编译原理

Complier Principles

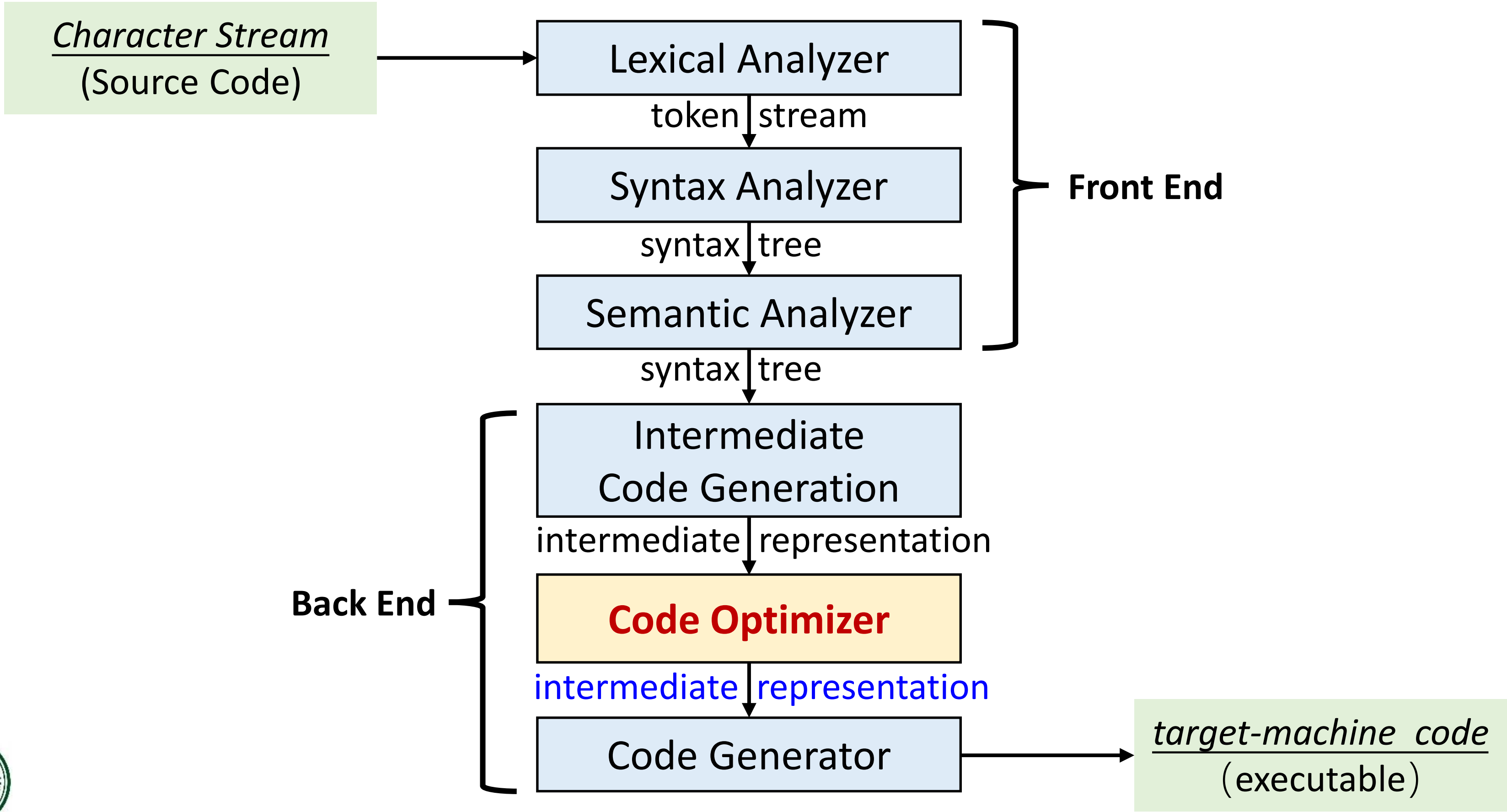
Lecture 9
Intermediate Code Optimization

赵帅

计算机学院

中山大学

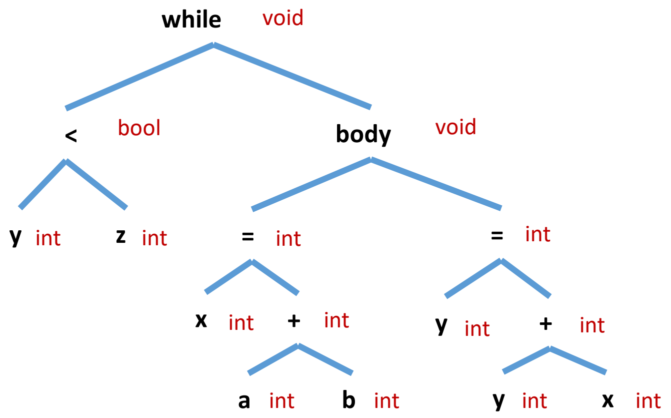
Compilation Phases[编译阶段]



Compilation Phases[编译阶段]

```
while (y<z){
    int x = a + b;
    y += x;
}
```

(keyword, **while**)
(id, **y**)
(sym, **<**)
(id, **z**)
(id, **x**)
(id, **a**)
(sym, **+**)
(id, **b**)
(sym, **;**)
(id, **y**)
(sym, **+=**)
(id, **x**)
(sym, **;**)

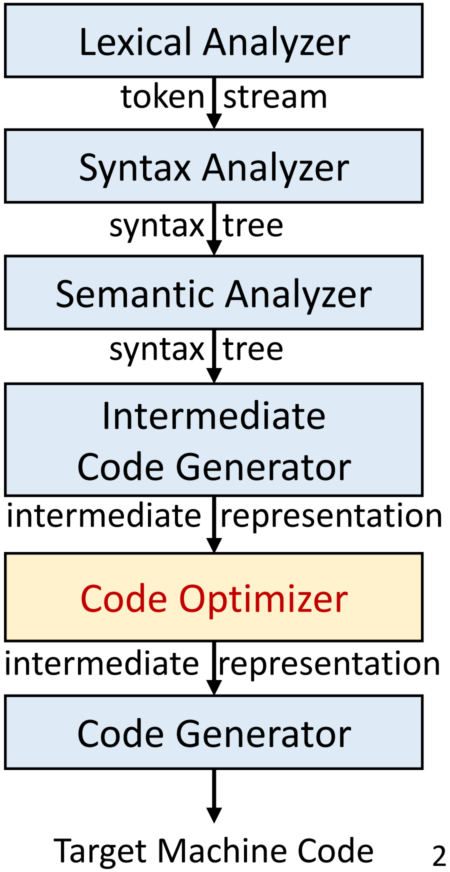


Annotated AST/Decorated AST
[带标注的抽象语法树]

```
goto L1
L2:
        t1 := a + b
        x := t1
        t2 := y + x
        y := t2
L1:
        if y < z goto L2
```

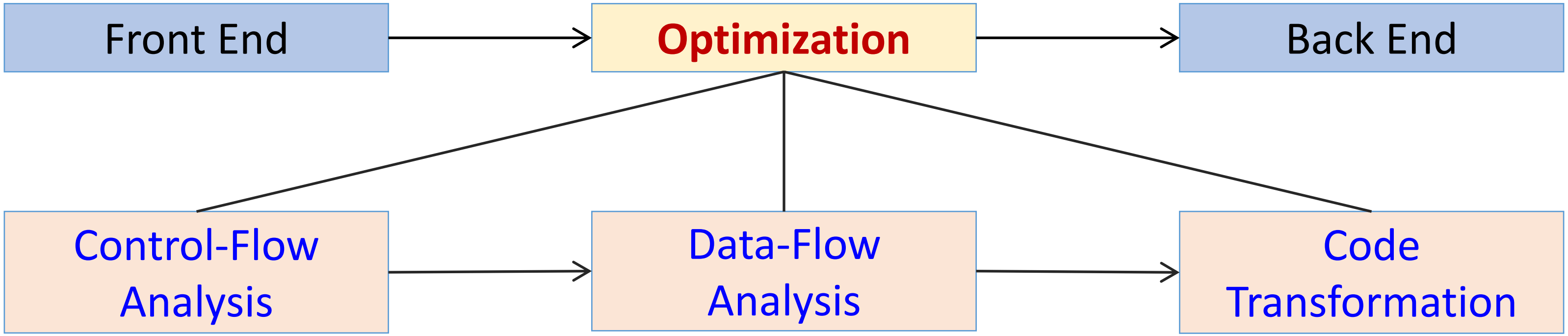
```
t1 = a + b
goto L1
L2:
        t2 := y + t1
        y := t2
L1:
        if y < z goto L2
```

Quiz: 这里做了什么优化?



Optimization[代码优化]

- What we have now?
  - IR of the source program (+ symbol table)
- Goal of the optimization?
  - Improve the efficiency and performance of the code
    - reducing execution time, memory usage, computational complexity.
  - This helps to make the code faster, more scalable, and able to handle increasing workloads.



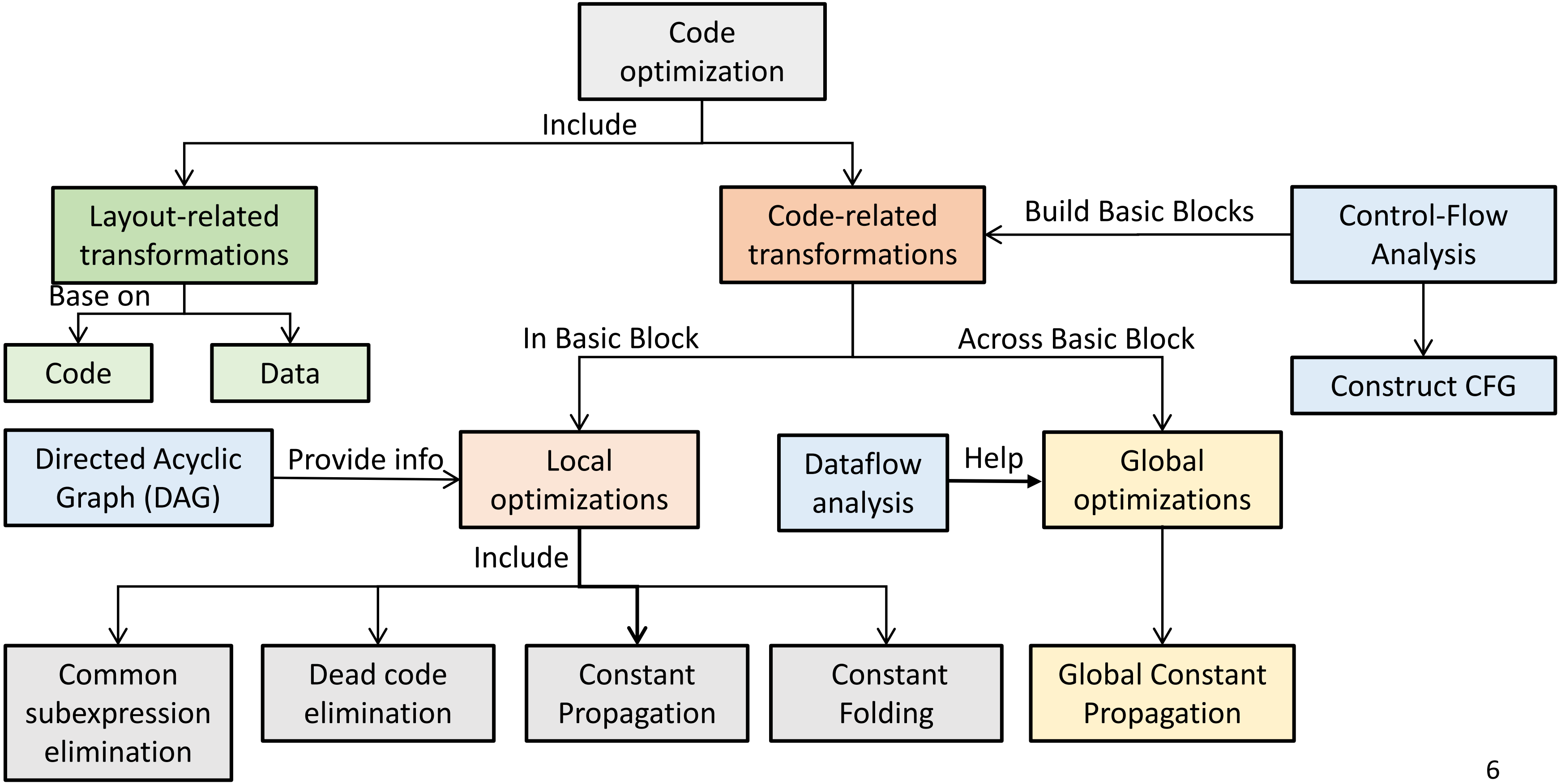
To Optimize: Who When Where?

- **Manual:** source code[人工, 源码]
  - ◆ Select appropriate algorithms and data structures
  - ◆ Write code that the compiler can effectively optimize
    - ▫ Need to understand the capabilities and limitations of compiler opts
- **Compiler**: intermediate representation[编译器, IR] Focus!
  - ◆ To generate more efficient TAC instructions
- **Compiler**: final code generation[编译器, 目标代码]
  - ◆ Selecting effective instructions to emit, allocating registers in a better way
- **Assembler/Linker:** after final code generation[汇编/链接, 目标代码]
  - ◆ Attempting to re-work the assembly code itself into something more efficient (e.g., link-time optimization)

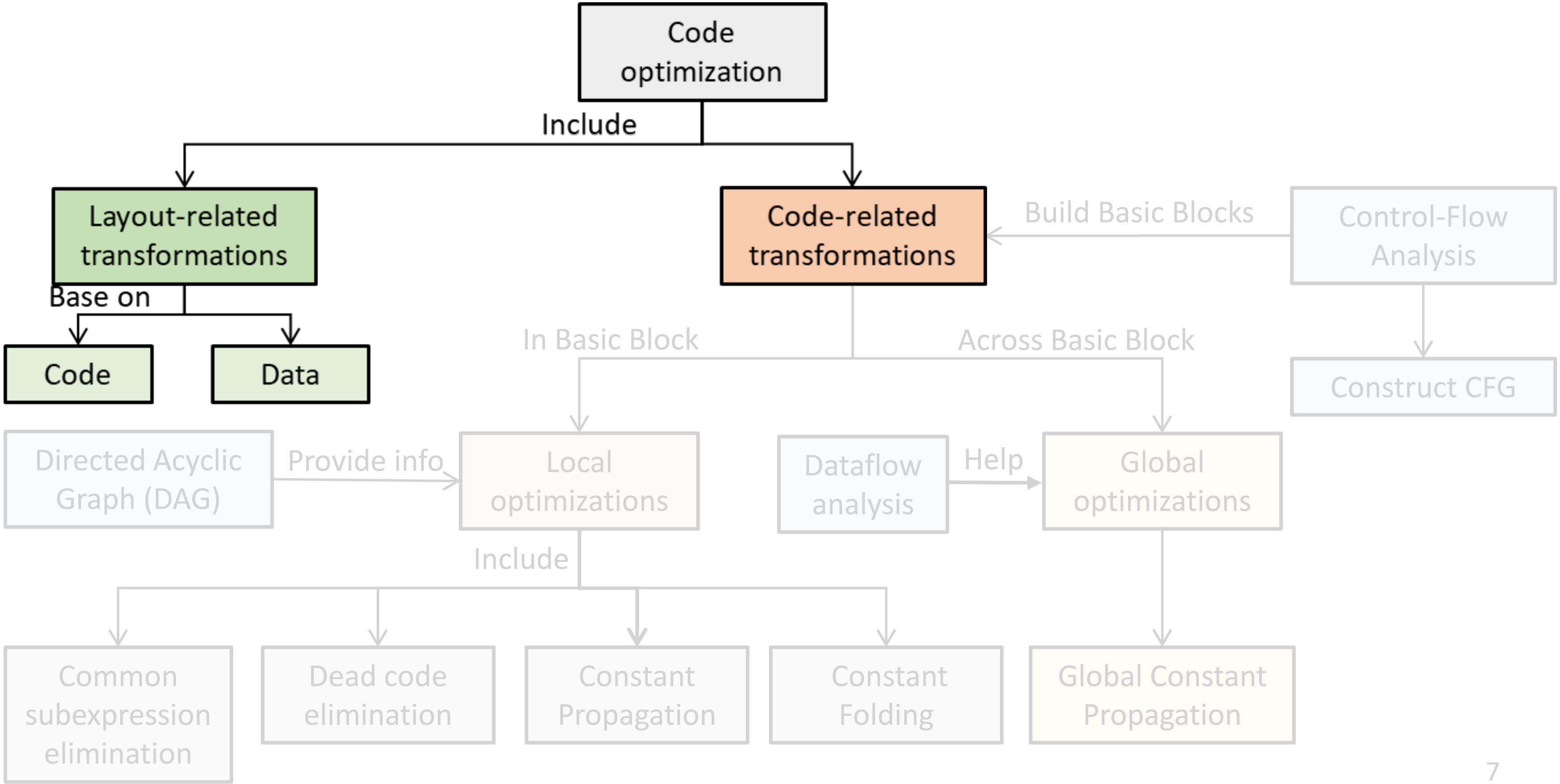
Overview of Optimizations

- Better one or more of the following (in the average case)
  - Execution time.
  - Memory usage.
  - Energy consumption.
  - Binary executable size...
- Some Principles
  - Equivalence: after optimization, the result should not be changed.
  - Efficient: make the optimized target code run in less time and less storage space is occupied.
  - Cost-effectiveness: should be as low as possible to obtain optimized results.

Code Optimization[代码优化]



Code Optimization[代码优化]

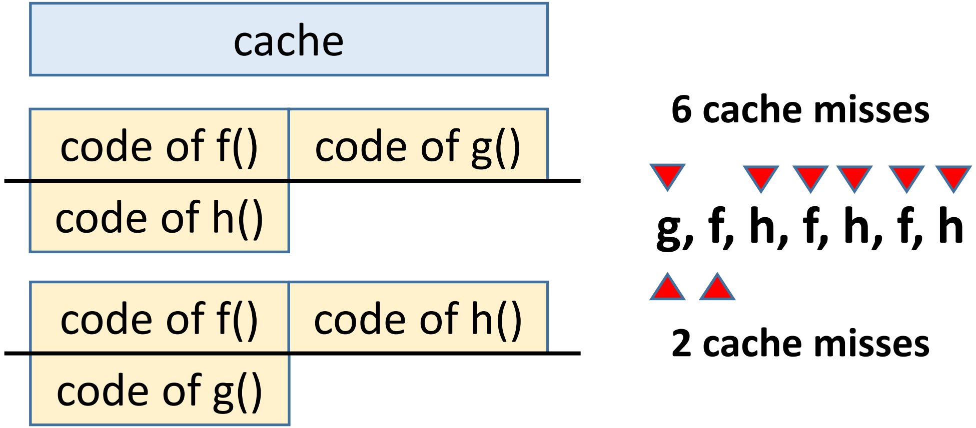


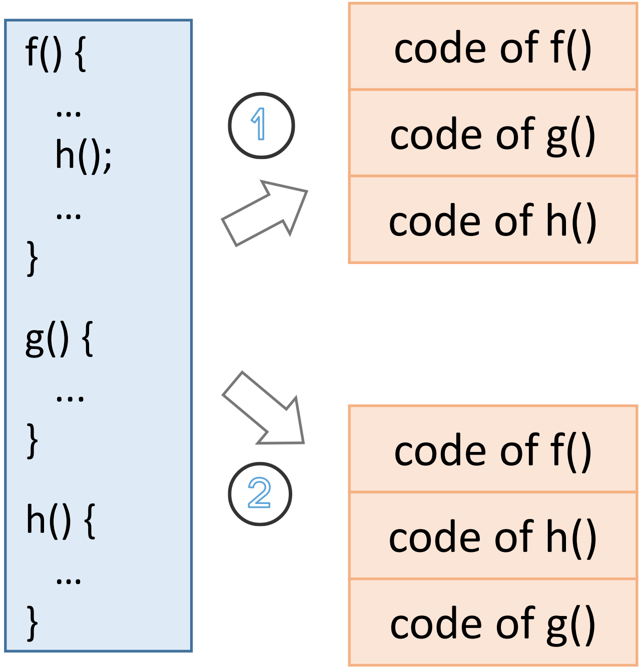
Types of Optimizations

- Compiler optimization is essentially a transformation[转换], including deleting/adding/moving/modifying.
- **Layout-related** transformations[布局相关]
  - Optimizes where in memory code and data is placed.
  - Goal: maximize the **spatial locality** [空间局部性], i.e., maximize the possibility that nearby locations will also be accessed soon during an access.
- **Code-related** transformations[代码相关] Focus!
  - Optimizes what code is generated.
  - Goal: execute least number of most costly instructions.

Layout-Related: Code

- For example, given two ways to layout code:
  - Which code layout is better?, assuming:
    - data cache has one N-word line
    - the size of each function is N/2-word long
    - access sequence is **"g, f, h, f, h, f, h"**





Layout-Related : Data

- **Example 1:** Change the variable declaration order

```
struct S {
    int a;
    int b[10];
    int c;
} obj[100];

for(...) {
    ... = obj[i].a + obj[i].c;
}
```

⇨

```
struct S {
    int a;
    int c;
    int b[10];
} obj[100];

for(...) {
    ... = obj[i].a + obj[i].c;
}
```

- After improving, a and c are likely resided in the same cache line.
- Futhermore, access to c will always hit in the cache.

Layout-Related : Data

- **Example 2:** Change AOS (array of structs) to SOA (struct of arrays)

```
struct S {
    int a;
    int b;
} objs[100];

for(...) {
    ... = objs[i].a + 1;
}
for(...) {
    ... = objs[i].b + 1;
}
```

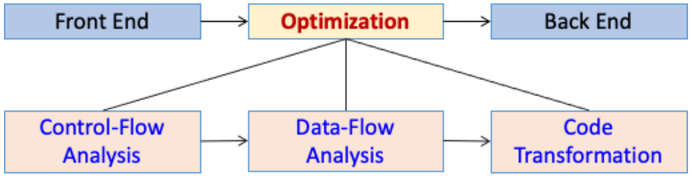
```
struct S {
    int a[100];
    int b[100];
} objs;

for(...) {
    ... = objs.a[i] + 1;
}
for(...) {
    ... = objs.b[i] + 1;
}
```

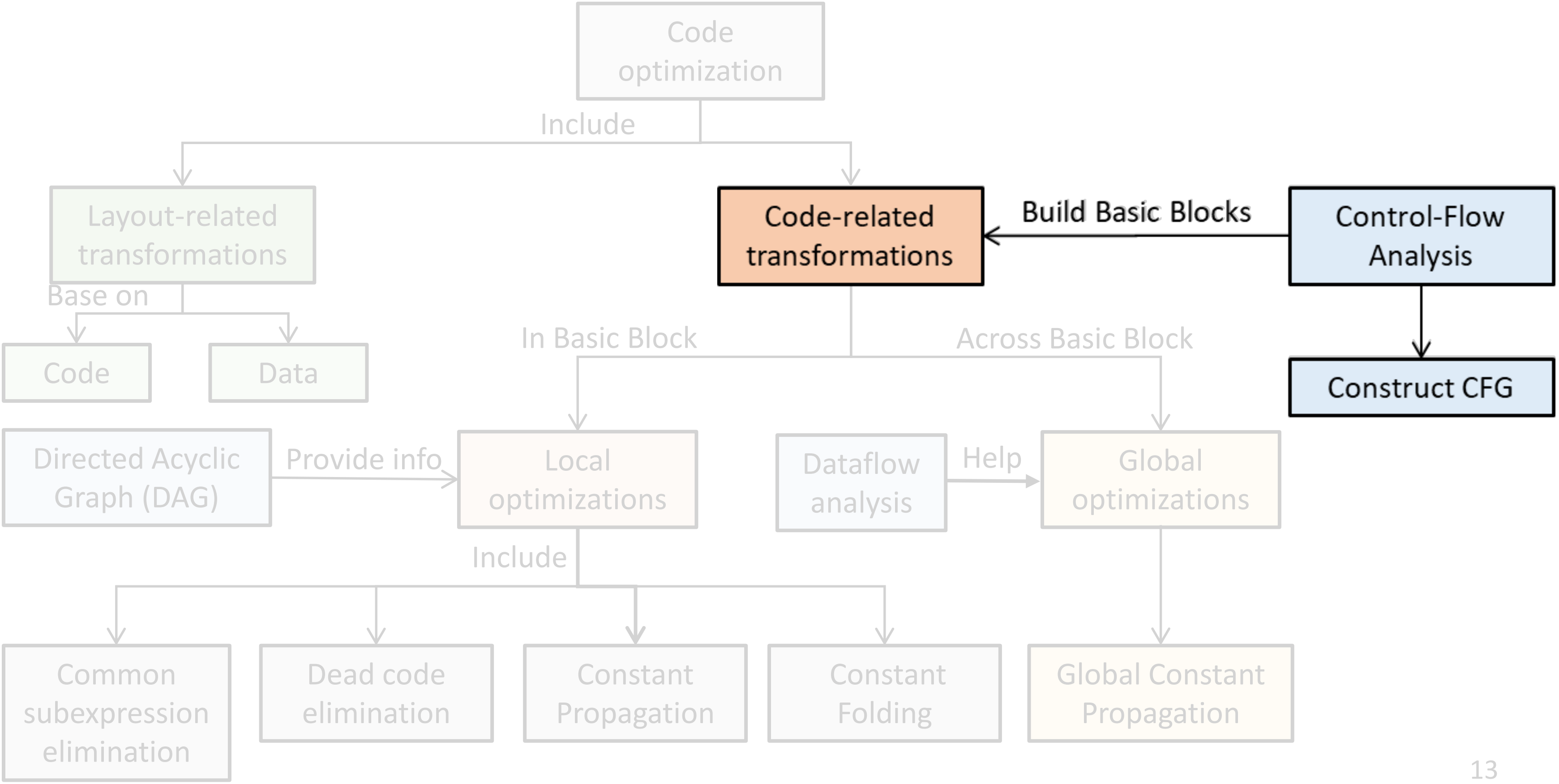
- Improved **spatial locality** for accesses to ‘a’s and ‘b’s

IR Optimization Revisit

- Goal of Optimization:
  - Execution time / Memory usage / Energy consumption / Binary executable size...
  - Equivalence / Efficient / Cost-effectiveness
- **Layout-related** transformations[布局相关]
  - Optimizes where in memory code and data is placed.
  - Goal: maximize the **spatial locality** [空间局部性].
    - code layout, declaration ordering, data structuring
- **Code-related** transformations[代码相关]
  - Optimizes what code is generated.
  - Goal: execute least number of most costly instructions.



Code Optimization[代码优化]



Code-Related Optimizations

- **Modifying** (e.g., strength reduction)
  - A = 2 * a; → A = a << 1;
- **Deleting** (e.g. dead code elimination)
  - A = 2; A = y; → A = y;
- **Moving** (e.g. code scheduling)

  A and C can be parallel

  - A = x / y; B = A + 1; C = y; → A = x / y; C = y; B = A + 1;
- **Inserting** (e.g., data prefetching[数据预取])
  - while(p != NULL) {process(p); p = p -> next;}
  - while(p != NULL) {prefetch(p -> next); process(p); p = p -> next;}

Now access to ‘p->next’ is likely to hit in cach

Control-Flow Analysis

- For many imperative programming languages[命令式编程], the control flow of a program is explicit in a program's source code.
  - As a result, control-flow analysis (CFA) [控制流分析] usually refers to a static analysis for determining the receiver(s) of function calls in programs written in a higher-order programming language.

- CFA is a static code analysis for determining the control flow of a program. The control flow is expressed as control-flow graph (CFG).

- To build CFG, we first divide the code into basic blocks.

Basic Block

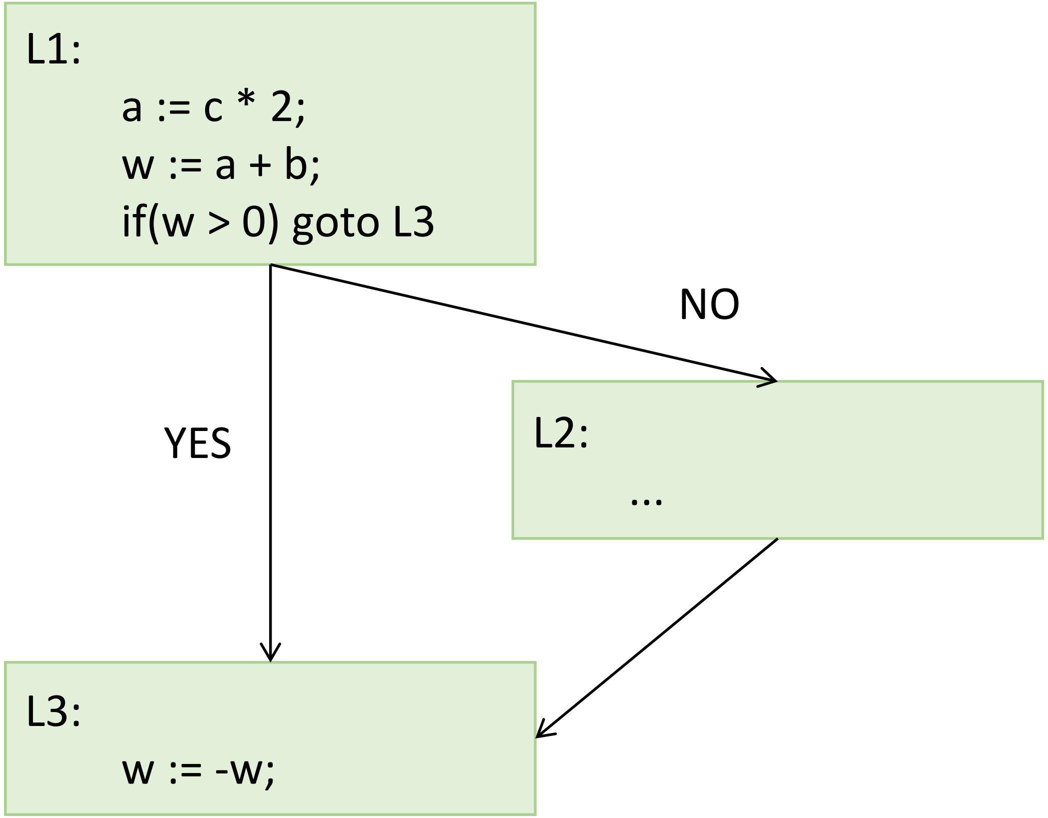
- A **basic block**[基本块] is a straight-line code sequence that
  - Except the first instruction, there are no other labels[只有第一条进入]
  - Except the last instruction, there are no jumps[只有最后一条跳出]
- The code in a basic block has:
  - ◆ **One entry point**, Can only jump into the beginning of a block
  - ◆ **One exit point**, Can only jump out at the end of a block
- Basic blocks forms the nodes in CFG.
  - cannot be divided further
  - All instructions in basic block execute or none at all [all or nothing]
  - decomposes programs to basic blocks as a first step in CFA.
- *Local optimizations* are limited to scope of one basic block.
- *Global optimizations* are across basic blocks.

Control Flow Graph[控制流图]

- A control flow graph is a directed graph in which:
  - Each **node** represents a basic block, i.e., a straight-line piece of code without any jumps or jump targets
  - Directed **edges** represent flow of execution between basic blocks
    - Flow from the end of one basic block to the begin of another
    - Flow can be result of a control flow divergence
    - Flow can be result of a control flow merge
  - Control statements introduce control flow edges
    - i.e., if-then-else, for-loop, while-loop, ...
- CFG is widely used
  - to represent a function
  - for program analysis, especially for global analysis and optimization

Example

```
L1:
      a := c * 2;
      w := a + b;
      if(w > 0) goto L3
L2:
      ...
L3:
      w := -w;
```

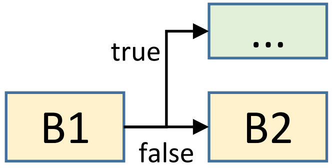


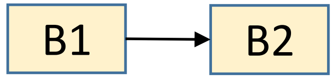
Construct CFG

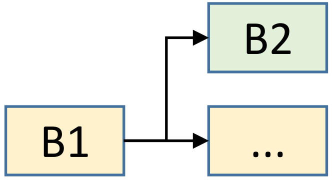
- **Step 1: partition code into basic blocks**[分解为基本块]
  - Identify the leaders instructions. An instruction is a leader if any of the following 3 conditions are met:
    - It is the first instruction.[首条指令]
    - The target of a conditional or an unconditional goto / jump. [跳转目标]
    - immediately follows a conditional or an unconditional goto / jump. [紧跟目标]
  - Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader.
  - Every basic block has a leader.

Construct CFG

- **Step 2 : add an edge between basic blocks B1 and B2 if**[连接基本块]
  - B2 follows B1, and B1 may "fall through" to B2[相邻]:
    1. B1 ends with a conditional jump to another basic block [若条件假，到达B2]
    2. B1 ends with a non-jump instruction (B2 is a target of a jump) [无跳转，B1顺序执行到达B2]
    - Note: if B1 ends in an unconditional jump, cannot fall through [B1无条件跳转，会绕开B2]
  - B2 doesn't follow B1, but B1 ends with a jump to B2 [不相邻，但B2是B1的跳转目标]

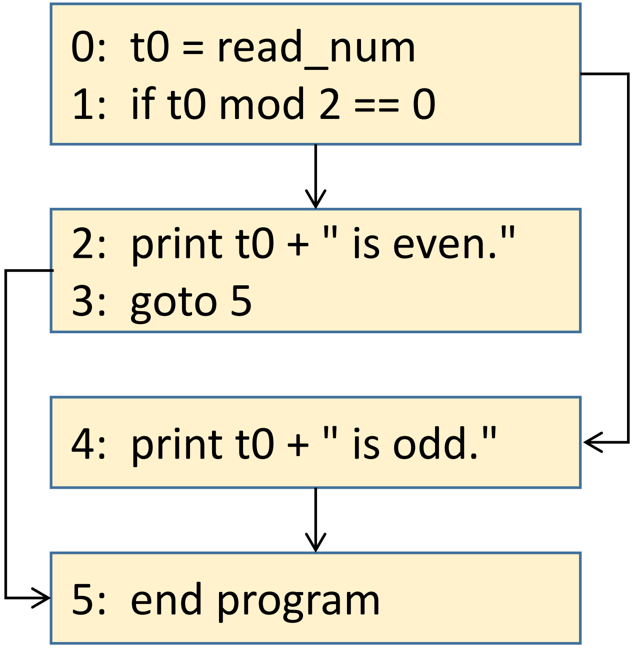






Example

```
0:        t0 = read_num
1:        if t0 mod 2 == 0
2:              print t0 + " is even."
3:              goto L1
4:        print t0 + " is odd."
5:  L1:  end program
```

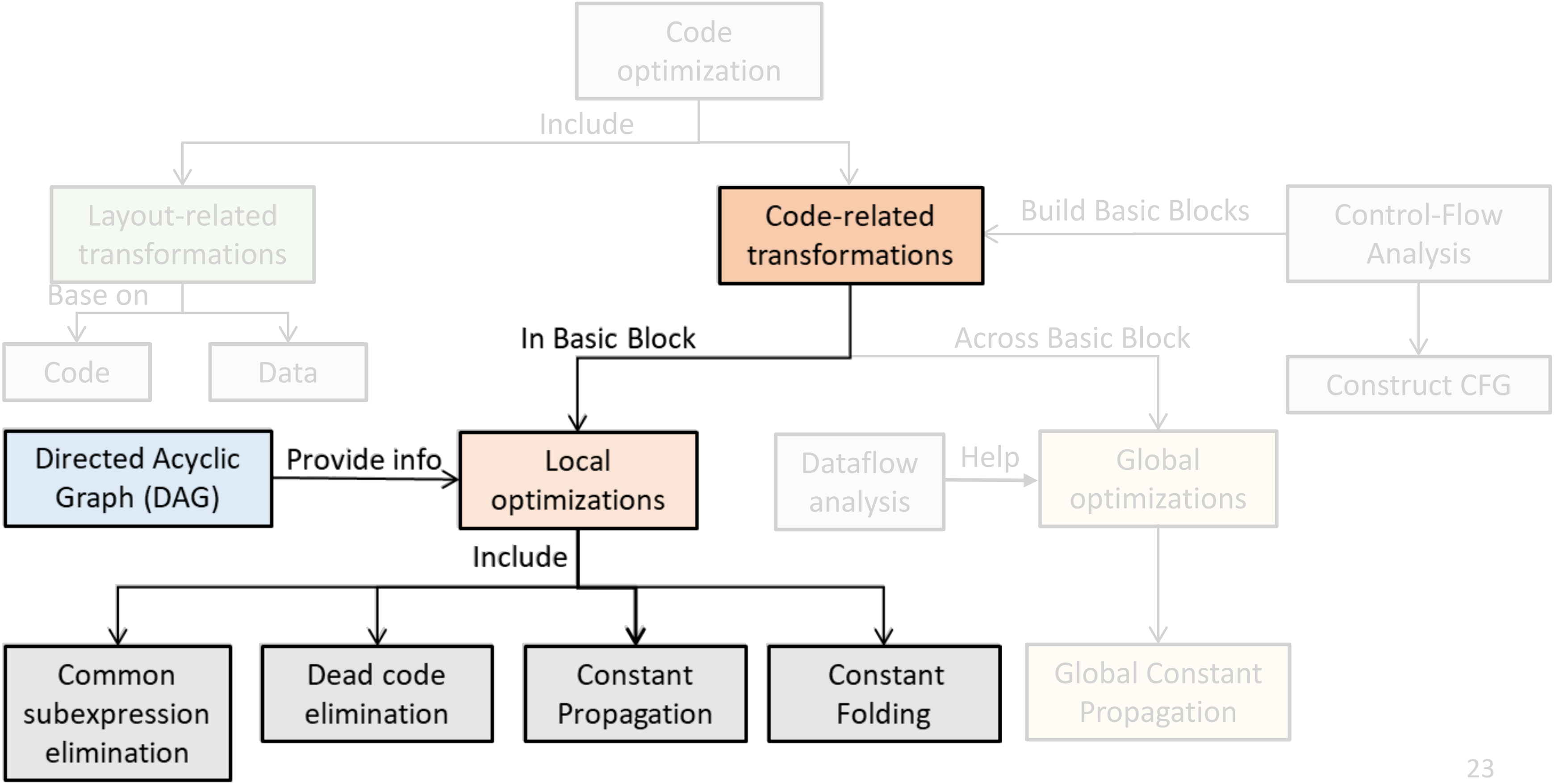


- Partition code into basic blocks.
  - the first instruction of a program: 0
  - target instructions of jump: 5
  - instructions immediately following jump: 2 ,4
- Add edges between basic blocks.

Local and Global Optimizations

- Local optimizations[局部]
  - Optimizations performed exclusively within a basic block.
  - Typically the easiest, never consider any control flow info.
    - All instructions in scope executed exactly once
  - Example:
    - *constant folding*[常量折叠, 在编译时直接计算常量表达式, 而不在运行时每次都计算]
      - const int a = 2 + 3; x=y+a; → x=y+5
    - *common subexpression elimination*[消除公共子表达式]
- Global optimizations[全局]
  - Optimizations performed across basic blocks.
    - Scope can contain if / while / for statements.
    - Some instants may not execute, or execute multiple times.
  - Note: global here doesn't mean across the entire program.
    - We usually optimize one function at a time.

Code Optimization[代码优化]



Example: Local Optimizations

- Common subexpression elimination (CSE)[删除公共子表达式]
  - Two operations are common if they produce the same.
    - It is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it[避免重复计算]
- Dead code elimination[删除无用代码]
  - If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream[结果从不使用]

```
y = x + z;
y = x * x + (x / 3);
z = x * x + y;
```

→

```
y = x + z;
t1 = x * x;
t2 = x / 3;
y = t1 + t2;
t3 = x * x;
z = t3 + y;
```

→

~~y = x + z;~~
t1 = x * x;
t2 = x / 3;
y = t1 + t2;
~~t3 = x * x;~~
z = t1 + y;

DAG of Basic Blocks

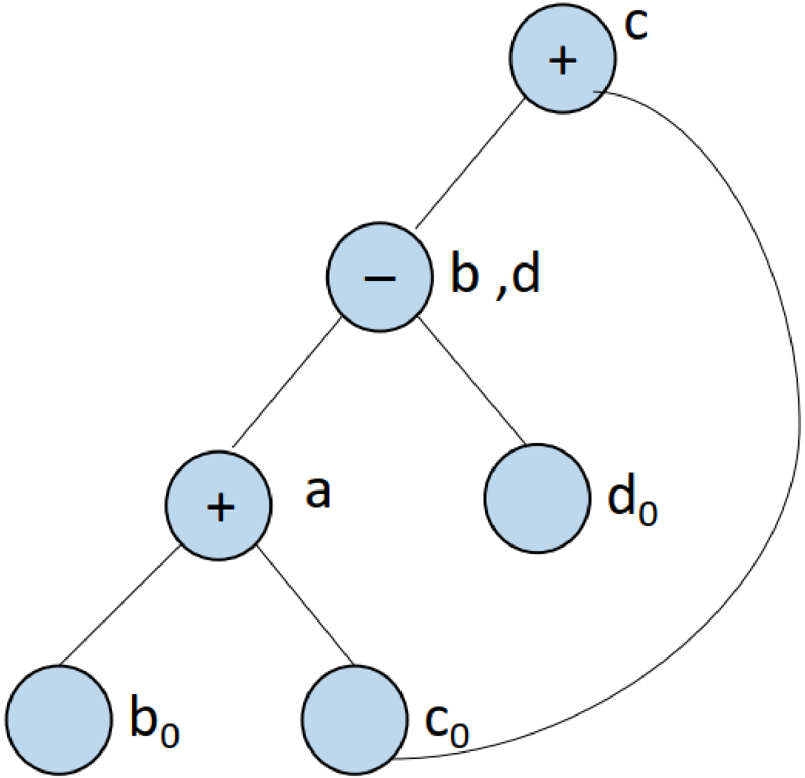
- The Directed Acyclic Graph (DAG) is used to represent the structure of a basic block
  - visualize the flow of values between basic blocks
  - provide optimization techniques in the basic block.
- A DAG is a three-address code.
  - ◆ is a type of data structure
  - ◆ facilitates the transformation of basic blocks.
  - ◆ is an efficient method for identifying common sub-expressions.

Algorithm: Construction of DAG

- Create a node for each initial value of the variables appearing in the basic block[为变量初始值创建节点 --- 叶子节点]
- Create a node N associated with each statement S within the block[为声明语句创建节点 --- 中间节点]
  - The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s[节点N的子节点为statement中操作符相关的节点]
  - Label N by the operator applied at S[用运算符标注节点N]
- Certain nodes are designated output nodes[某些节点为输出节点]

Example

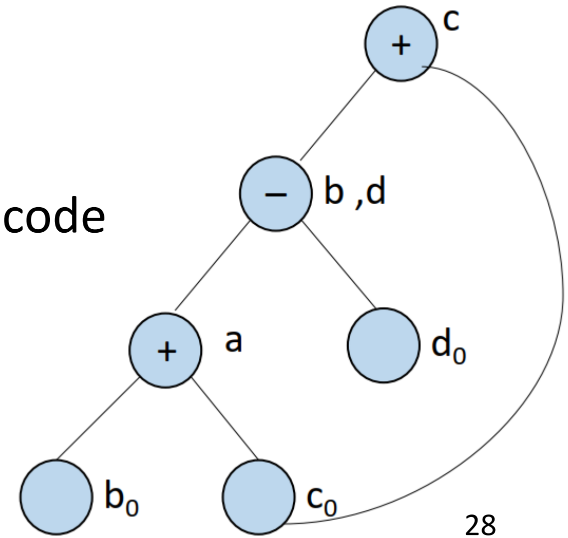
(1) a = b + c
(2) b = a – d
(3) c = b + c
(4) d = a – d



Local Opt.: Elimination

- If b is not live on exit from the block
  - No need to keep b = a – d
- If both b and d are live
  - Remove either (2) or (4) : **Common Subexpression Elimination**
  - Add a 4th statement to copy one to the other
- If only a is live on exit
  - Remove nodes from the DAG correspond to dead code
    - c -> (b,d) -> d_0
  - This is actually **dead code elimination**

(1) a = b + c
(2) b = a – d
(3) c = b + c
(4) d = a – d

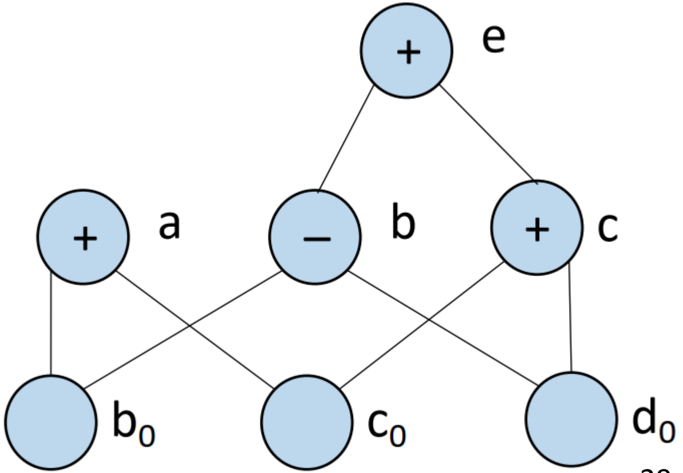


Local Opt.: Elimination

SUN YAT-SEN UNIVERSITY

- When finding common subexpressions, we really are finding expressions that are guaranteed to compute the same value, no matter how that value is computed[过于严苛]
    - Thus miss the fact that (1) and (4) are the same
        - $b + c = (b - d) + (c + d) = b_0 + c_0$
- **Solution**: algebraic identities[代数恒等式]

(1) a = b + c
(2) b = b – d
(3) c = c + d
(4) e = b + c



Using Algebraic Identities

- Eliminate computations by applying mathematical rules[使用数学规则]
  Identities: a * 1 ≡ a, a * 0 ≡ 0, b & true ≡ b
  - Reassociation and commutativity[重组合与交换]
    - (a + b) + c ≡ a + (b + c), a + b ≡ b + a
- Strength Reduction[强度削减]
  - Replacing expensive operations (multiplication, division) by less expensive operations (add, sub, shift)
  - Some ops can be replaced with cheaper ops

x = y / 8 → x = y >> 3
y = y * 8 → x = y << 3
x^2 → x * x
2 * x → x + x

Local Opt.: Constant Folding

- Constant Folding[常量折叠]
  - Computing operations on constants at compile time
  - Example:

```
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print(”error”);
```

  - After constant folding

```
x = 200;
if (false) print(”error”);
```

  - Dead code elimination can further remove the above *if* statement
  - Inherently local since scope limited to statement

Local Opt.: Constant Propagation

- Constant Propagation[常量传播]
  - Substituting values of known constants at compile time
  - Local Constant Propagation (LCP)

```
x = 4
y = x * 3
```
→
```
x = 4
y = 4 * 3
```
→
```
x = 4
y = 12
```

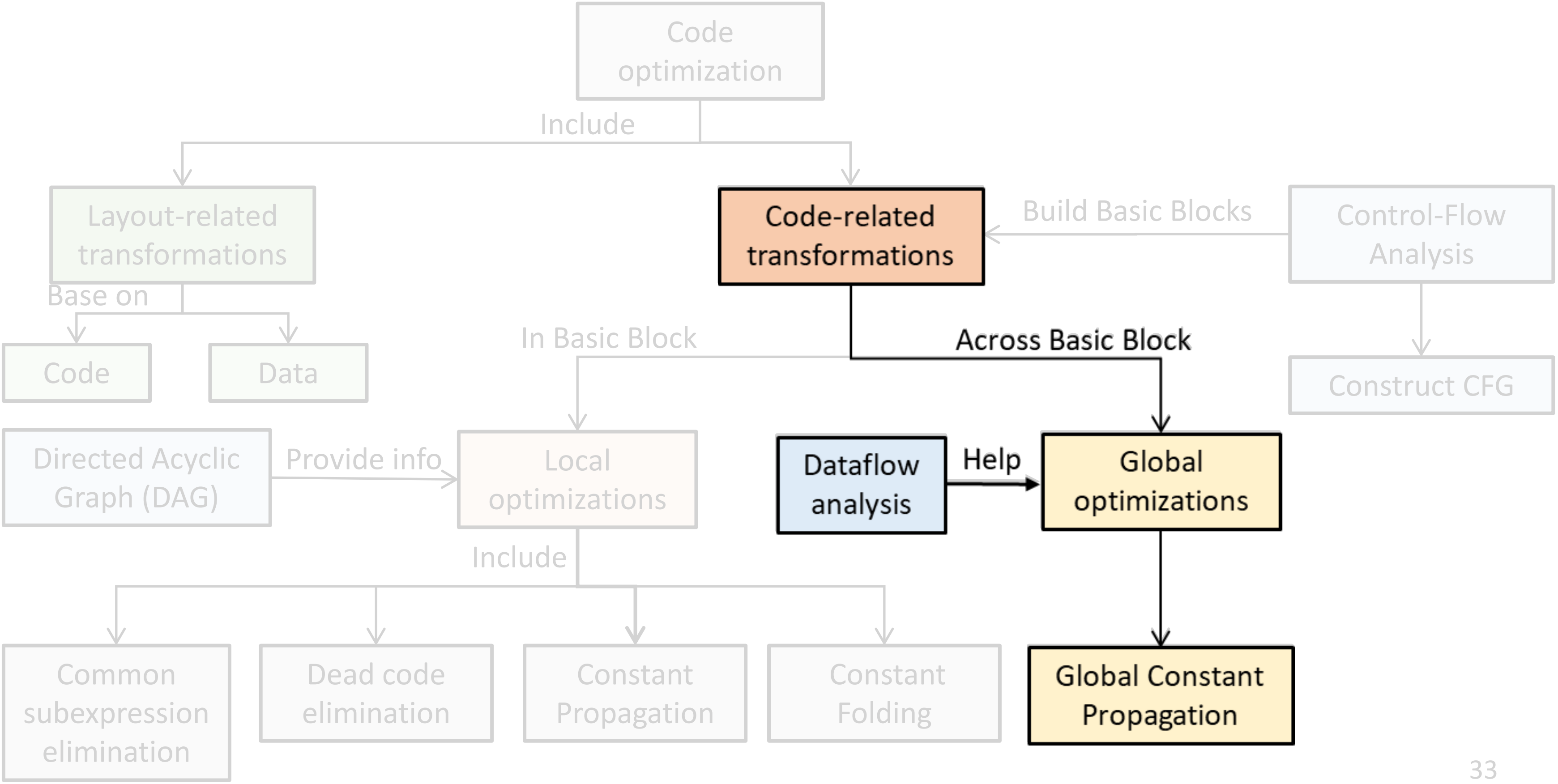
- Some optimizations have both local and global versions
  - Global Constant Propagation (GCP), more powerful but complicated

```
a = 1;
x = 3;
if (…)
        x = a + 2;
y = x;
```

```
a = 1;
x = 3;
if (…)
        x = 1 + 2;
y = x;
```

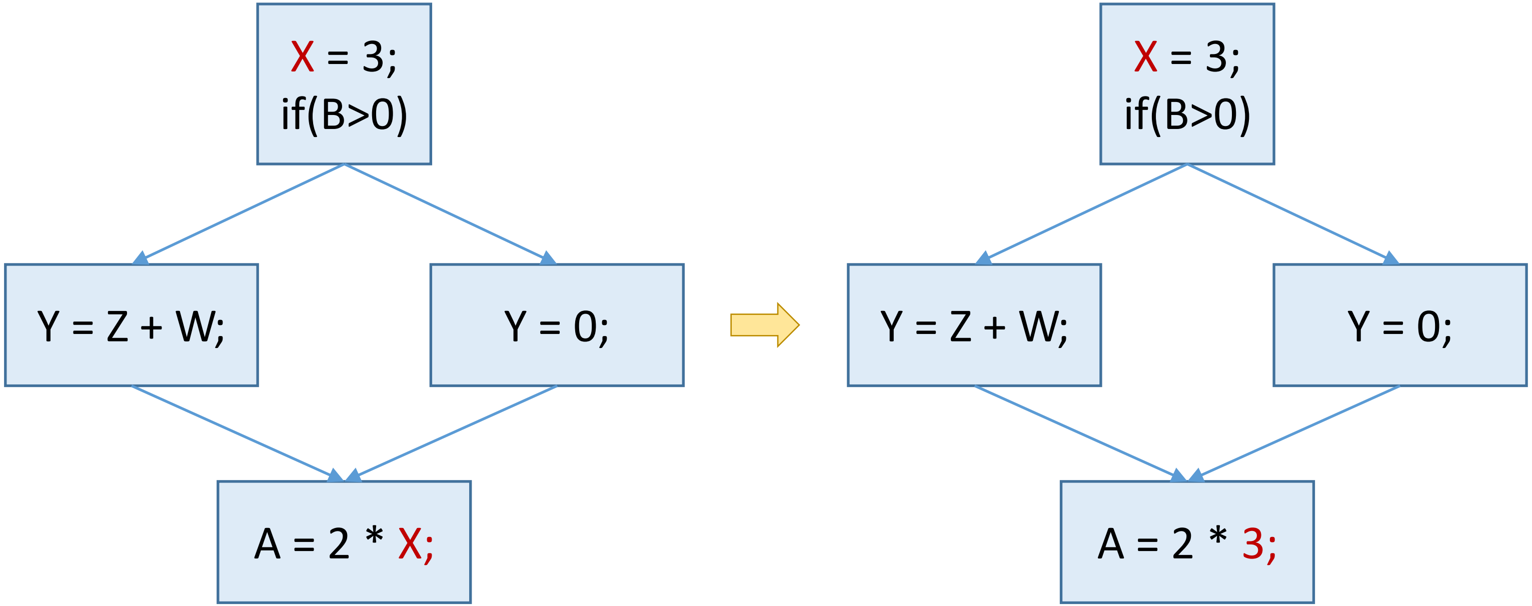
```
a = 1;
x = 3;
if (…)
        x = 3;
y = 3;
```

Code Optimization[代码优化]

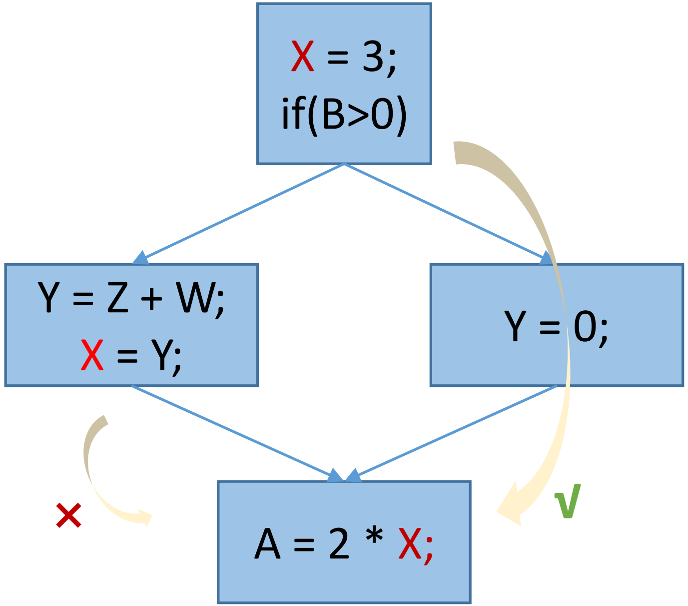


Global Optimizations

- Extend optimizations to flow of control, i.e., CFG
  - Along all paths.
  - Optimization must be stopped if incorrect in even one path



Global Optimizations



Global Opt.: Conservative

- Compiler must prove some property X at a particular point
  - Need to prove at that point property X holds along all paths.
  - Need to be **conservative**[保守性]to ensure correctness.
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say don't know
    - If analysis result is don't know, no optimization done
    - May lose optimization opportunities but guarantees correctness
- Property X often involves data flow of program
  - e.g., Global Constant Propagation (GCP)
  - Needs knowledge of data flow, as well as control flow
    - Whether data flow is interrupted between points A and B

Global Opt.: Data Flow

- Most optimizations rely on a property at given point, called **values**
  - ◆ For Global Constant Propagation (GCP):

    A = B + C; // Property: {A=?, B=10, C=?}

  - ◆ After optimization:

    A = 10 + C;

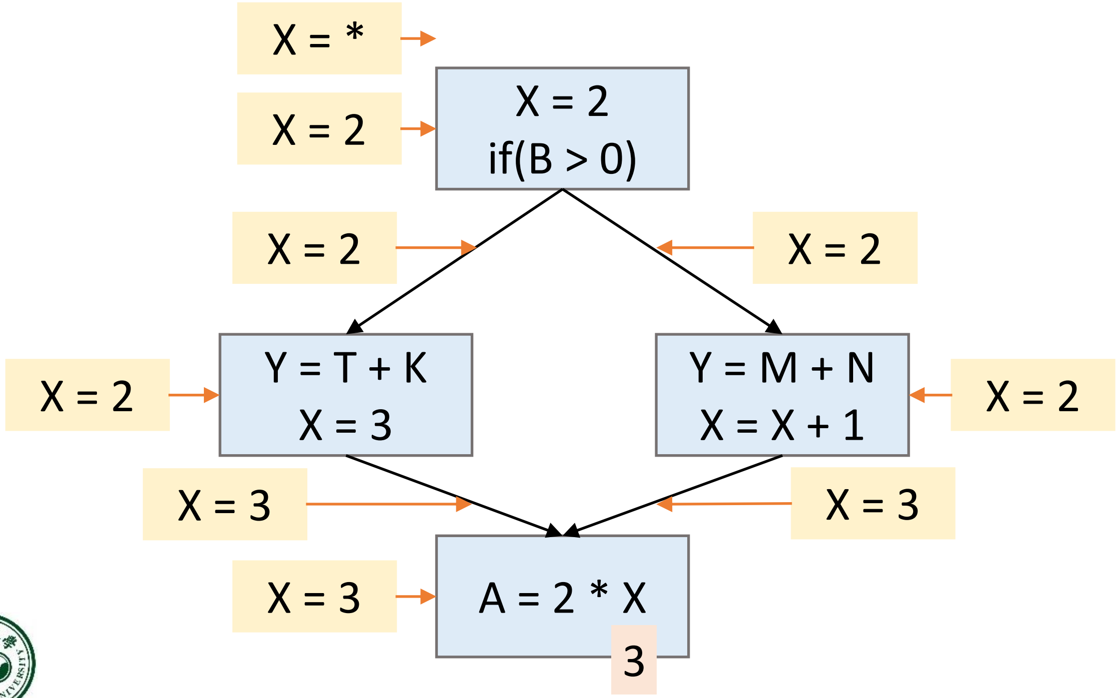
- **Dataflow analysis**[数据流分析]: compiler analysis that calculates values for each point in a program
  - ◆ Values get propagated from one statement to the next
  - ◆ Statements can modify values (for GCP, assigning to vars)
  - ◆ Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: a framework for dataflow analysis that guarantees correctness for all paths
  - ◆ To be feasible, makes **conservative** approximations

Global Constant Propagation

- Let's apply dataflow analysis to compute values for:
  - Emulates what human does when tracing through code
- Let's use following notation to express the state of a variable:
  - x=*: not assigned (default)
  - x=1, x=2, ...: assigned to a constant value
  - x=#: assigned to multiple values
- All values start as x=* and are iteratively refined
  - Until they stabilize and reach a fixed point
- Once fixed point is reached, can replace with constants:
  - x=*: replace with any constant (typically 0)
  - x=1, x=2, ...: replace with given constant value
  - x=#: cannot do anything

Example

- In this example, constants can be propagated to **X+1**, **2*X**
- Statements visited in reverse post-order (predecessor first)



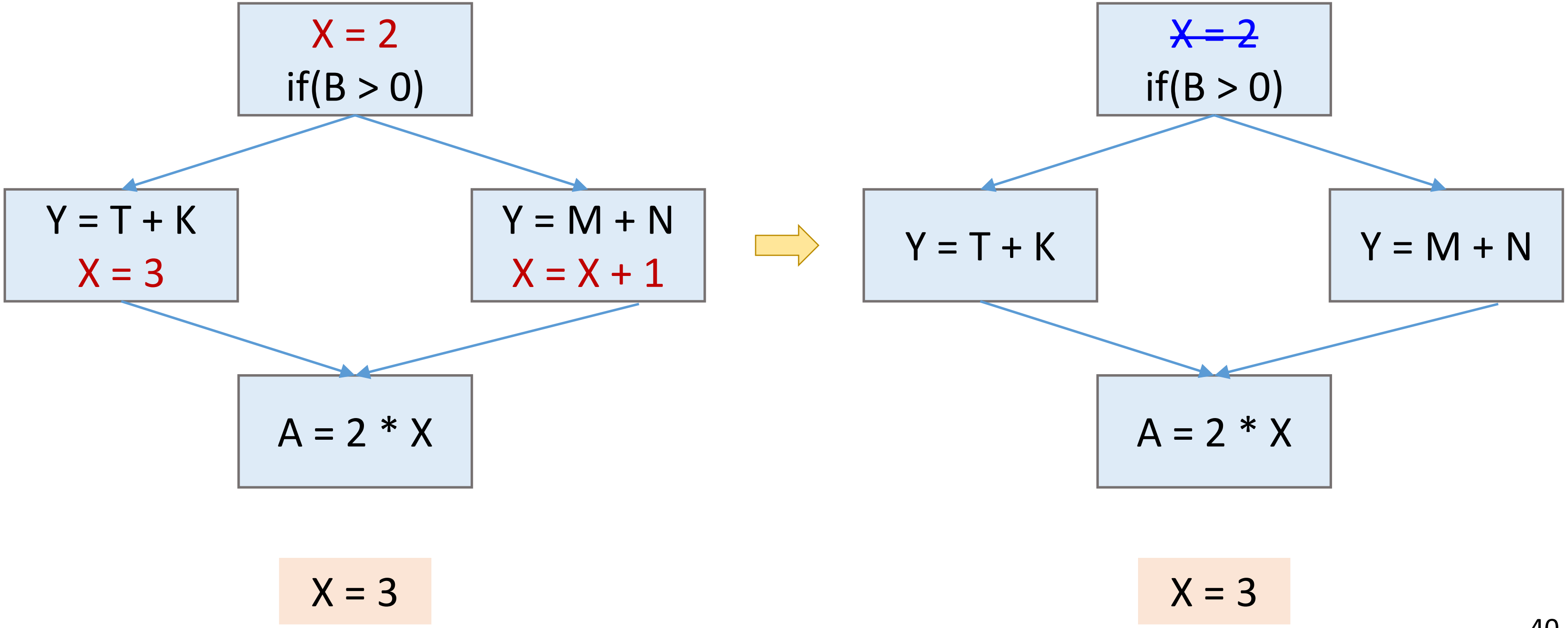
x=*: not assigned (default)

x=1, x=2, ...: assigned to a constant value

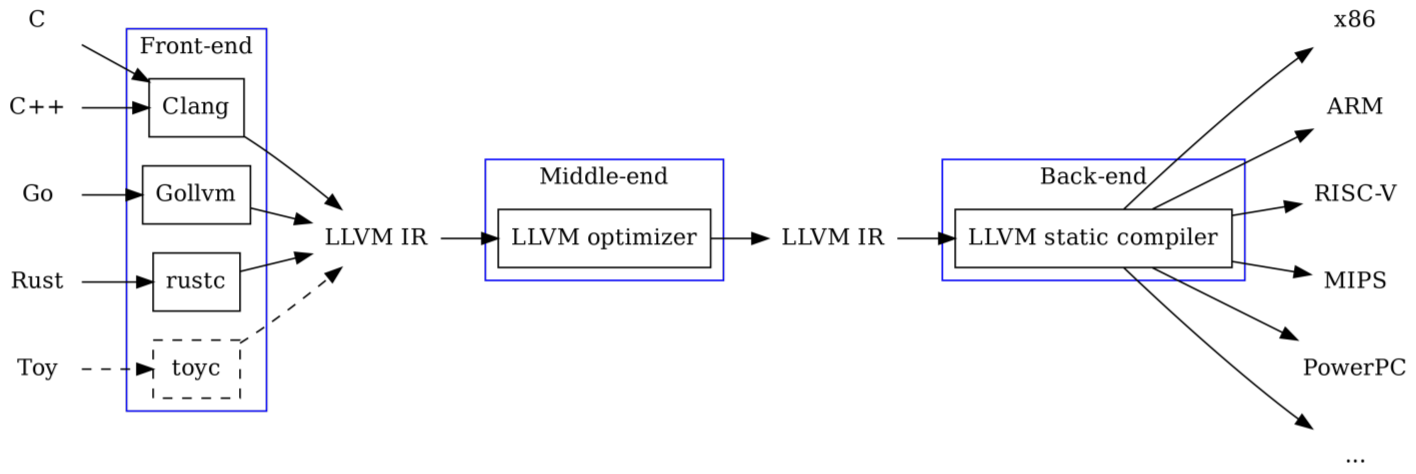
x=#: assigned to multiple values

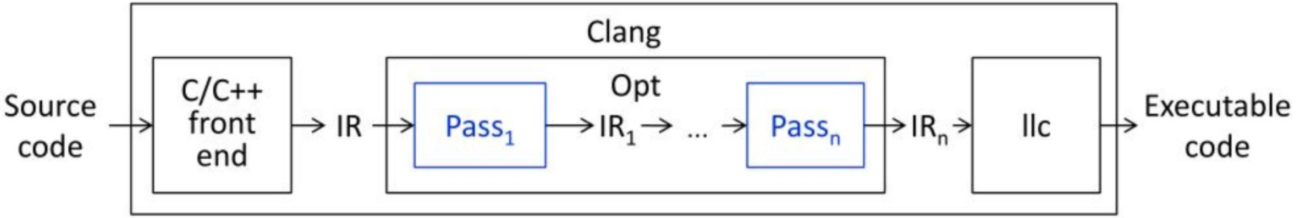
Example

- Once constants have been globally propagated, we would like to eliminate the dead code



IR Optimization of LLVM

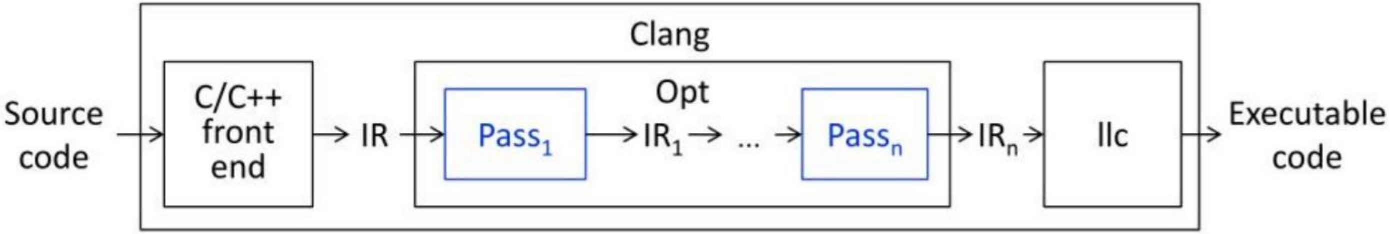




Source url: https://www.slideserve.com/quinlan-dominguez/llvm-pass-and-code-instrumentation

LLVM Passes

- Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program
- A Pass receives an LLVM IR and performs analyses and/or transformations
  - Using opt, it is possible to run each Pass
- A Pass can be executed in a middle of compiling process from source code to binary code
  - The pipeline of Passes is arranged by Pass Manager



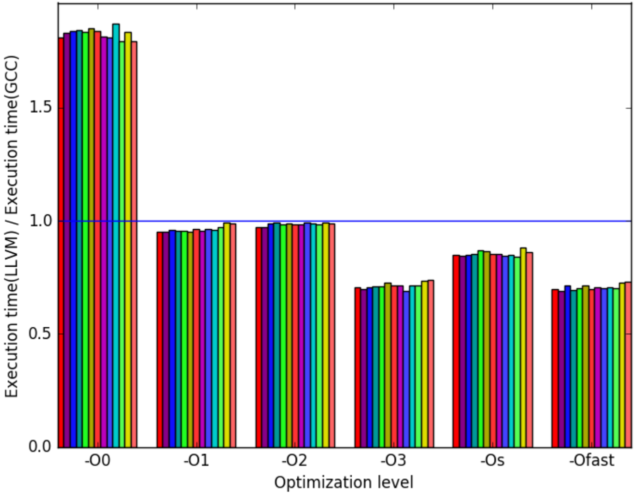
Source url: https://www.slideserve.com/quinlan-dominguez/llvm-pass-and-code-instrumentation

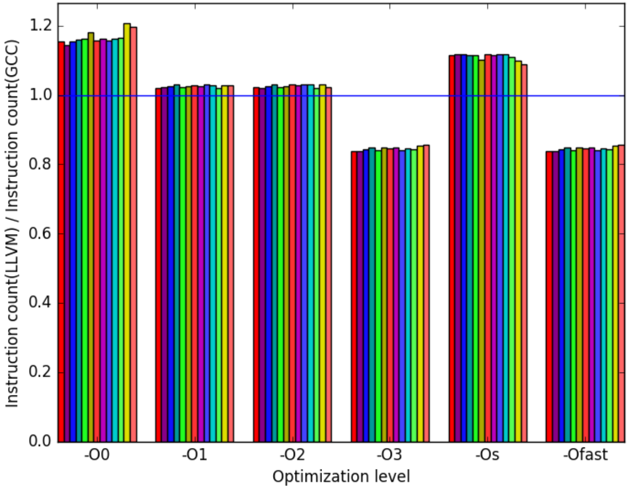
General Optimization Flags

- O0: no optimization
  - Compiles the fastest and generates the most debuggable code
- O1: somewhere between O0 and O2
- O2: moderate level of optimization enabling most optimizations
- O3: like O2,
  - except that it enables opts that take longer to perform or that may generate larger code (in an attempt to make the program run faster)
- Os: like O2 with extra opts to reduce code size
- Oz: like Os, but reduce code size further
- O4: enables link-time opt Clang has support for O4, but not opt

Performance at Varying Flags

- Compare the performance of the benchmark when compiled with either GCC or LLVM
  - ◆ Compile benchmark at six optimization levels
  - ◆ Each workload was run 3 times with each executable on the Intel Core i7-2600 machines





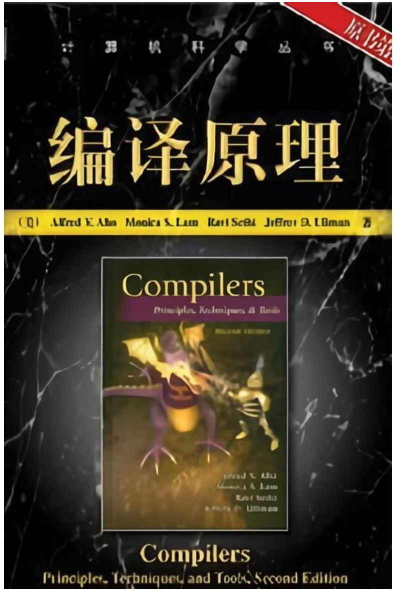
Source url: https://webdocs.cs.ualberta.ca/~amaral/AlbertaWorkloadsForSPECCPU2017/reports/exchange2_report.html

Summary

- **Layout-related** transformations[布局相关]
  - Goal: maximize the spatial locality [空间局部性].
    - code layout, declaration ordering, data structuring
- **Code-related** transformations[代码相关]
  - Goal: execute least number of most costly instructions.
  - Basic blocks
  - Control flow graphs
  - DAG construction of basic blocks
  - Local optimization
    - CSE/ DCE / Algebraic identities / Constant folding and propagation
  - Global Optimization
    - Constant propagation

Further Reading

- Dragon Book, 2nd Edition
  - ◆ Comprehensive Reading:
    - Section 8.4 8.5 on DAG based block optimization.
    - Section 9.1 on an example of loop optimization.
    - Section 9.6.1, 9.6.6 on basic concepts of loop optimization.
    - Section 9.2.1 9.2.4 on data flow equations and reaching definition analysis.
  - ◆ Skip Reading:
    - Section 9.2.5-9.2.6 on liveness and available expression analysis.
    - Section 9.6.4 on properties of reducible flow graphs.



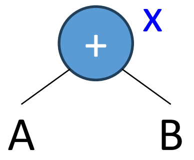
DAG Construction Revisit

- Three possible scenarios
  1. x = y op z
  2. x = op y
  3. x = y
- Step 1:
  - If the y operand is not defined, then create a node (y).
  - For case 1 - If the z operand is not defined, create a node(z).
- Step 2:
  - For case 1 - Create node(OP), with node(y) as its left child and node(z) as its right.
  - For case 2 - See if there is a node operator (OP) with one child node (y).
  - For case 3 - Node node(x) will be node(y).
- Step 3:
  - Remove x from the list of node identifiers
  - Add x to the list of attached identifiers for node n.

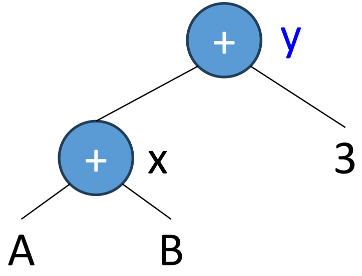
https://www.geeksforgeeks.org/directed-acyclic-graph-in-compiler-design-with-examples/

Example

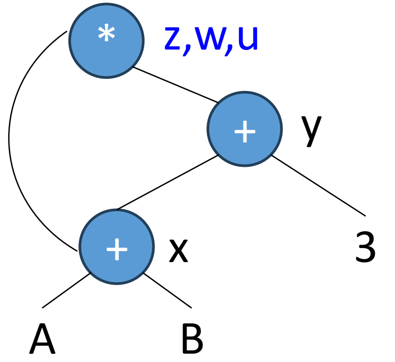
- x = A + B



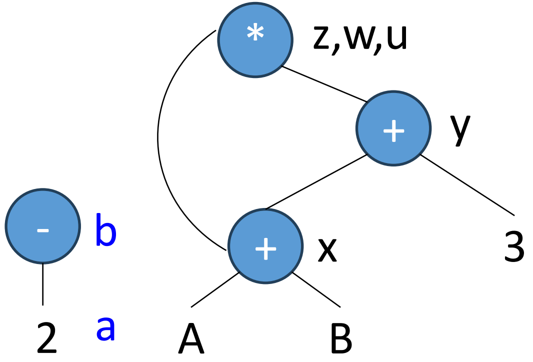
- y = x + 3



- z = x * y
- w = x * y
- u = w

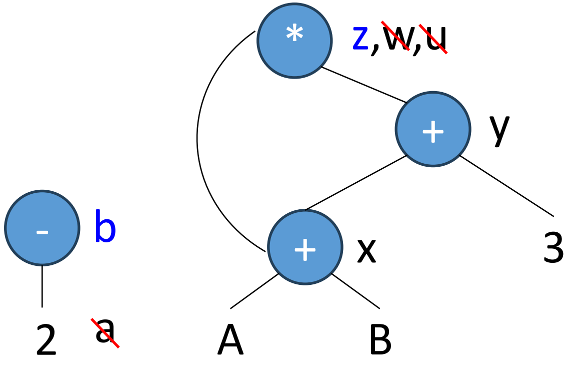


- a = 4 / 2
- b = - a



DAG-based Optimization

- If z and b are alive



x = A + B

y = x + 3

z = x * y

~~w = x * y~~

~~u = w~~

~~a = 4 / 2~~

b = - 2