, align 4  
    %8 = load ptr, ptr %4, align 8  
    %9 = getelementptr inbounds i32, ptr %8, i64 0  
    %10 = load i32, ptr %9, align 4  
    %11 = mul i32 %7, %10  
    ret i32 %11
```



LLVM IR Пример с getelementptr (структура)

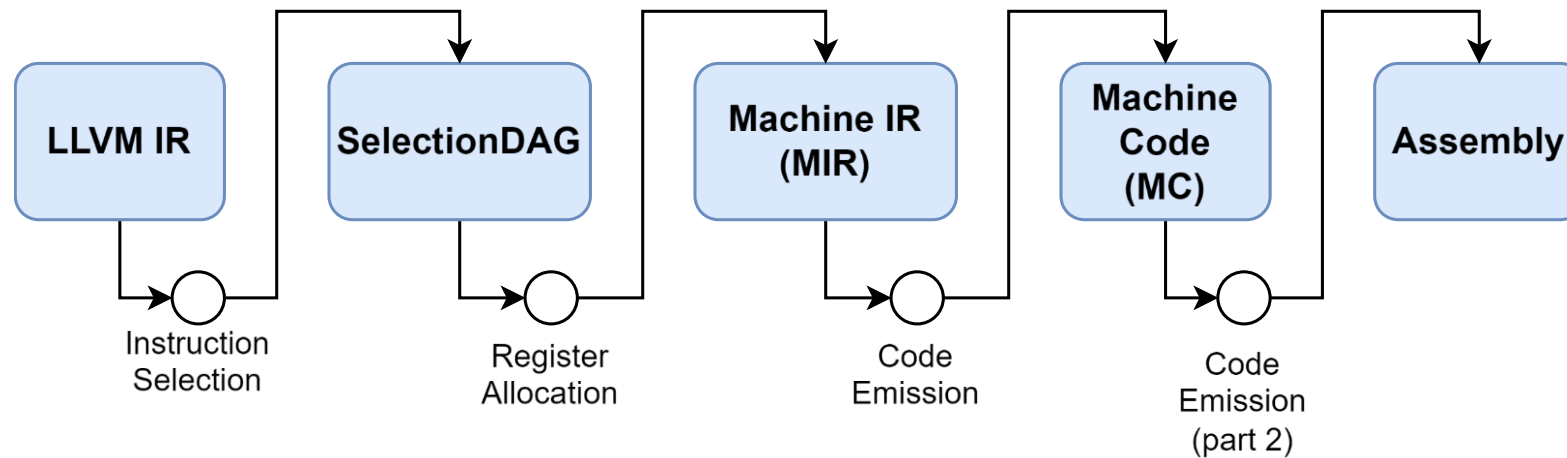
```
typedef struct X {  %struct.X = type { i64, i64, i64, i64 }  
    unsigned long long x, y;  
    unsigned long long u, w;  
} X;  
  
unsigned long long mul(X x) {  
    return x.x * x.y + x.z + x.w;  
}  
  
define i64 @mul(ptr byval(%struct.X) align 8 %0):  
    %2 = getelementptr %struct.X, ptr %0, i32 0, i32 0  
    ...  
    %4 = getelementptr %struct.X, ptr %0, i32 0, i32 1  
    %7 = getelementptr %struct.X, ptr %0, i32 0, i32 2  
    ...  
    %9 = getelementptr %struct.X, ptr %0, i32 0, i32 3  
    ...
```



Числа Фибоначчи на LLVM IR

```
define i32 @func(i32) :  
entry:  
    %1 = icmp ult i32 %0, 2  
    br i1 %1, label %final, label %st  
st: ; main recursion entry  
    %2 = add i32 %0, -1  
    %3 = call i32 @func(i32 %2)  
    %4 = add i32 %0, -2  
    %5 = call i32 @func(i32 %4)  
    %6 = add i32 %3, %5  
    br label %final  
final:  
    %7 = phi i32 [%6, %st], [1, %entry]  
    ret i32 %7
```

Повторение: Back end



LLVM Selection DAG

Selected selection DAG: %bb.0 'mul:entry'

SelectionDAG has 21 nodes:

t0: ch,glue = EntryToken

t2: i64,ch = CopyFromReg t0, Register:i64 %0

t4: i64,ch = CopyFromReg t0, Register:i64 %1

t29: ch = SW<Mem:(store (s32) into %ir.x.addr)> t2, TargetFrameIndex:i64<0>, TargetConstant:i64<0>, t0

t28: ch = SW<Mem:(store (s32) into %ir.y.addr)> t4, TargetFrameIndex:i64<1>, TargetConstant:i64<0>, t29

t9: ch = CopyToReg t0, Register:i64 %2, t2

t13: ch = CopyToReg t0, Register:i64 %3, t4

t23: ch = TokenFactor t9, t13, t28

t32: i64,ch = LW<Mem:(dereferenceable load (s32) from %ir.x.addr)> TargetFrameIndex:i64<0>, TargetConstant:i64<0>, t28

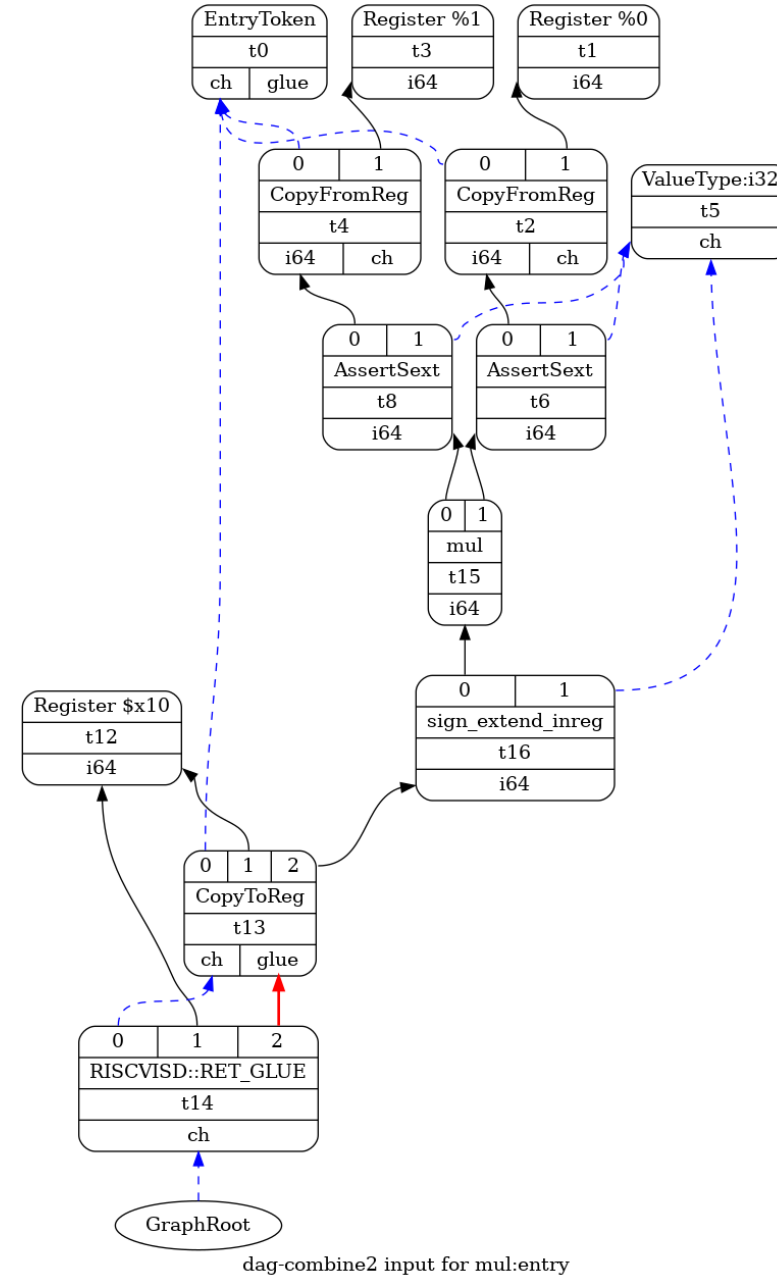
t30: i64,ch = LW<Mem:(dereferenceable load (s32) from %ir.y.addr)> TargetFrameIndex:i64<1>, TargetConstant:i64<0>, t28

t34: i64 = MULW t32, t30

t26: ch,glue = CopyToReg t23, Register:i64 \$x10, t34

t27: ch = PseudoRET Register:i64 \$x10, t26, t26:1

LLVM Selection DAG



LLVM Machine IR

```
# Machine code for function mul: NoPHIs, TracksLiveness, NoVRegs,  
TiedOpsRewritten, TracksDebugUserValues
```

```
Function Live Ins: $x10, $x11
```

```
bb.0.entry:
```

```
  liveins: $x10, $x11
```

```
  renamable $x10 = MULW killed renamable $x11, killed renamable $x10
```

```
  PseudoRET implicit killed $x10
```

```
# End machine code for function mul.
```

To be continued ...

На следующем занятии

- Рассмотрим необычные применения LLVM backend