Compilers 101

Optimizations – Part 1

Previously…

Preprocessing

Lexical analysis

Syntax analysis

Semantic analysis

IR Generation

IR Optimization

Optimization

Code generation

Fronten
d

Middle-end

Backend

Today

- PHI nodes
- LLVM Passes
- Local optimizations

Memory SSA

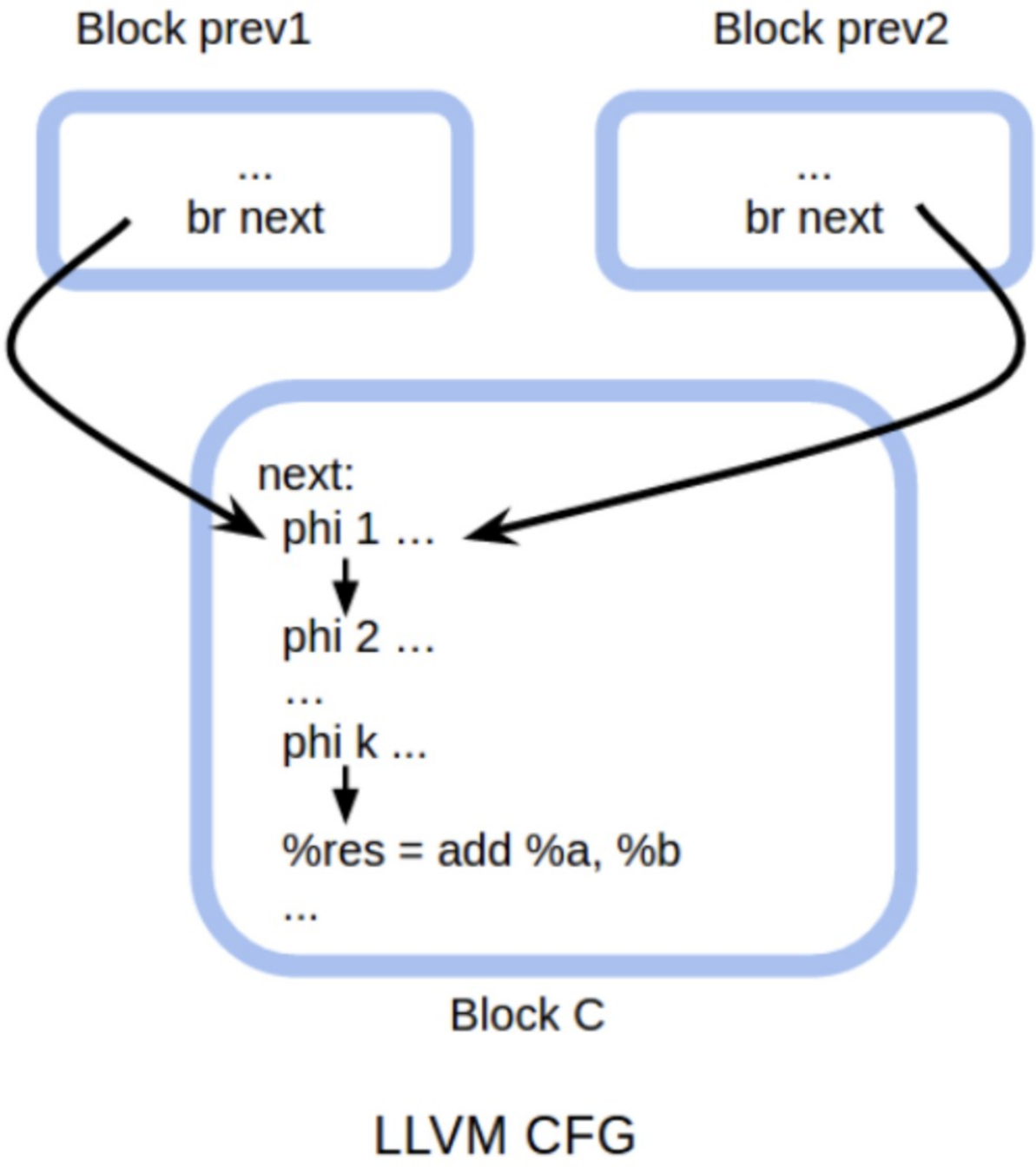
```
1   void bar(int);
2   void foo(int k) {
3       int a;
4       if (k > 10) {
5           a = 100;
6       } else {
7           a = k / 2;
8       }
9       bar(a);
10  }
```

```
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  call void @llvm.dbg.declare(metadata i32* %2,
  call void @llvm.dbg.declare(metadata i32* %3,
  %4 = load i32, i32* %2, align 4, !dbg !18
  %5 = icmp sgt i32 %4, 10, !dbg !20
  br i1 %5, label %6, label %7, !dbg !21

6:
  store i32 100, i32* %3, align 4, !dbg !22
  br label %10, !dbg !24

7:
  %8 = load i32, i32* %2, align 4, !dbg !25
  %9 = sdiv i32 %8, 2, !dbg !27
  store i32 %9, i32* %3, align 4, !dbg !28
  br label %10

10:
  %11 = load i32, i32* %3, align 4, !dbg !29
  call void @_Z3bari(i32 noundef %11), !dbg !30
  ret void, !dbg !31
```

https://godbolt.org/z/x6bf1fxxv

What are PHI nodes?

PHI node is a special kind of instruction in LLVM IR



Why PHI nodes?

- PHI nodes are essential for representing variable values that depend on the control flow in LLVM IR
- They solve the problem of SSA (Static Single Assignment) form constraints, where each variable is assigned exactly once but used multiple times in different control flow paths.
- PHI nodes effectively resolve the issue of variable assignments in loops and conditional branches, where the exact value of a variable can depend on the path taken through the program.

```
if (condition)
    x1 = value1; // Value assigned in the 'true' branch
else
    x2 = value2; // Value assigned in the 'false' branch

// After the merge point
x3 = PHI(x1, x2); // x3 takes the value of x1 or x2 depending on the path taken
```

Example

```
int func(bool cond) {
    int x;
    if (cond) {
        x = 1;
    } else {
        x = 2;
    }
    return x;
}
```

Original C++ code with condition

```
define i32 @func(i1 %cond) {
entry:
    br i1 %cond, label %if.then, label %if.else

if.then:                                          ; preds = %entry
    br label %if.end

if.else:                                          ; preds = %entry
    br label %if.end

if.end:                                           ; preds = %if.else, %if.then
    %x = phi i32 [ 1, %if.then ], [ 2, %if.else ]
    ret i32 %x
}
```

PHI nodes

```
define i32 @func(i1 %cond) {
entry:
    %x.addr = alloca i32
    br i1 %cond, label %if.then, label %if.else

if.then:                                          ; preds = %entry
    store i32 1, i32* %x.addr
    br label %merge

if.else:                                          ; preds = %entry
    store i32 2, i32* %x.addr
    br label %merge

merge:                                            ; preds = %if.else, %if.then
    %x = load i32, i32* %x.addr
    ret i32 %x
}
```

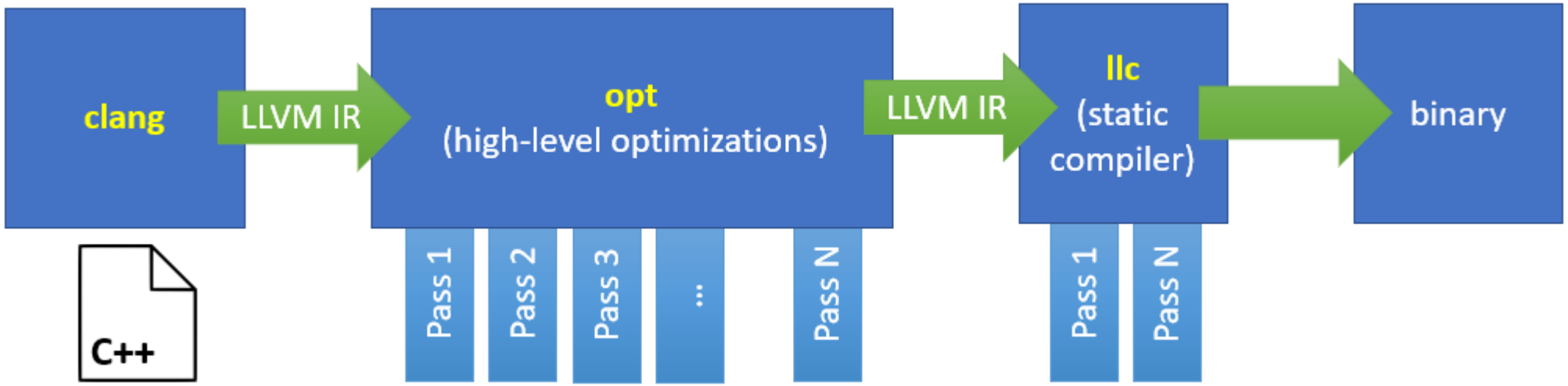
Memory SSA

Working with LLVM IR

- LLVM framework provides means to process LLVM IR, called Passes
- We usually want to work with particular abstractions in IR: functions, loops, etc
  - LLVM provides that too
- To run multiple passes on IR, you need a pass manager

Pass

A pass in a compiler refers to a complete traversal of the source code or intermediate representation (IR) to perform specific processing tasks, such as lexical analysis, syntax analysis, optimization, or code generation.



https://egorbo.com/opt-for-llvm-guide.html

LLVM pass

- Kinds of passes:
- Analysis
  - Examples:
    - Loop Analysis (loop-analysis): This group of analysis passes provides detailed information about loops, including their nesting structure, induction variables, and trip counts.
    - Basic Alias Analysis (basicaa): This pass provides a very basic Alias Analysis implementation that can answer alias queries based on simple pointer analysis. It's a foundation for more complex analyses.
- Transformation
- Utility - for utility functions like printing the IR to a file for debugging purposes

Pass types

Passes are split by types:

- ModulePass
  - operates on the whole compilation unit
  - use case: optimizations or analyses that need to consider multiple functions at once or need to deal with global variables
- FunctionPass
  - operates on one function
  - optimizations or analyses that are local to a function, such as dead code elimination, constant propagation, or loop optimizations. These passes can run independently on each function, making them suitable for parallel execution.

Other types of passes

- LoopPass
  - Specifically designed to operate on loops within a function. A LoopPass runs on each loop in a function, making it suitable for loop transformations and analysis.
    - Examples: loop unrolling, vectorization, or loop-invariant code motion.
- CallGraphSCCPass
  - Operates on strongly connected components (SCCs) in the call graph. An SCC is a subset of functions that are mutually recursive. This pass allows optimizations that need to consider recursion and how functions interrelate more deeply than simple function-to-function calls.
    - In practice used in AMD passes
- RegionPass
  - Targets a control-flow region within a function. A region can be a single basic block up to the entire function. Region passes allow for optimizations and analyses on these intermediate structures, which can be particularly useful for certain kinds of control flow and data flow optimizations.
    - Used in some control flow graph passes

Legacy vs new pass managers (PM)

- Legacy PM has been deprecated
- Note:
  - There are a lot of old information relative to old pass manager on the Internet
  - Consider reading carefully
  - Official docs are still based on the legacy PM in some places
- Good entrypoint for reading the docs:
  - https://llvm.org/docs/NewPassManager.html

Analysis vs Transformation

- Passes run on some piece of IR (module, function, etc)
- Analysis passes produce results lazily
  - Passes need to request the result first
  - Results are cached
- Transformation passes modify the IR

Writing a pass

```
struct HelloWorld : llvm::PassInfoMixin<HelloWorld>
{
  llvm::PreservedAnalyses run(llvm::Function &F,
                              llvm::FunctionAnalysisManager
&FAM) {
    llvm::dbgs() << F.getName() << “\n”;
    return llvm::PreservedAnalyses::all();
  }
}
```

Registering a pass

```
ModulePassManager MPM;
FunctionPassManager FPM;
FPM.addPass(HelloWorld());
MPM.addPass(createModuleToFunctionPassAdaptor(std::
move(FPM)));
```

Why optimizing IR?

- IR generation produces a lot of redundant code
- Programmers are lazy

Challenges

- Optimizer should not change observable behavior
- Optimizations should not regress code performance
- Optimized code must be of reasonable size
- Almost all interesting optimizations are NP-hard

Optimization targets

- Runtime – make programs run faster
- Memory usage – make programs use less memory
- Code size – produce smaller programs
- Power consumption – select instructions that use less power at the cost of performance
- ...

-O<level> clang flags (1)

- -O0, -O1, -O2, -O3
  - -O0: This level applies no optimizations; it aims for the fastest compilation time.
  - -O1: This level applies a minimal set of optimizations that reduce code size and execution time without significantly increasing the compilation time
  - -O2: Applies a broad set of optimizations that aim to improve code execution speed without incurring excessive compilation time. It does not perform optimizations that increase the size of the code significantly.
  - -O3: This level enables more aggressive optimizations than -O2, including optimizations that may significantly increase code size (e.g., loop unrolling). It aims for the highest execution speed, regardless of compilation time or code size.

-O<level> clang flags (2)

- -Os
  - Optimizes for code size without significantly compromising execution speed.
  - Use case: Good for systems with limited memory or when distributing smaller binaries is desirable.
- -Oz
  - Similar to -Os but more aggressive in reducing code size, possibly at a greater cost to execution speed.
  - Use case: Used for applications where the smallest possible code size is paramount, such as embedded systems with very limited memory resources.
- -Ofast
  - All the optimizations from -O3 + optimizations that are not standard-compliant but are likely benefit performance. E.g., enables aggressive math optimizations.
  - Use Case: Suitable for applications where execution performance is top priority and where deviations from standard math behavior are acceptable.
- -Og
  - Provides a good opt level while maintaining reasonable debug capabilities.

Semantics-preserving optimizations

- An optimization is semantics-preserving if it does not alter the semantics of the original program
- Examples
  - Dead code elimination
  - Loop invariant code motion
- Non-examples
  - Replace bubble sort with quick sort

https://llvm.org/docs/Passes.html#always-inline-inliner-for-always-inline-functions
https://llvm.org/docs/Passes.html#inline-function-integration-inlining

Function inlining

- Simply insert function body at call site
- Not always profitable: there are heuristics to only inline small functions
  - Using ‘inline’ keyword in C/C++ only suggests compiler to inline function, not guarantees that
  - There is `__attribute__((always_inline))` used for guaranteed inlining by the compiler
- Usually more aggressive on GPUs and less aggressive on FPGAs

Note: there is an `__attribute__((noinline))` that can be used to explicitly prevent function inlining, ensuring that the function call remains as a normal function call instead of being inlined by the compiler

https://llvm.org/docs/Passes.html#dce-dead-code-elimination
https://llvm.org/docs/Passes.html#globaldce-dead-global-elimination
https://llvm.org/docs/Passes.html#adce-aggressive-dead-code-elimination

Dead instruction elimination

- Example:

```
%0 = add i32 %arg0, %arg1
%1 = sub i32 %arg0, %arg1
ret %1
; %0 is not used on RHS
```

Liveness analysis

- A variable is live at some point if it holds a value that may be needed in the future
- Analysis is performed from the end of the function
- Given block s:

$$\mathrm{LIVE}_{in}[s] = \mathrm{GEN}[s] \cup (\mathrm{LIVE}_{out}[s] - \mathrm{KILL}[s])$$
$$\mathrm{LIVE}_{out}[final] = \emptyset$$
$$\mathrm{LIVE}_{out}[s] = \bigcup_{p \in succ[s]} \mathrm{LIVE}_{in}[p]$$

$$\mathrm{GEN}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{x_1, \ldots, x_n\}$$
$$\mathrm{KILL}[d : y \leftarrow f(x_1, \cdots, x_n)] = \{y\}$$

https://en.wikipedia.org/wiki/Live_variable_analysis

https://llvm.org/docs/Passes.html#dse-dead-store-elimination

Dead store elimination

- Removes stores to local variables, that are never read
- LLVM variant only considers a single basic block
- Example:

```
store i32 0, i32* %ptr    <- going to be removed
store i32 42, i32* %ptr
```

https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register

Store to load forwarding

- Replace loads from memory with SSA values
- Example:

```
store i32 42, i32* %ptr
%0 = load i32* %ptr
add i32 %0, 1
```

Replace with:

```
add i32 42, 1
```

https://llvm.org/docs/Passes.html#reg2mem-demote-all-values-to-stack-slots

Spilling registers on stack

Inverse of mem2reg pass

```
1   define i32 @add(i32 %a, i32 %b) {
2   entry:
3       %sum = add i32 %a, %b
4       ret i32 %sum
5   }
```

Why?

- For memory sanitizing
- Converting SSA to Non-SSA for Legacy Compilers
- Software Fault Injection & Fuzzing
- Lowering for Stack-Based Architectures

```
1   define i32 @add(i32 %a, i32 %b) {
2   entry:
3       %sum = alloca i32
4       %a_addr = alloca i32
5       %b_addr = alloca i32
6
7       store i32 %a, i32* %a_addr
8       store i32 %b, i32* %b_addr
9
10      %a_val = load i32, i32* %a_addr
11      %b_val = load i32, i32* %b_addr
12
13      %sum_val = add i32 %a_val, %b_val
14      store i32 %sum_val, i32* %sum
15
16      %result = load i32, i32* %sum
17      ret i32 %result
18  }
```

https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions
https://llvm.org/docs/Passes.html#aggressive-instcombine-combine-expression-patterns

Constant folding

- Example:

```
%0 = add i32 42, 1
%1 = add i32 %arg0, %0
```

Equivalent to:

```
%0 = add i32 %arg0, 43
```

https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions
https://llvm.org/docs/Passes.html#aggressive-instcombine-combine-expression-patterns

Arithmetic simplification

- Example:

%0 = mul %arg0, 4

Equivalent to

%0 = shl %arg0, 2

https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions
https://llvm.org/docs/Passes.html#aggressive-instcombine-combine-expression-patterns

Combine redundant instructions

- Implementation is rules based in LLVM
- A variation of arithmetic simplification
- Often called peephole optimization
- Example

```
%1 = add i32 %0, 1
%2 = add i32 %1, 1
```

Equivalent to

```
%1 = add i32 %0, 2
```

Common subexpression elimination (CSE)

- Example: a = b + c + d; e = b + c + f; => t = b + c; a = t + d; e = t + f;
- Principle
  - An expression a op b is available at a point p in a program if:
    - Every path from the initial node to p evaluates a * b before reaching p
    - There are no assignments (stores) to a or b after the evaluation but before p

https://llvm.org/docs/Passes.html#gvn-global-value-numbering

Global value numbering

- GVN is a technique of determining equivalent computations in program
- GVN works by assigning a numeric value to expressions
- Two expressions have equal values if they are provably equal (thus algorithm requires SSA)

Memcpy optimizations

- Eliminates memcpy calls and replaces sets of stores with memset

```
1  void foo(char *a, int N) {
2      for (int i = 0; i < N; i++) {
3          a[i] = 10;
4      }
5  }
```

```
2  define dso_local void @_Z3fooPci(i8* nocapture noundef writeonly %0, i32 noundef %1)
3    %3 = icmp sgt i32 %1, 0
4    br i1 %3, label %4, label %6
5
6  4:                                                ; preds = %2
7    %5 = zext i32 %1 to i64
8    call void @llvm.memset.p0i8.i64(i8* align 1 %0, i8 10, i64 %5, i1 false), !tbaa !3
9    br label %6
10
11 6:                                                ; preds = %4, %2
12   ret void
13 }
```

https://godbolt.org/z/odsKjnxra

Many more…

LLVM Home | Documentation » User Guides » LLVM's Analysis and Transform Passes

LLVM's Analysis and Transform Passes

- Introduction
- Analysis Passes
  - **aa-eval**: Exhaustive Alias Analysis Precision Evaluator
  - **basic-aa**: Basic Alias Analysis (stateless AA impl)
  - **basiccg**: Basic CallGraph Construction
  - **da**: Dependence Analysis
  - **domfrontier**: Dominance Frontier Construction
  - **domtree**: Dominator Tree Construction
  - **dot-callgraph**: Print Call Graph to "dot" file
  - **dot-cfg**: Print CFG of function to "dot" file
  - **dot-cfg-only**: Print CFG of function to "dot" file (with no function bodies)
  - **dot-dom**: Print dominance tree of function to "dot" file
  - **dot-dom-only**: Print dominance tree of function to "dot" file (with no function bodies)
  - **dot-post-dom**: Print postdominance tree of function to "dot" file
  - **dot-post-dom-only**: Print postdominance tree of function to "dot" file (with no function bodies)
  - **globals-aa**: Simple mod/ref analysis for globals
  - **instcount**: Counts the various types of **Instructions**
  - **iv-users**: Induction Variable Users
  - **kernel-info**: GPU Kernel Info
  - **lazy-value-info**: Lazy Value Information Analysis
  - **lint**: Statically lint-checks LLVM IR
  - **loops**: Natural Loop Information
  - **memdep**: Memory Dependence Analysis
  - **print<module-debuginfo>**: Decodes module-level debug info
  - **postdomtree**: Post-Dominator Tree Construction
  - **print-alias-sets**: Alias Set Printer
  - **print-callgraph**: Print a call graph
  - **print-callgraph-sccs**: Print SCCs of the Call Graph
  - **print-cfg-sccs**: Print SCCs of each function CFG
  - **function(print)**: Print function to stderr
  - **module(print)**: Print module to stderr
  - **regions**: Detect single entry single exit regions
  - **scalar-evolution**: Scalar Evolution Analysis
  - **scev-aa**: ScalarEvolution-based Alias Analysis

Available passes list:

https://llvm.org/docs/Passes.html

Important notes on passes

- There is no silver bullet pass or pass pipeline
  - Different target architectures, programming domains have different demands
  - Different targets require different optimizations (what do we optimize?)
- Pass execution order matters
  - Some passes have prerequisites
  - Some passes are useless after other passes have been applied
- Reuse of already known implementations and concepts is important
  - But implementing something new may still be necessary

Running optimizations

- Use `opt` tool
- See full list of available passes names: https://llvm.org/docs/Passes.html (`opt --print-pass-names`)

Typical workflow:

1. Get LLVM IR:

```
clang -S -emit-llvm example.c -o example.ll
```

2. Apply optimizations

```
opt -S -mem2reg example.ll -o optimized.ll
```

```
opt -S -loop-unroll -instcombine -simplifycfg example.ll -o
optimized.ll
```

Next time…

- Loop optimizations

Test

https://forms.gle/VaqaruCTsACXQ6zf7

Submission time: **10 minutes**

По каким признакам пассы классифицируются? Приведите примеры классификаций, приведите примеры пассов для каждой категории

Your answer

Приведите примеры каких-либо LLVM IR пассов с их описанием (больше - лучше)
Note: точное название пасса и флага не требуется

Your answer

Backup: me@gooddoog.ru

Extra materials

- LLVM IR language reference - https://llvm.org/docs/LangRef.html
- Single-Static Assignment Form and PHI - https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/control-structures/ssa-phi.html
- LLVM Developers' Meeting: A. Warzynski "Writing an LLVM Pass: 101" https://www.youtube.com/watch?v=ar7cJl2aBuU
- List of LLVM Analyses and Transformations - https://llvm.org/docs/Passes.html