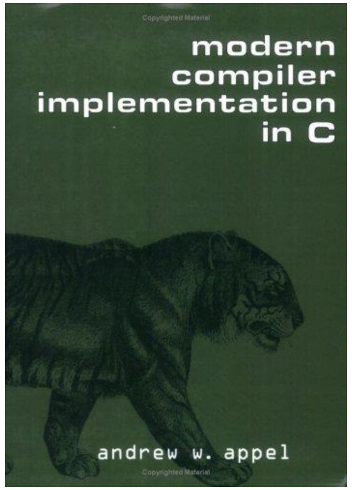
Crash course on optimizing compilers

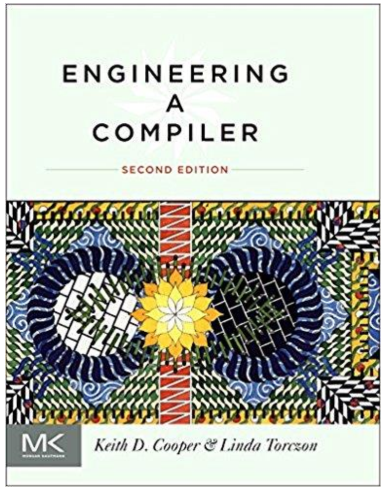
Areg Melik-Adamyan, PhD

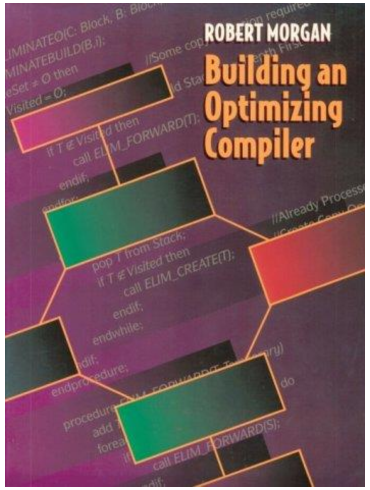
Engineering Manager, Intel Developer Products Division

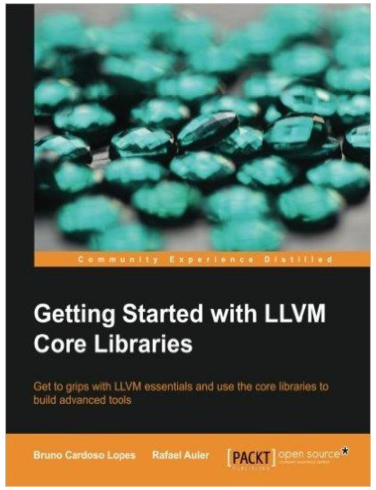
Textbooks and References

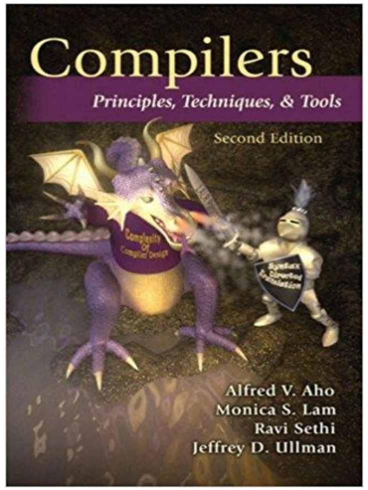
- Again hitting only the tip of the iceberg
- Explain main concepts only
- 40 years of research
- But allow better understand how modern compilers are constructed and work. And what are the implications



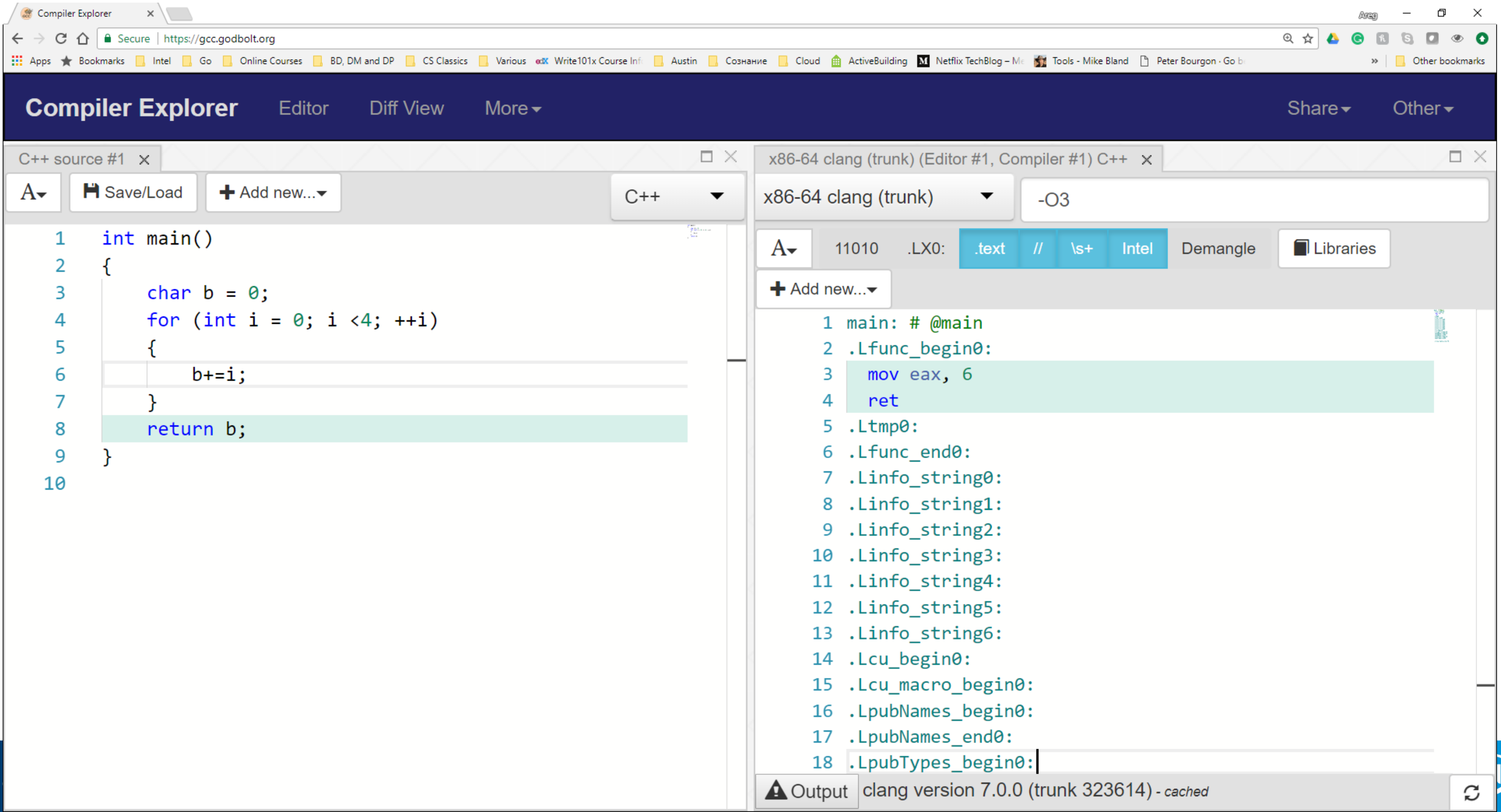








Compiler Exploration Tool



Role of compilers

Bridge complexity and evolution in architecture, languages, & applications

Help programs with correctness, reliability, program understanding

Compiler optimizations can significantly improve performance

- 1 to 10x on conventional processors

Performance stability: one line change can dramatically alter performance

- unfortunate, but true

Performance Anxiety

But does performance **really** matter?

- Computers are *really* fast
- **Moore's law**

Real bottlenecks are elsewhere (Vtune will help):

- Disk
- Network
- Human!

Compilers Don't Help Much

Do compilers improve performance anyway?

- **Proebsting's law**
  (Todd Proebsting, Microsoft Research):
  - Difference between optimizing and non-optimizing compiler **in average** ~ 4x
  - Assume compiler technology represents 36 years of progress (actually more)

⇨ **Compilers double program performance every 18 years!**

⇨ Not quite Moore's Law…

A Big BUT

Why use high-level languages anyway?

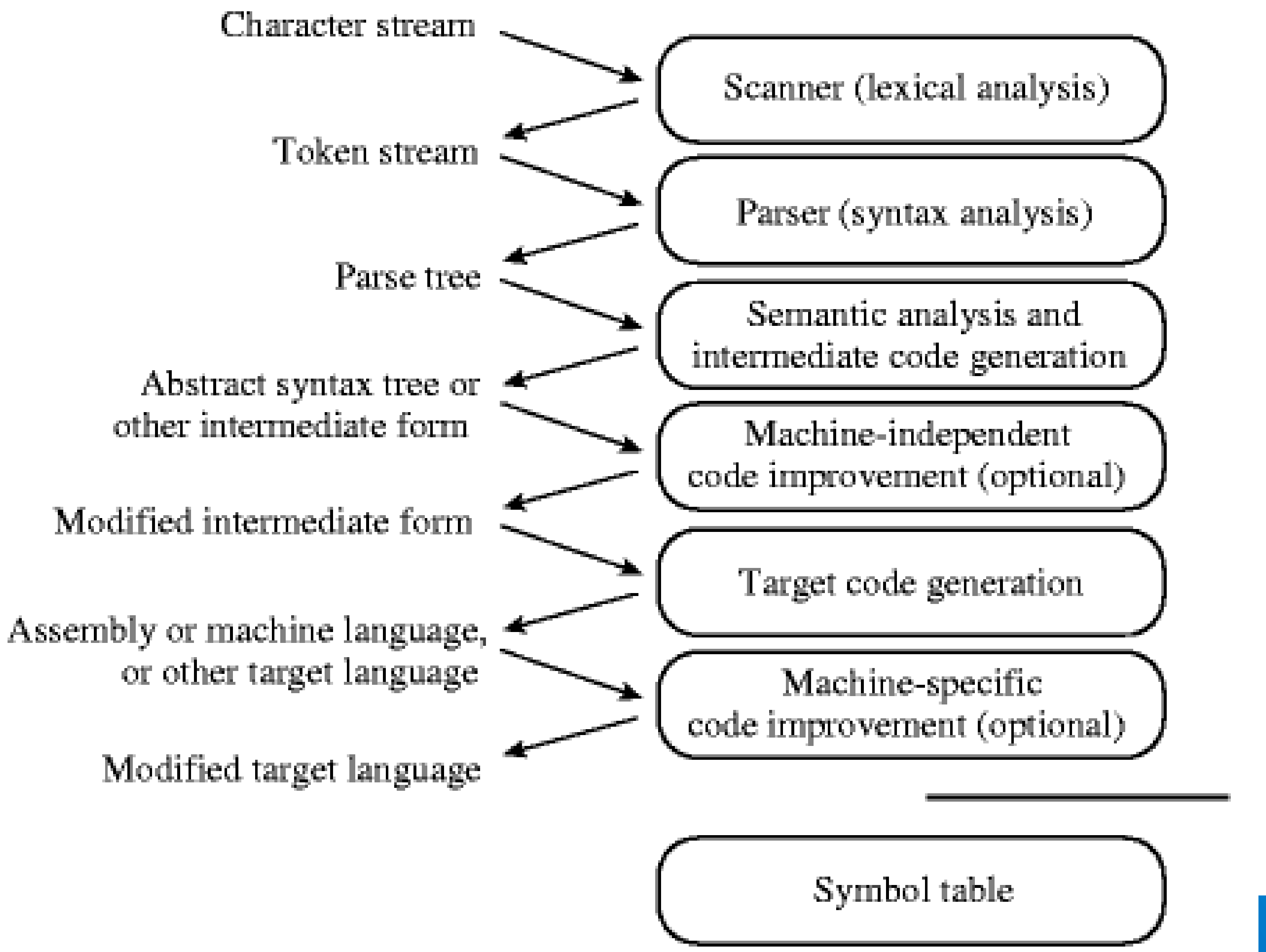
- Easier to write & maintain
- Safer (think Java)
- More convenient (higher level abstractions, libraries, GC…)

But: people will *not* accept massive performance hit for these gains

- Compile with optimization!
- Still use C and C++!!
- Hand-optimize their code!!!
- *Even* write assembler code (gasp)!!!!

Apparently performance *does* matter… Now even more than before…

Phases of Compilation



Scanning/Lexical analysis

Break program down into its smallest meaningful symbols (tokens, atoms)

Tools for this include lex, flex

Tokens include e.g.:

- “Reserved words” : do if float while
- Special characters: ({ , + - = ! /
- Names & numbers: myValue 3.07e02

Start symbol table with new symbols found

Parsing

Construct a parse tree from symbols

A pattern-matching problem

- Language **grammar** defined by set of rules that identify legal (meaningful) combinations of symbols
- Each application of a rule results in a node in the parse tree
- Parser applies these rules repeatedly to the program until leaves of parse tree are "atoms"

If no pattern matches, it's a syntax error

yacc, bison, etc are tools for this (generate c code that parses specified language)

Parse tree

Output of parsing

Top-down description of program syntax

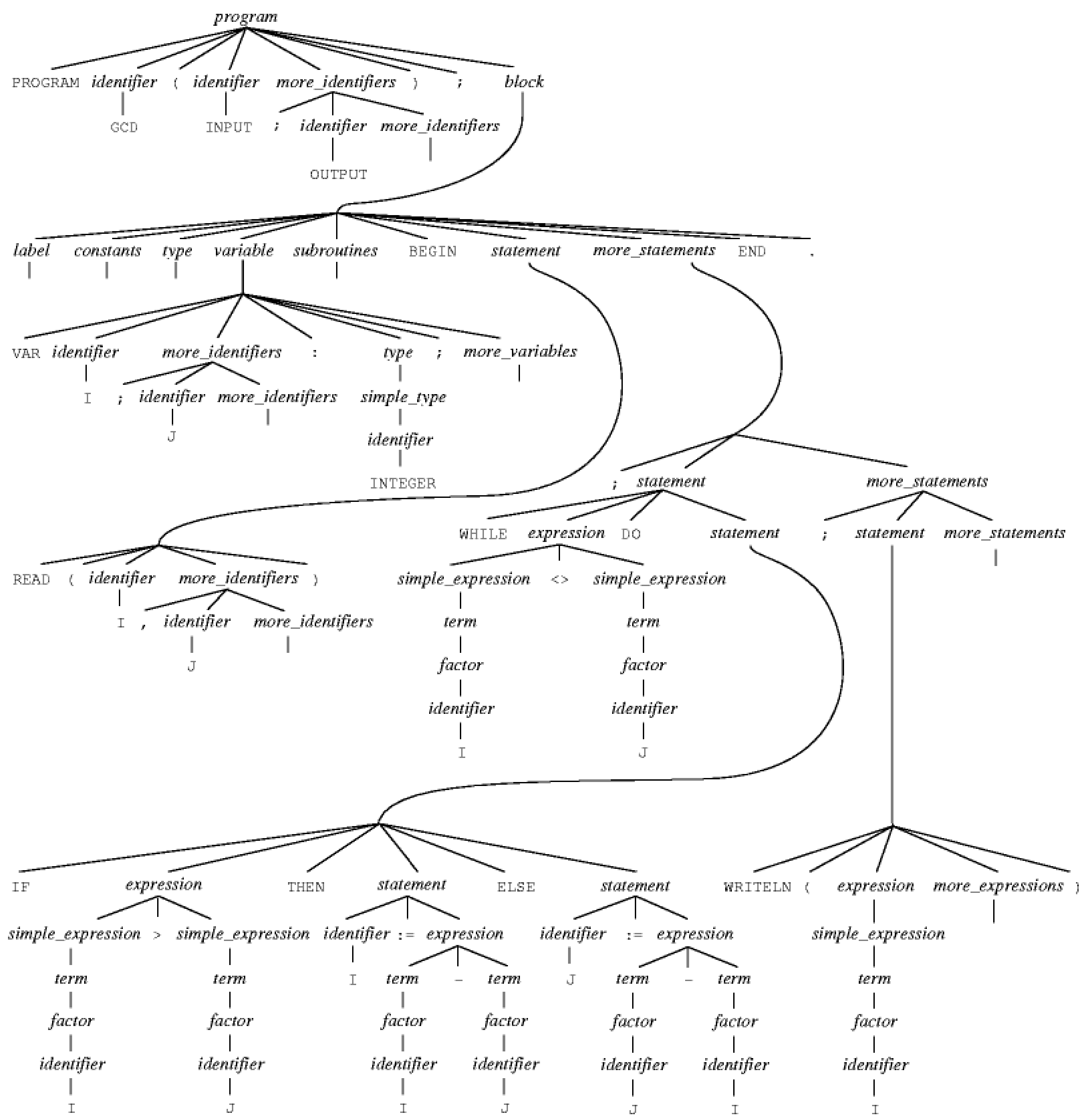
- Root node is entire program

Constructed by repeated application of rules in Context Free Grammar (CFG)

Leaves are tokens that were identified during lexical analysis

Example: parse tree

```
program gcd (input, output)
var i, j : integer
begin
    read (i , j)
    while i <> j do
        if i>j then i := i  -  j;
        else j := j  -  i ;
    writeln (i);
end .
```



Semantic analysis

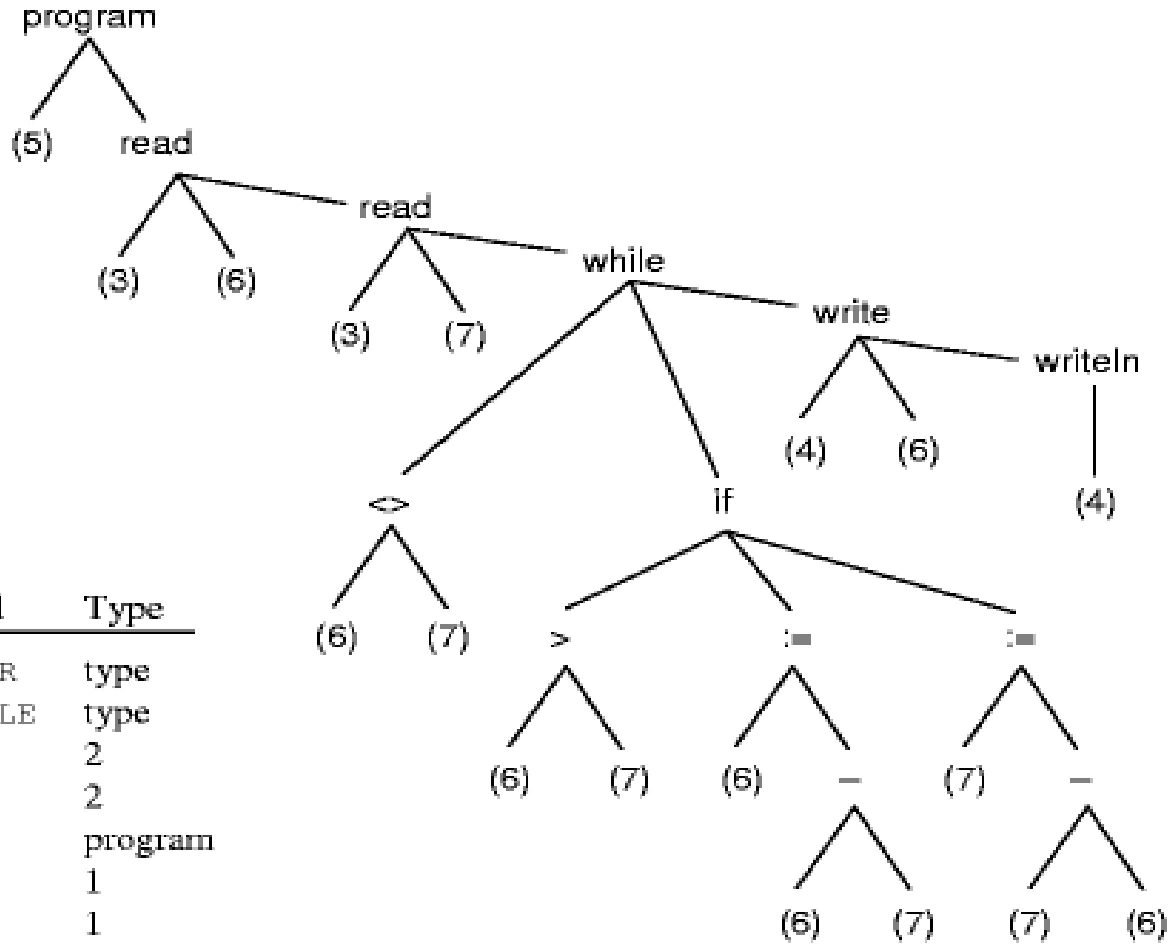
Discovery of meaning in a program using the symbol table

- Do static semantics check
- Simplify the structure of the parse tree (from parse tree to abstract syntax tree (AST))

Static semantics check

- Making sure identifiers are declared before use
- Type checking for assignments and operators
- Checking types and number of parameters to subroutines
- Making sure functions contain return statements
- Making sure there are no repeats among switch statement labels

Example: AST



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

The 80/20 Rule

In general, *80% percent of a program's execution time is spent executing 20% of the code*

90%/10% for performance-hungry programs

Spend your time optimizing the important 10/20% of your program

Optimize the common case even at the cost of making the uncommon case slower

The Golden Rules of Optimization

The best and most important way of optimizing a program is using **good algorithms**

- E.g. O(n*log) rather than $O(n^2)$

However, we still need lower level optimization to get more of our programs

In addition, asymptotic complexity is not always an appropriate metric of efficiency

- Hidden constant may be misleading
- *E.g.* a linear time algorithm than runs in $100*n+100$ time is slower than a cubic time algorithm than runs in n^3+10 time <u>if the problem size is small</u>

General Optimization Techniques

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- Different levels of optimization to achieve different goals: O0, O1, O2, O3, Os,··· More than 200 optimizations

Strength reduction

- Use the fastest version of an operation
- *E.g.*

```
x >> 2        instead of    x / 4
x << 1        instead of    x * 2
```

Common sub expression elimination

- Eliminate redundant calculations
- *E.g.*

```
double x = d * (lim / max) * sx;
double y = d * (lim / max) * sy;

double depth = d * (lim / max);
double x = depth * sx;
double y = depth * sy;
```

General Optimization Techniques

Code motion

- *Invariant* expressions should be executed only once
- *E.g.*

```
for (int i = 0; i < x.length; i++)
    x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;
```

General Optimization Techniques

Loop unrolling

- The overhead of the loop control code can be reduced by executing more than one iteration in the body of the loop. *E.g.*

```
double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i++)
    x[i] *= picosy;

double picosy = Math.PI * Math.cos(y);
for (int i = 0; i < x.length; i += 2) {
    x[i] *= picosy;
    x[i+1] *= picosy;
}
```

A efficient "+1" in array indexing is required

Optimization Notice

Compiler Optimizations

Compilers try to generate *good* code

- *i.e.* Fast

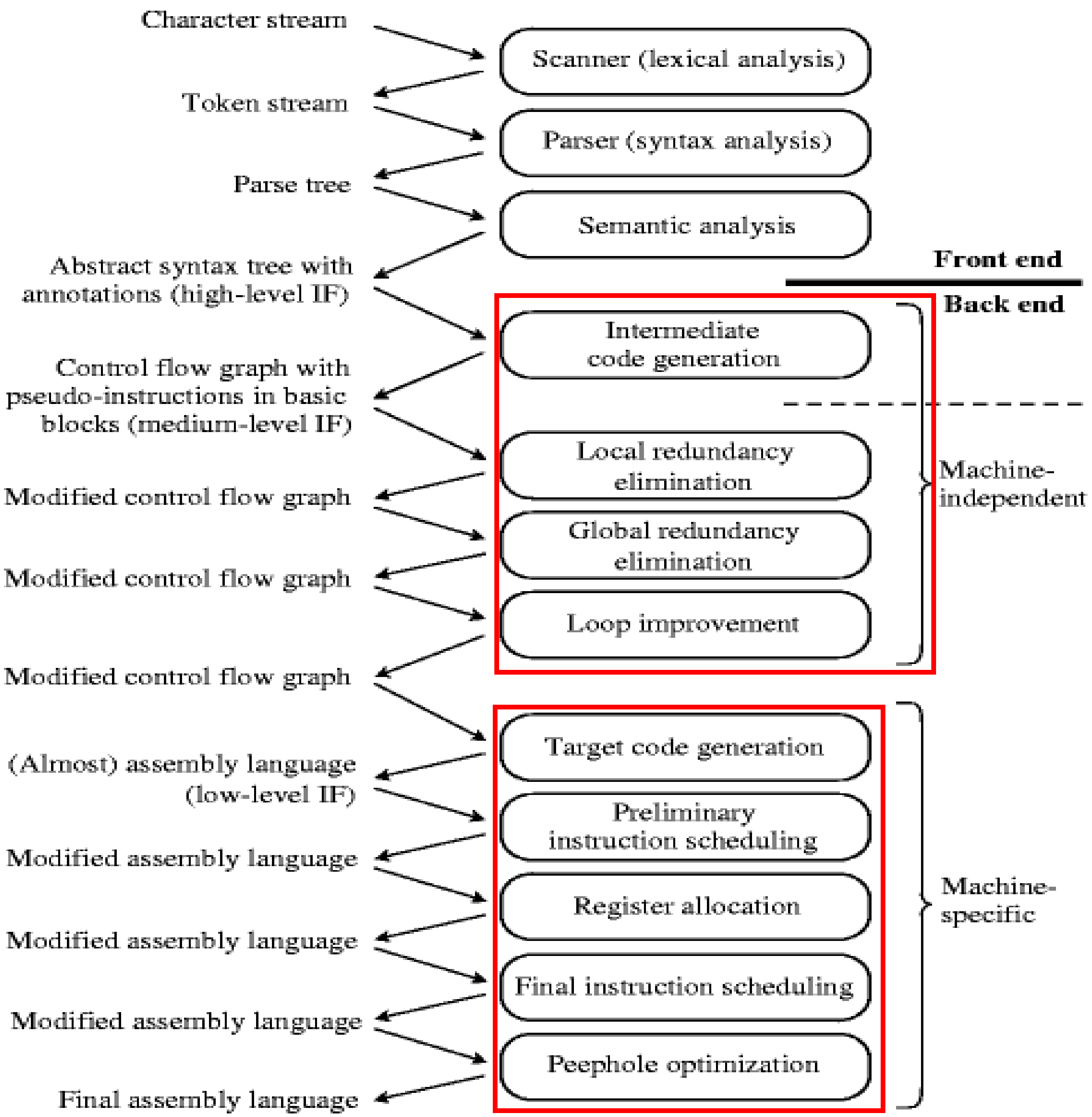
Code improvement is challenging

- Many problems are NP-hard

Code improvement may slow down the compilation process

- In some domains, such as just-in-time compilation, compilation speed is critical

Backend Phases



Basic Blocks

A **basic block** is a maximal sequence of consecutive instructions with the following properties:

- The flow of control can only enter the basic block thru the 1st instr.
- Control will leave the block without halting or branching, except possibly at the last instr.

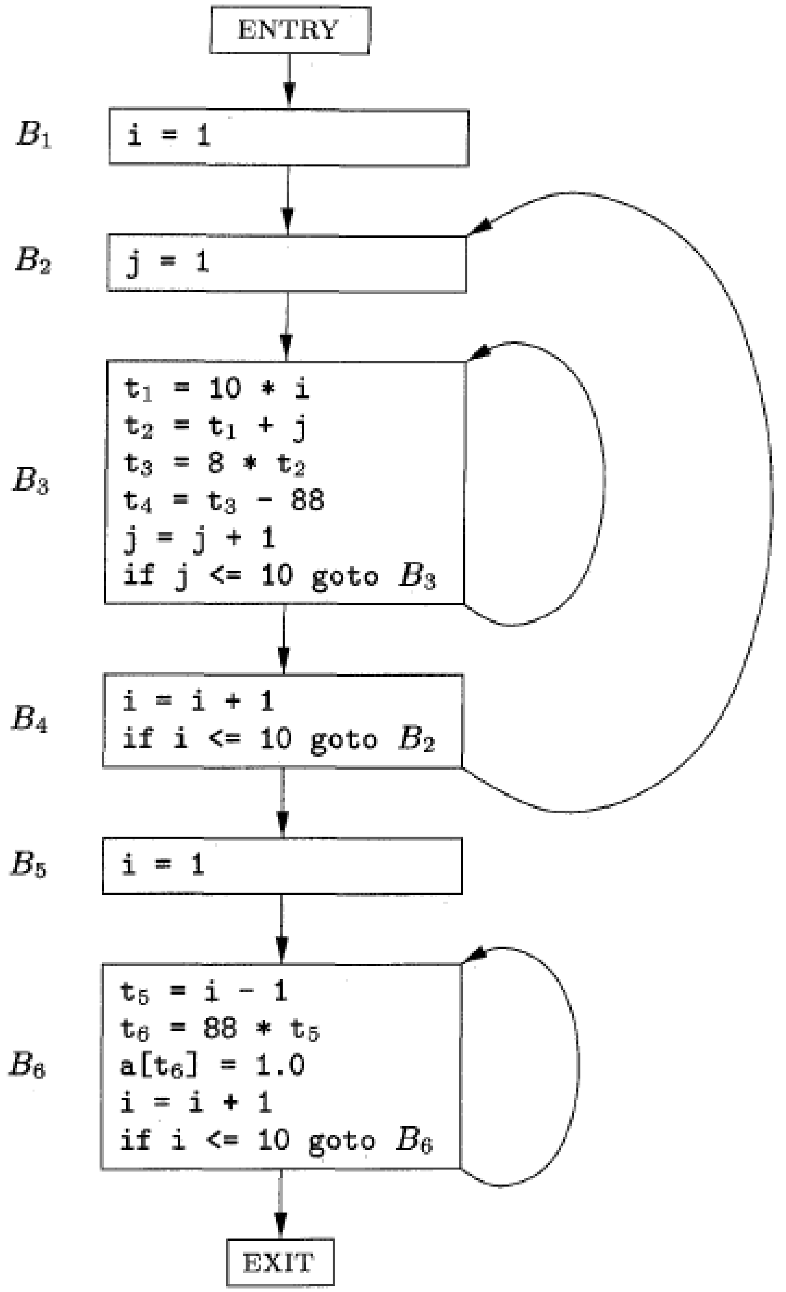
Basic blocks become the nodes of a **flow graph**, with edges indicating the order.

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Control-Flow Graphs

Control-flow graph:

- Node: an instruction or sequence of instructions (a basic block)
  - Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
- Directed edge: *potential* flow of control
- Distinguished start node *Entry & Exit*
  - First & last instruction in program



Transformations on basic blocks

Common subexpression elimination: recognize redundant computations, replace with single temporary

Dead-code elimination: recognize computations not used subsequently, remove quadruples

Interchange statements, for better scheduling

Renaming of temporaries, for better register usage

All of the above require **symbolic execution** of the basic block, to obtain definition/use information

Computing dependencies in a basic block: the DAG

Use directed acyclic graph (DAG) to recognize common subexpressions and remove redundant quadruples.

Intermediate code optimization:

- basic block => DAG => improved block => assembly

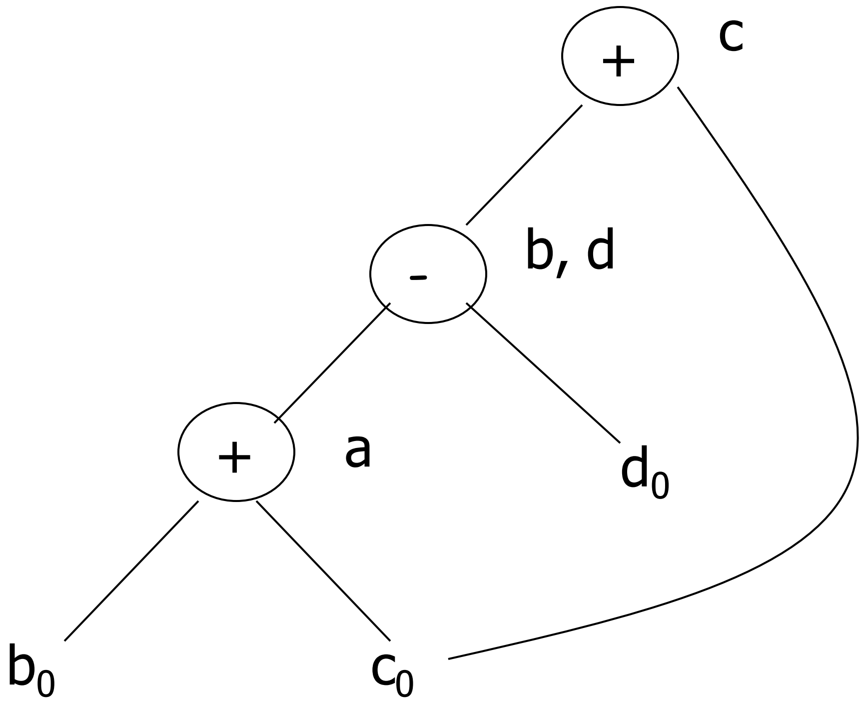
Leaves are labeled with identifiers and constants.

Internal nodes are labeled with operators and identifiers

DAG Example

Transform a basic block into a DAG.

```
a = b + c
b = a - d
c = b + c
d = a - d
```



SSA: Motivation

SSA (Static Single-Assignment): A program is said to be in SSA form iff

- Each variable is statically defined exactly only once, and
- each use of a variable is dominated by that variable's definition.

Provide a uniform basis of an IR to solve a wide range of classical dataflow problems

Encode both dataflow and control flow information

A SSA form can be constructed and maintained efficiently

Many SSA dataflow analysis algorithms are more efficient (have lower complexity) than their CFG counterparts.

Static Single-Assignment Form

Each variable has only one definition in the program text.

This single *static* definition can be in a loop and may be executed many times. Thus even in a program expressed in SSA, a variable can be dynamically defined many times.

Advantages of SSA

Simpler dataflow analysis

No need to use use-def/def-use chains, which requires N×M space for N uses and M definitions

SSA form relates in a useful way with dominance structures.

Differentiate unrelated uses of the same variable

- E.g. loop induction variables

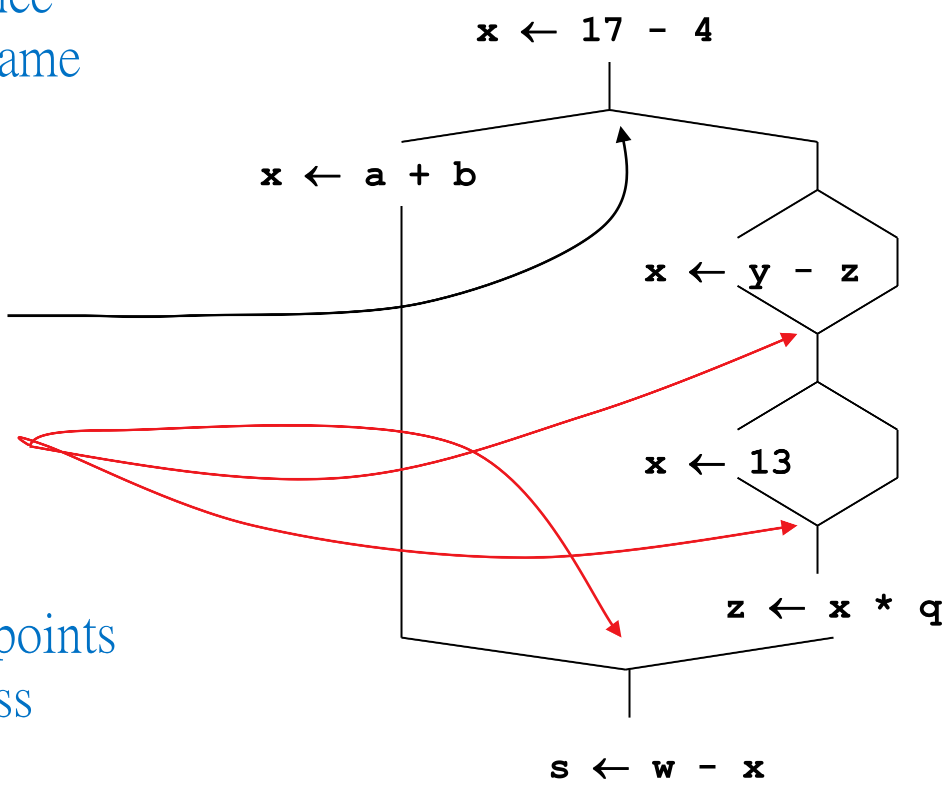
SSA Form – An Example

SSA-form
- Each name is defined exactly once
- Each use refers to exactly one name

What's hard
- Straight-line code is trivial
- Splits in the CFG are trivial

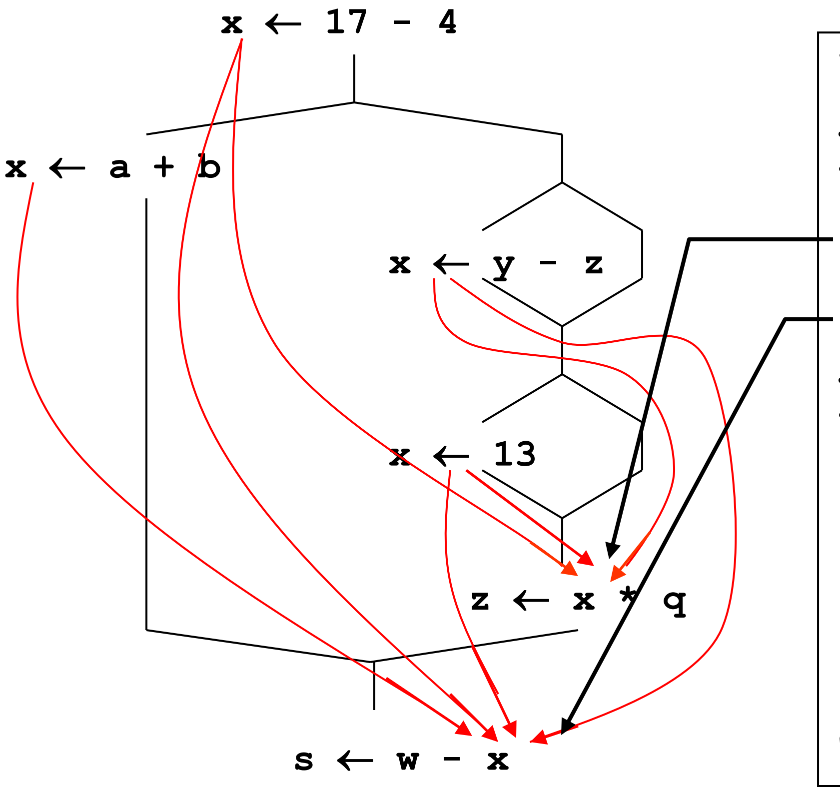
- Joins in the CFG are hard

Building SSA Form
- Insert Φ(phi)-functions at birth points
- *Rename* all values for uniqueness



Birth Points

Consider the flow of values in this example:



The value x appears everywhere

It takes on several values.

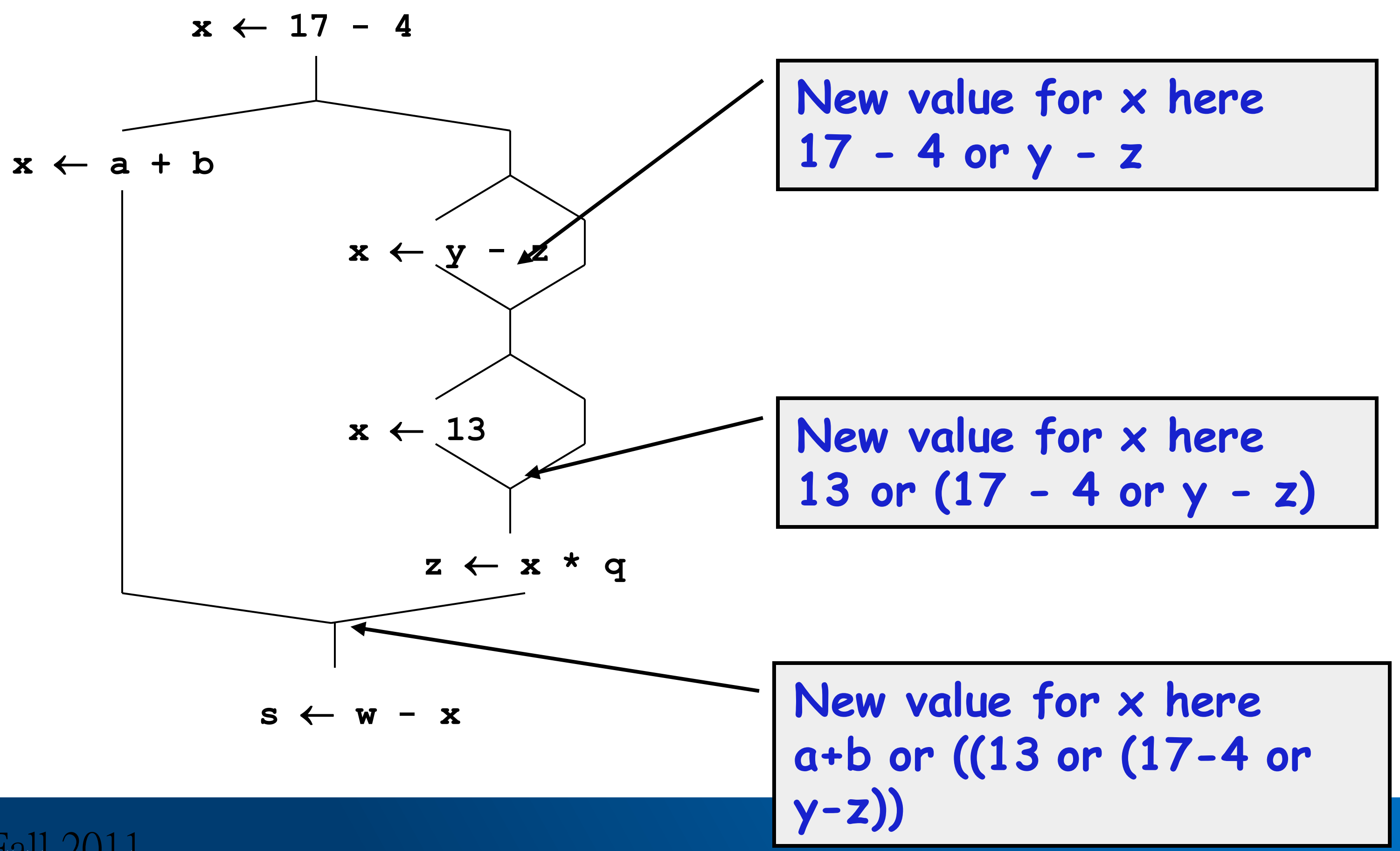
- Here, x can be 13, y-z, or 17-4
- Here, it can also be a+b

If each value has its own name …

- Need a way to merge these distinct values
- Values are "born" at merge points

Birth Points

Consider the flow of values in this example:



Review

SSA-form

Each name is defined exactly once

Each use refers to exactly one name

What's hard

Straight-line code is trivial

Splits in the CFG are trivial

Joins in the CFG are hard

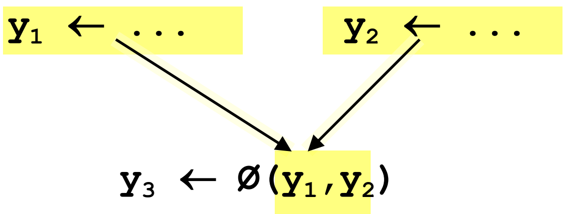
Building SSA Form

Insert Ø-functions at birth points

Rename all values for uniqueness

A Ø-function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.



Real machines do not implement a Ø-function directly in hardware (not yet!)

LLVM Compiler System

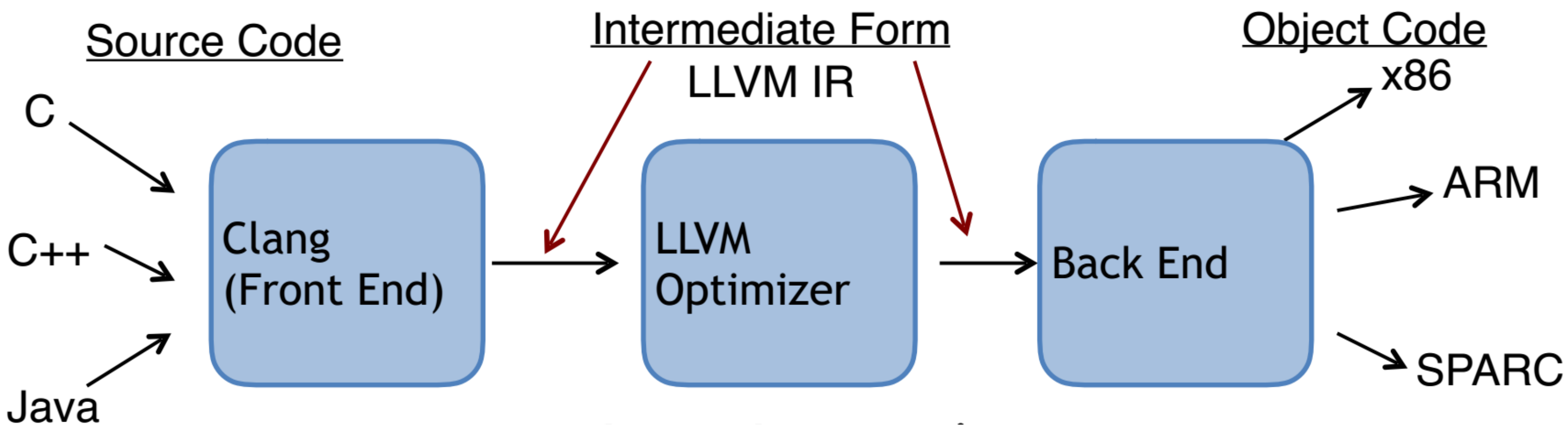
The LLVM Compiler Infrastructure

- Provides reusable components for building compilers
- Reduce the time/cost to build a new compiler
- Build different kinds of compilers
- Our homework assignments focus on static compilers
- There are also JITs, trace-based optimizers, etc.

The LLVM Compiler Framework

- End-to-end compilers using the LLVM infrastructure
- Support for C and C++ is robust and aggressive
- Java, Scheme and others are in development
- Emit C code or native code for x86, SPARC, PowerPC

Components of LLVM



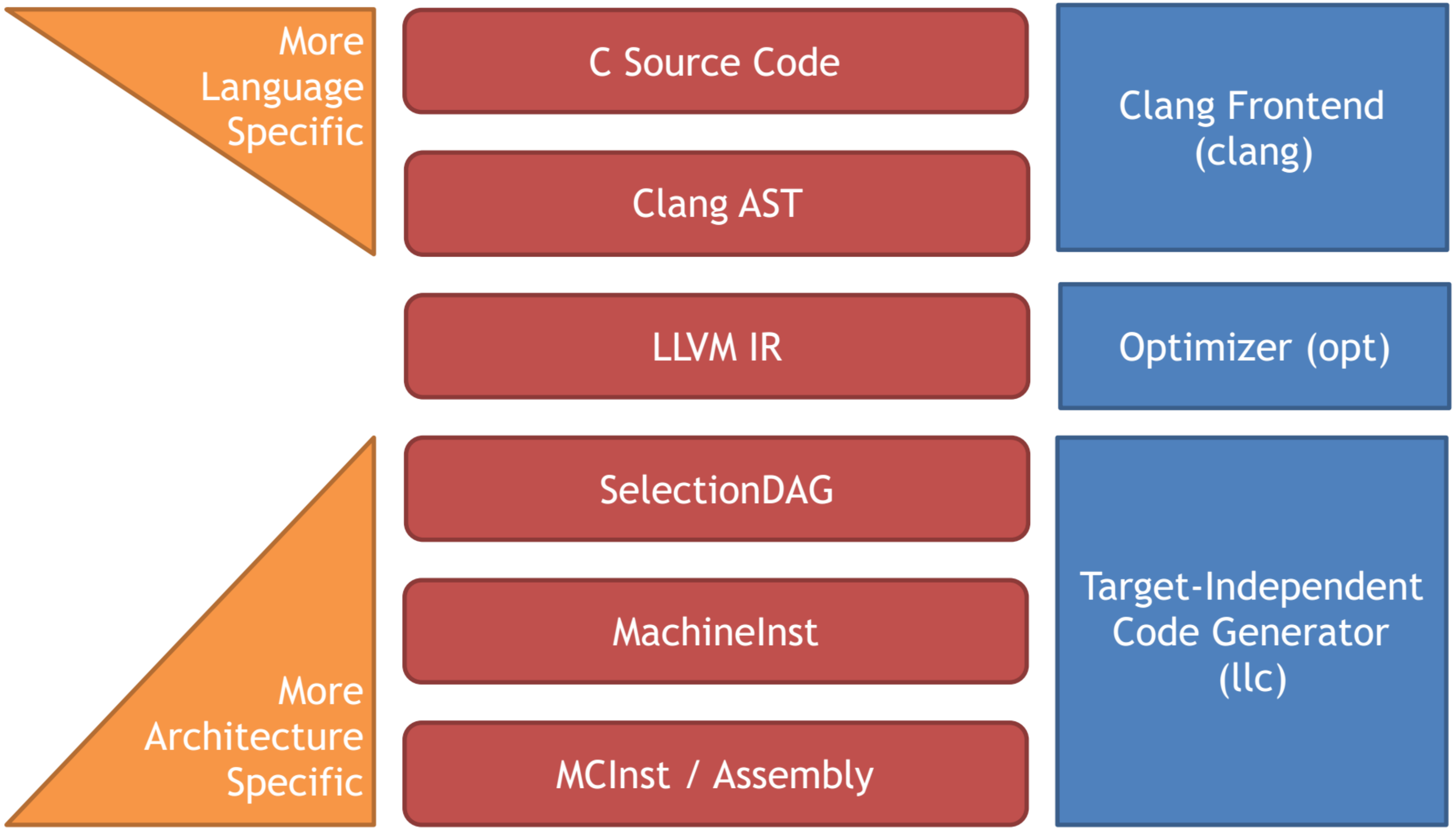
The LLVM Optimizer is a series of "passes"

- Analysis and optimization passes, run one after another
- *Analysis* passes do not change code, *optimization* passes do

LLVM Intermediate Form is a *Virtual Instruction Set*

- Language- and target-independent form: used to perform the same passes for all source and target languages
- Internal Representation (IR) and external (persistent) representation

LLVM Diagram



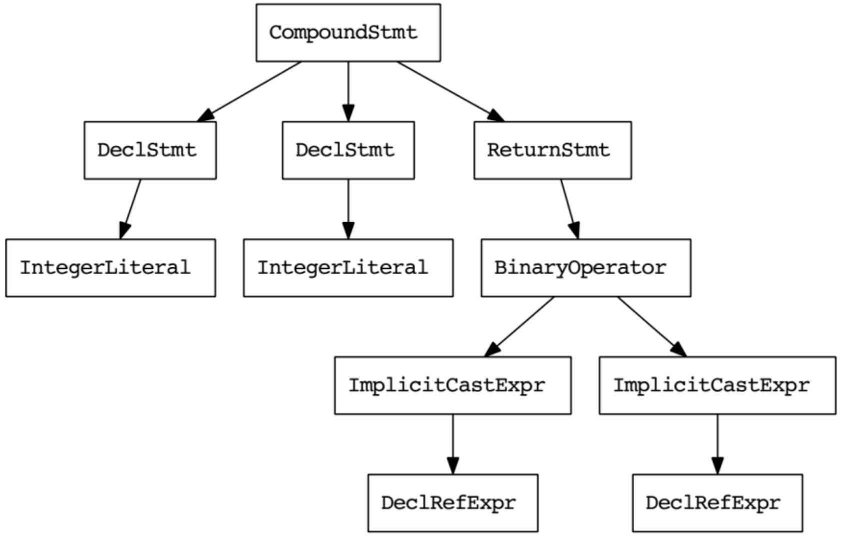
LLVM Code Transformation Path

Read "Life of an instruction in LLVM" : http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

```
int main() {
    int a = 5;
    int b = 3;
    return a - b;
}
```

Clang AST

```
TranslationUnitDecl 0xd8185a0 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xd818870 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list
'char *'
`-FunctionDecl 0xd8188e0 <example.c:1:1, line:5:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xd818a90 <col:12, line:5:1>
    |-DeclStmt 0xd818998 <line:2:5, col:14>
    | `-VarDecl 0xd818950 <col:5, col:13> col:9 used a 'int' cinit
    |   `-IntegerLiteral 0xd818980 <col:13> 'int' 5
    |-DeclStmt 0xd818a08 <line:3:5, col:14>
    | `-VarDecl 0xd8189c0 <col:5, col:13> col:9 used b 'int' cinit
    |   `-IntegerLiteral 0xd8189f0 <col:13> 'int' 3
    `-ReturnStmt 0xd818a80 <line:4:5, col:16>
      `-BinaryOperator 0xd818a68 <col:12, col:16> 'int' '-'
        |-ImplicitCastExpr 0xd818a48 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xd818a18 <col:12> 'int' lvalue Var 0xd818950 'a' 'int'
        `-ImplicitCastExpr 0xd818a58 <col:16> 'int' <LValueToRValue>
          `-DeclRefExpr 0xd818a30 <col:16> 'int' lvalue Var 0xd8189c0 'b' 'int'
```



LLVM IR Intermediate Representation

In-Memory Data Structure

Bitcode (.bc files)

```
42 43 C0 DE  21 0C 00 00
06 10 32 39  92 01 84 0C
0A 32 44 24  48 0A 90 21
18 00 00 00  98 00 00 00
E6 C6 21 1D  E6 A1 1C DA
…
```

llvm-dis

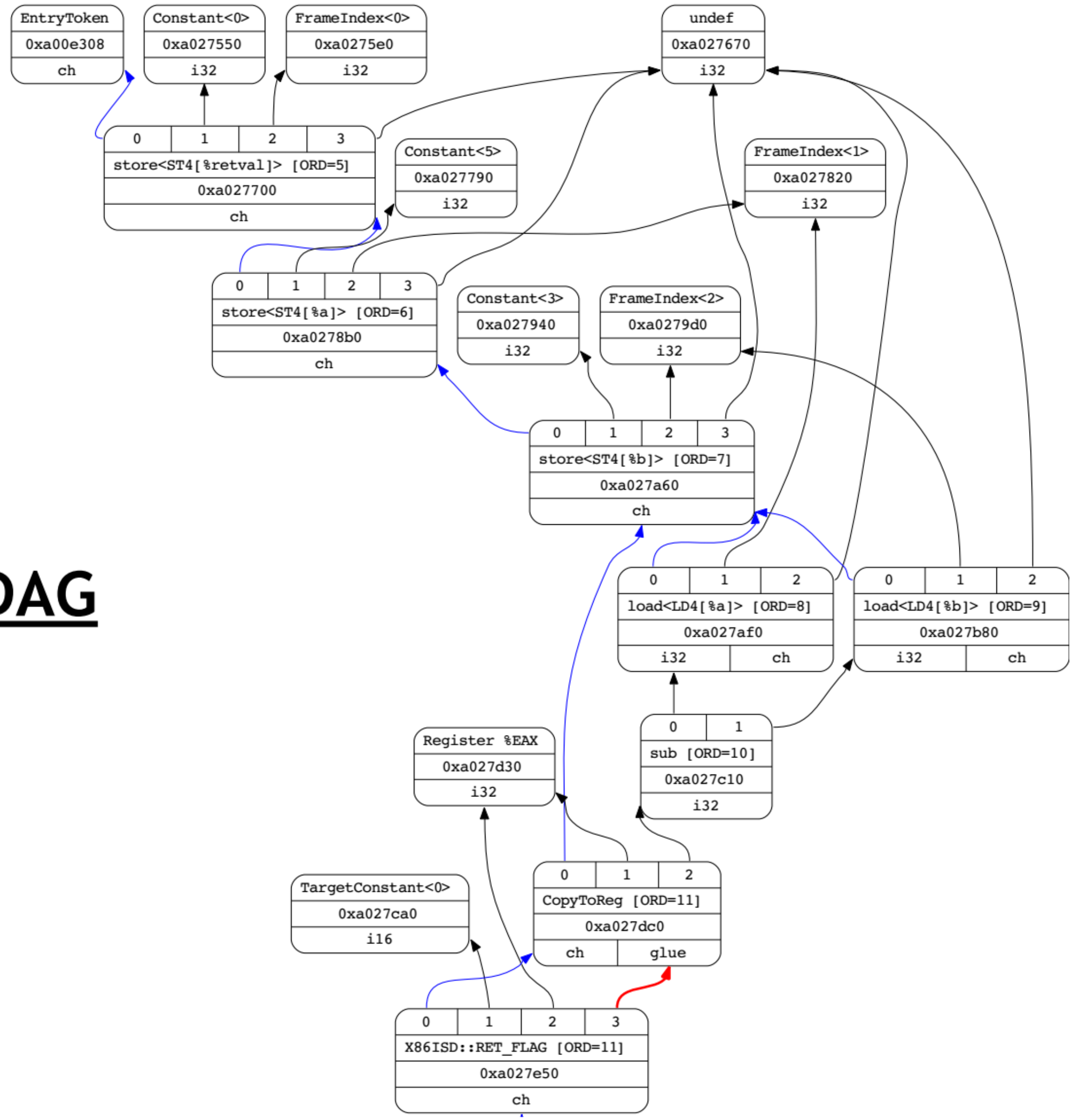
llvm-asm

Text Format (.ll files)

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
...
```

Bitcode files and LLVM IR text files are **lossless serialization formats!**
We can pause optimization and come back later.

SelectionDAG



MachineInst Target Machine Instruction Generation

```
BB#0: derived from LLVM BB %entry
    Live Ins: %EBP
            PUSH32r %EBP<kill>, %ESP<imp-def>, %ESP<imp-use>; flags: FrameSetup
            %EBP<def> = MOV32rr %ESP; flags: FrameSetup
            %ESP<def,tied1> = SUB32ri8 %ESP<tied0>, 12, %EFLAGS<imp-def,dead>; flags:
FrameSetup
            MOV32mi %EBP, 1, %noreg, -4, %noreg, 0; mem:ST4[%retval]
            MOV32mi %EBP, 1, %noreg, -8, %noreg, 5; mem:ST4[%a]
            MOV32mi %EBP, 1, %noreg, -12, %noreg, 3; mem:ST4[%b]
            %EAX<def> = MOV32rm %EBP, 1, %noreg, -8, %noreg; mem:LD4[%a]
            %EAX<def,tied1> = ADD32ri8 %EAX<kill,tied0>, -3, %EFLAGS<imp-def,dead>
            %ESP<def,tied1> = ADD32ri8 %ESP<tied0>, 12, %EFLAGS<imp-def,dead>
            %EBP<def> = POP32r %ESP<imp-def>, %ESP<imp-use>
            RETL %EAX
```

McInst Pass and Assembly

MCInst

```
#BB#0:                              # %entry
pushl   %ebp                  # <MCInst #2191 PUSH32r
                                    #  <MCOperand Reg:20>>
movl    %esp, %ebp                  # <MCInst #1566 MOV32rr
                                    #  <MCOperand Reg:20>
                                    #  <MCOperand Reg:30>>
subl    $12, %esp                   # <MCInst #2685 SUB32ri8
                                    #  <MCOperand Reg:30>
                                    #  <MCOperand Reg:30>
                                    #  <MCOperand Imm:12>>
movl    $0, -4(%ebp)          # <MCInst #1554 MOV32mi
                                    #  <MCOperand Reg:20>
                                    #  <MCOperand Imm:1>
                              #  <MCOperand Reg:0>
                                    #  <MCOperand Imm:-4>
                                    #  <MCOperand Reg:0>
                                    #  <MCOperand Imm:0>>
                    ….
```

```
main:                                   # @main
# BB#0:                                 # %entry
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
        movl    $0, -4(%ebp)
        movl    $5, -8(%ebp)
        movl    $3, -12(%ebp)
        movl    -8(%ebp), %eax
        addl    $-3, %eax
        addl    $12, %esp
        popl    %ebp
        retl
```

Backup

Peephole Optimization

Simple compiler do not perform machine-independent code improvement

- They generates *naive* code

It is possible to take the target hole and optimize it

- Sub-optimal sequences of instructions that match an optimization pattern are transformed into optimal sequences of instructions
- This technique is known as **peephole optimization**
- Peephole optimization usually works by sliding a window of several instructions (a *peephole*)

Peephole Optimization

Goals:

- improve performance
- reduce memory footprint
- reduce code size

Method:

1. Exam short sequences of target instructions
2. Replacing the sequence by a more efficient one.

- redundant-instruction elimination
- algebraic simplifications
- flow-of-control optimizations
- use of machine idioms

Peephole Optimization
Common Techniques

Elimination of redundant loads and stores

$r2 := r1 + 5$ $i := r2$ $r3 := i$ $r4 := r3 \times 3$	becomes	$r2 := r1 + 5$ $i := r2$ $r4 := r2 \times 3$

Constant folding

$r2 := 3 \times 2$	becomes	$r2 := 6$

Peephole Optimization
Common Techniques

Constant propagation

r2 := 4 r3 := r1 + r2 r2 := ...	becomes	r2 := 4 r3 := r1 + 4 r2 := ...	and then	r3 := r1 + 4 r2 := ...
r2 := 4 r3 := r1 + r2 r3 := *r3	becomes	r3 := r1 + 4 r3 := *r3	and then	r3 := *(r1+4)
r1 := 3 r2 := r1 × 2	becomes	r1 := 3 r2 := 3 × 2	and then	r1 := 3 r2 := 6

Peephole Optimization
Common Techniques

Copy propagation

r2 := r1 r3 := r1 + r2 r2 := 5	becomes	r2 := r1 r3 := r1 + r1 r2 := 5	and then	r3 := r1 + r1 r2 := 5

Strength reduction

$r1 := r2 \times 2$	becomes	$r1 := r2 + r2$	or	$r1 := r2 << 1$
$r1 := r2 / 2$	becomes	$r1 := r2 >> 1$		
$r1 := r2 \times 0$	becomes	$r1 := 0$		