Code analysis *and* transformation

Loops

Simone Campanoni
simone.campanoni@northwestern.edu

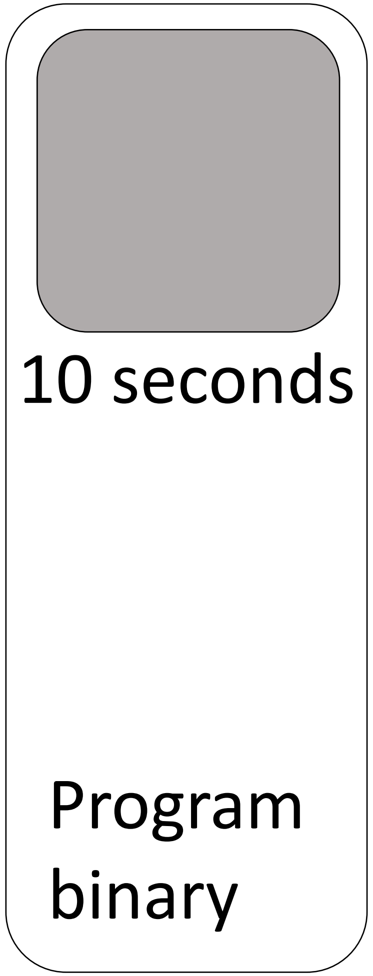
Outline

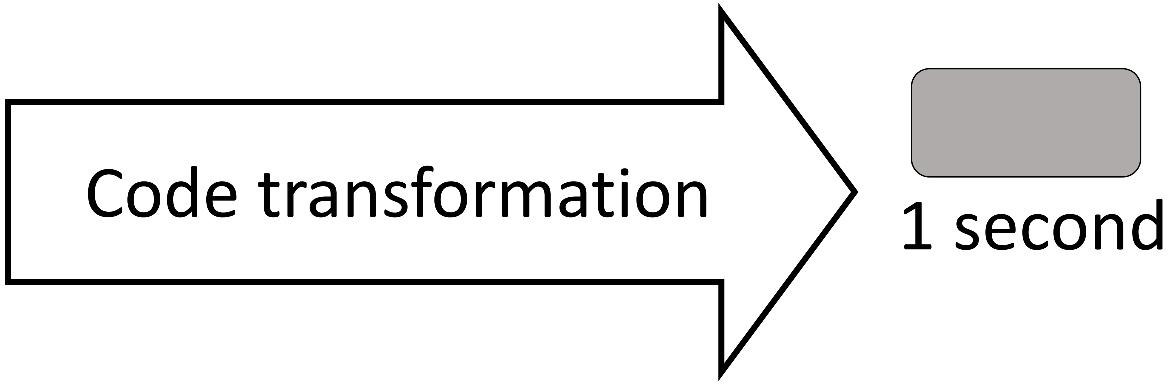
- Loops

- Identify loops

- Induction variables

Impact of optimized code to program





How much did we optimize the overall program?

- Coverage of optimized code
- 10% coverage: Speedup=~1.10x (100->91 seconds)
- 20% coverage: Speedup=~1.22x (100->82 seconds)
- 90% coverage: Speedup=~5.26x (100->19 seconds)

90% of time is spent in 10% of code

Cold code

Loop
Hot code

Identify hot code to succeed!!!

Loops …

... but where are they?

... How can we find them?

Loops in source code

```
for (i=0; i < 10; i++){
   …
}
```

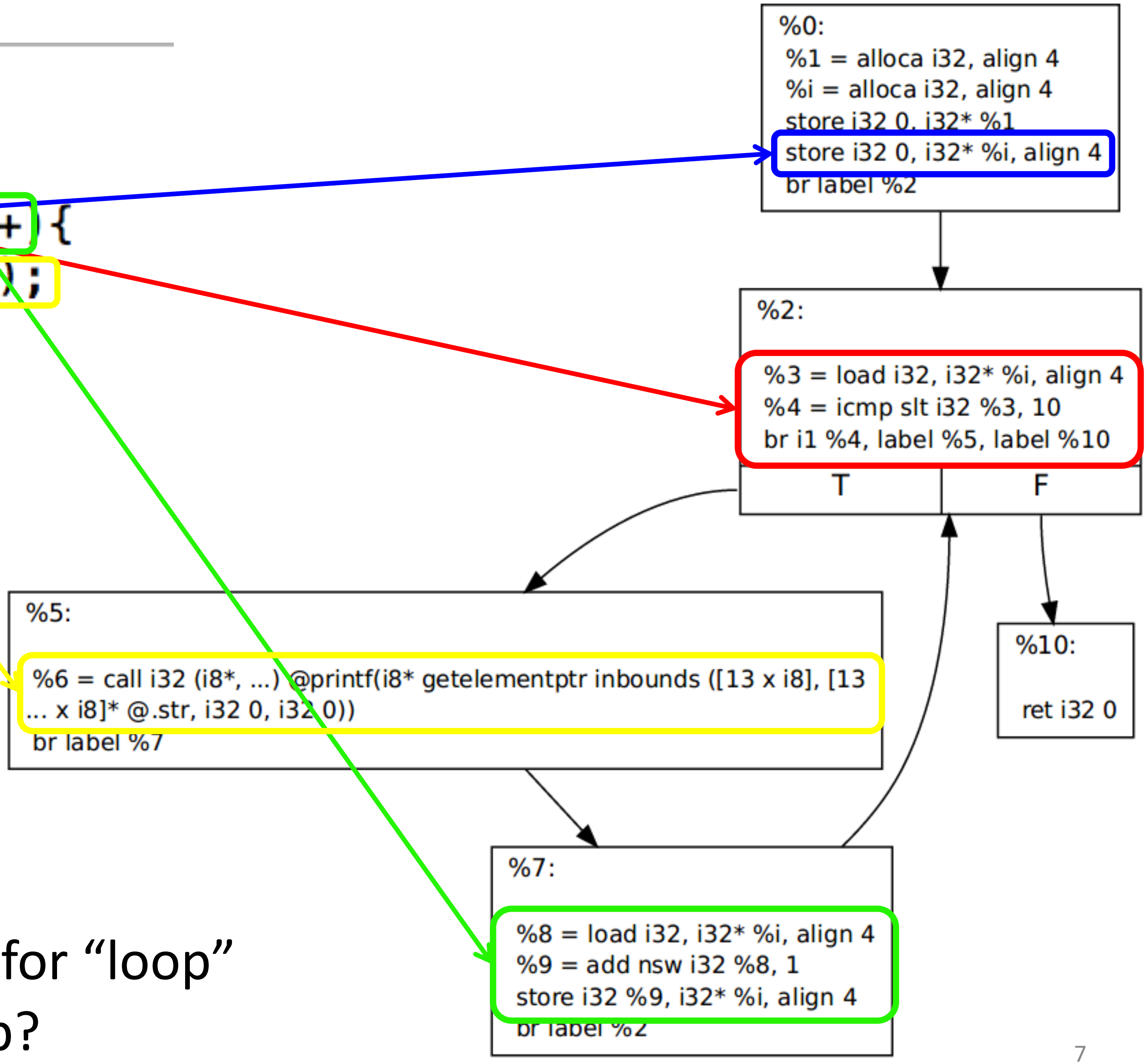
```
i=0;
while (i < 10){
   …
   i++;
}
```

```
i=0;
do {
   …
   i++;
} while (i < 10);
```

```
S={0,1,…,10}
for (i : S){
  …
}
```

Is there a LLVM IR instruction “for”?
There is no IR instruction for “loop”

```c
#include <stdio.h>

int main (){
  for (int i=0; i < 10; i++){
    printf("Hello world\n");
  }
  return 0;
}
```

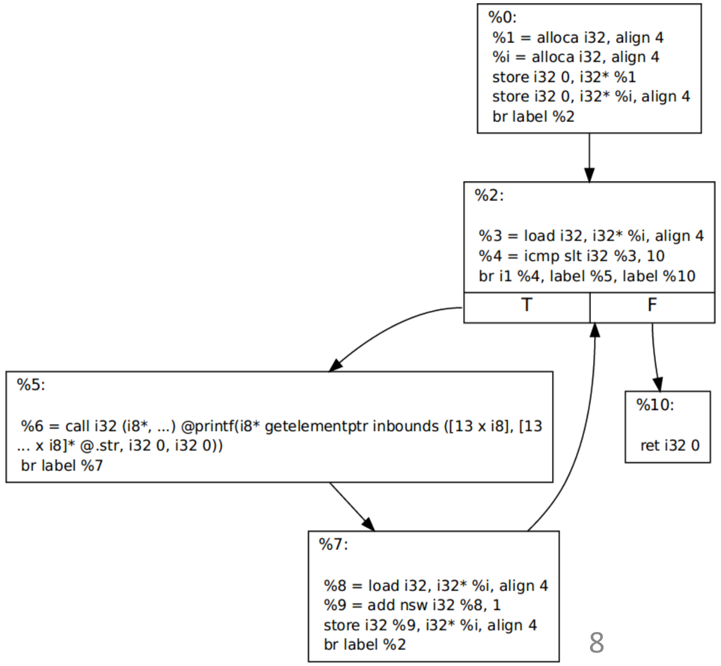


CFG for 'main' function

- Target optimization:
  we need to identify loops
- There is no IR instruction for “loop”
- How to identify an IR loop?

Loops in IR

- Loop identification control flow analysis:
  - Input: Control-Flow-Graph
  - Output: loops in CFG
  - Not sensitive to input syntax: a uniform treatment for all loops
- Define a loop in graph terms
- Intuitive properties of a loop
  - Single entry point
  - Edges must form at least a cycle in CFG
- How to check these properties automatically?



CFG for 'main' function

Outline

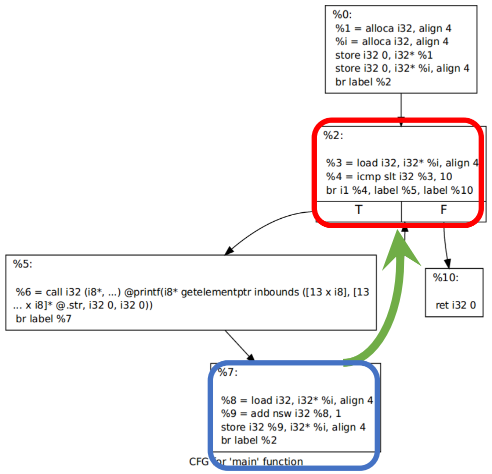
- Loops
- Identify loops
- Induction variables

Natural loops in CFG

- **Header**: node that dominates all other nodes in a loop
  Single entry point of a loop

- **Back edge**: edge (tail -> head) whose head dominates its tail

- **Natural loop** of a back edge:
  the smallest set of nodes that includes
  the head and tail of that back edge,
  and has no predecessors outside the set,
  except for the predecessors of the header.



Identify natural loops

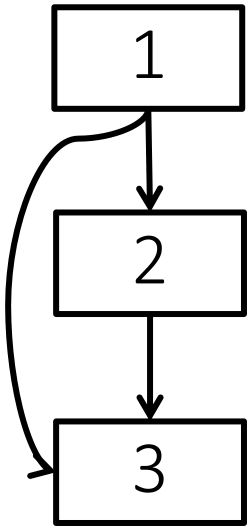
① Find the dominator relations in a flow graph

② Identify the back edges

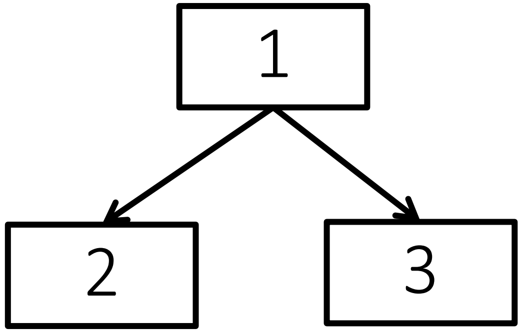
③ Find the natural loop associated with the back edge

Immediate dominators

Definition: the immediate dominator of a node *n*
is the unique node that strictly dominates *n* (i.e., it isn't n)
but does not strictly dominate another node that strictly dominates *n*



CFG



Dominator tree

Immediate dominators

Identify natural loops

① Find the dominator relations in a flow graph

② Identify the back edges

③ Find the natural loop associated with the back edge

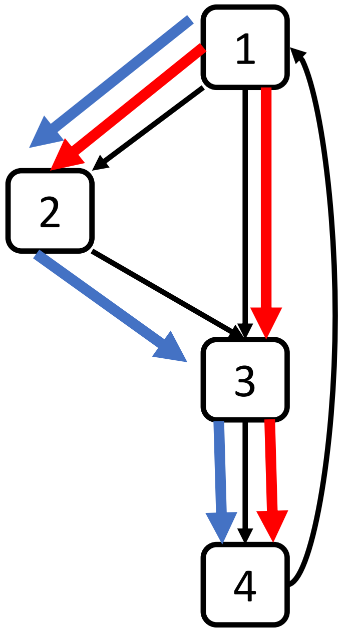
Finding back-edges

Definition:
a back-edge is an arc (tail -> head) whose head dominates its tail

(A) Depth-first spanning tree

Spanning tree of a graph

Definition:
A tree T is a *spanning tree* of a graph G if
T is a subgraph of G that contains all the vertices of G.

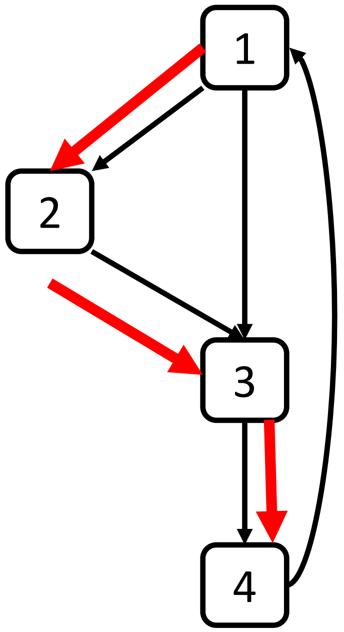


Depth-first spanning tree of a graph

Idea:
Make a path as long as possible,
and then go back (backtrack) to add branches also as long as possible.

Algorithm

```
s = new Stack();  s.add(G.entry); mark(G.entry);
While (!s.empty()){
  1: v = s.pop();
  2: if (v’ = adjacentNotMarked(v, G)){
  3:     mark(v’) ; DFST.add((v, v’));
  4:     s.push(v’);
  } }
```

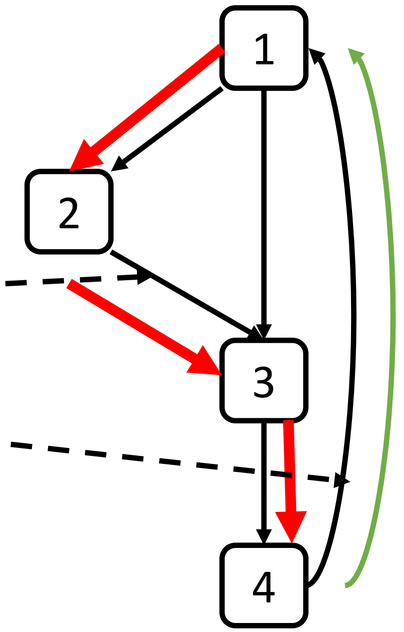


Finding back-edges

Definition:
a back-edge is an arc (tail -> head) whose head dominates its tail

(A) Depth-first spanning tree

- Compute retreating edges in CFG:
  - **Advancing edges**: from ancestor to proper descendant
  - **Retreating edges**: from descendant to ancestor



(B) For each retreating edge t->h, check if h dominates t

- If h dominates t, then t->h is a back-edge

Identify natural loops

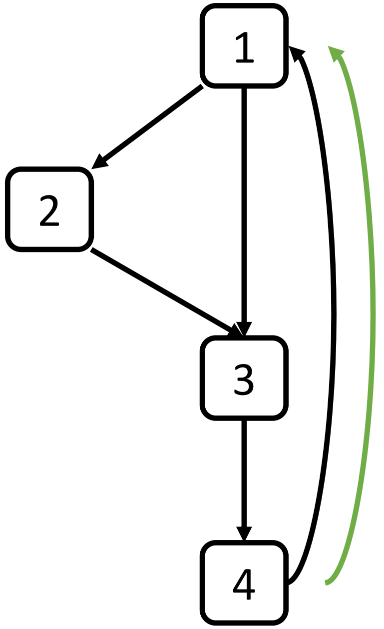
① Find the dominator relations in a flow graph

② Identify the back edges

③ Find the natural loop associated with the back edge

Finding natural loops

Definition: the natural loop of a back edge is the smallest set of nodes that includes the head and tail of the back edge,
and has no predecessors outside the set,
except for the predecessors of the header



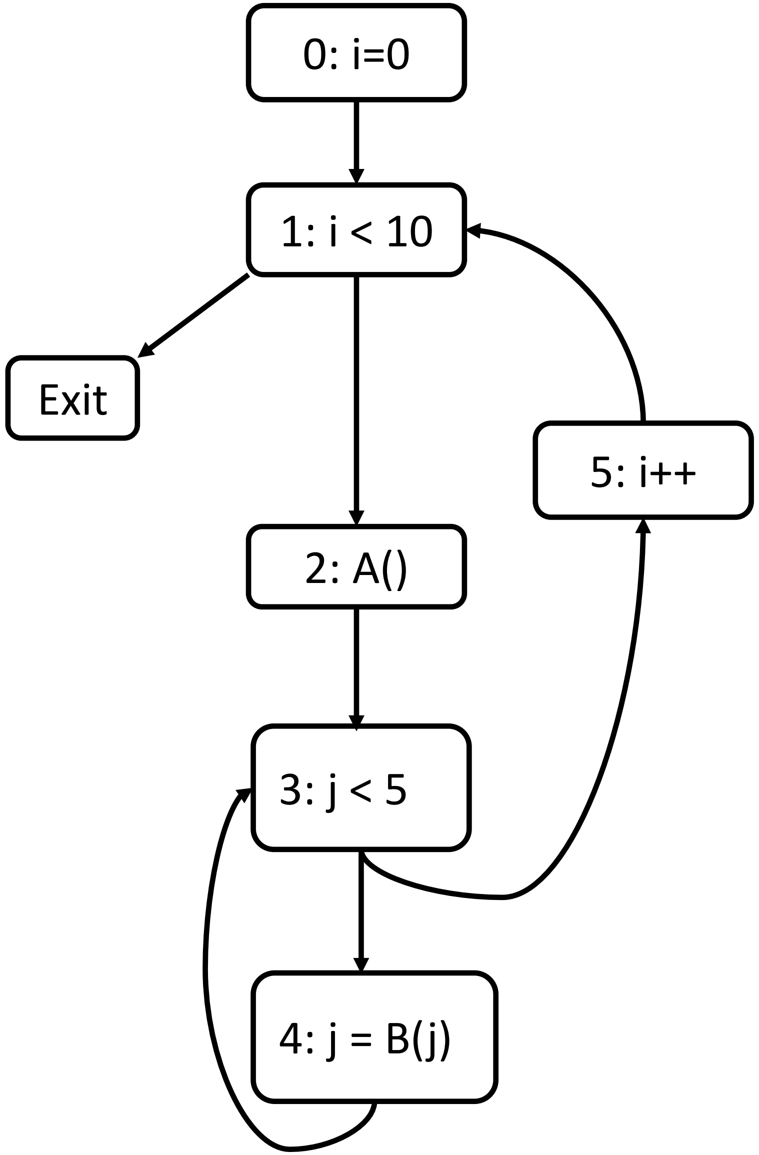
Let *t->h* be the back-edge

A. Delete *h* from the flow graph
B. Find those nodes that can reach *t* 2 3 4
   from the outgoing edges of *h*
   (those nodes plus *h* form the natural loop of *t->h*)

1

Natural loop example

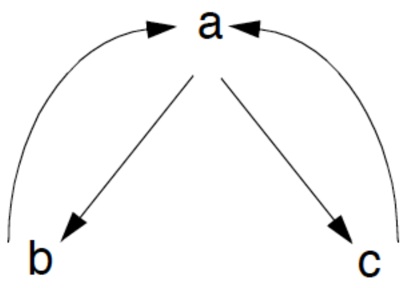
```
For (int i=0; i < 10; i++){
   A();
   while (j < 5){
       j = B(j);
   }
}
```



Identify inner loops

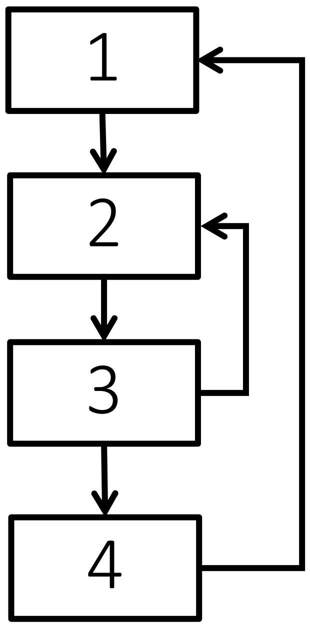
- If two loops do not have the same header
  - They are either disjoint, or
  - One is entirely contained (nested within) the other
    - Outer loop, inner loop
    - Loop nesting relation **Graph/DAG/tree? Why?**
- What about if two loops share the same header?

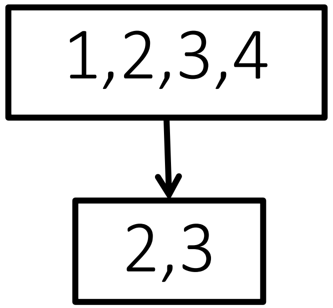
```
while (a: i < 10){
    b: if (i == 5) continue;
    c: ...
}
```



Loop nesting tree

- **Loop-nest tree**: each node represents the blocks of a loop, and parent nodes are enclosing loops.
- The leaves of the tree are the inner-most loops.

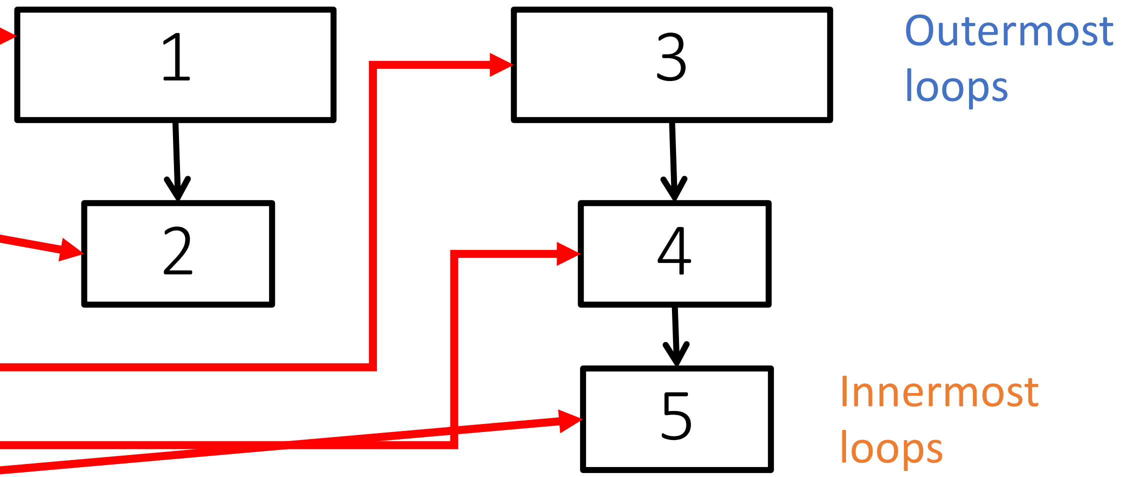




How to compute the loop-nest tree?

Loop nesting forest

```
void myFunction (){
1: while (…){
2:     while (…){ … }
   }
   …
3: for (…){
4:     do {
5:         while(…) {…}
       } while (…)
   }
}
```



Defining loops in graphic-theoretic terms

Is it good? Bad? Implications?

```
L1: ...
    if (X < 10) goto L2;
    goto L1;

L2: ...
```

The good

```
if (...) goto L1;
...
do {
    ...
L1: ...
    } while (X < 10);
```

The bad

Loops in LLVM

Function ⇒ Natural loops ⇒ Merged natural loops (loops with the same header are merged)

Identify loops in LLVM

- Rely on other passes to identify loops

```
#include "llvm/Analysis/LoopInfo.h"
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<LoopInfoWrapperPass>();
  AU.setPreservesAll();
}
```

- Fetch the result of the LoopInfoWrapperPass analysis

```
LoopInfo& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```

- Iterate over outermost loops

```
for (auto i : LI){
  Loop *loop = &*i;
  ...
}
```

```
void myFunction (){
1: while (…){
2:     while (…){ … }
   }
   …
3: for (…){
4:     do {
5:         while(…) {…}
       } while (…)
   }
}
```

Loops in LLVM: sub-loops

- Iterate over sub-loops of a loop

```
vector<Loop *> subLoops = loop->getSubLoops();
for (auto j : subLoops){
  Loop *subloop = &*j;
  ...
}
```

```
void myFunction (){
1: while (…){
2:    while (…){ … }
   }
   …
3: for (…){
4:    do {
5:        while(…) {…}
      } while (…)
   }
}
```

Outline

- Loops
- Identify loops
- Induction variables

Code example

```
int myF (int k){
  int i;
  int s = 0;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

O0

```
; <label>:3:                                      ; preds = %7, %2
  %.01 = phi i32 [ 0, %2 ], [ %6, %7 ]
  %.0 = phi i32 [ 0, %2 ], [ %8, %7 ]
  %4 = icmp slt i32 %.0, 100
  br i1 %4, label %5, label %9

; <label>:5:                                      ; preds = %3
  %6 = add nsw i32 %.01, %0
  br label %7

; <label>:7:                                      ; preds = %5
  %8 = add nsw i32 %.0, 1
  br label %3
```

Is adding “k” to “s” for every loop iteration really needed?

Code example

```
int myF (int k){
  int i;
  int s = 0;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

Value of s

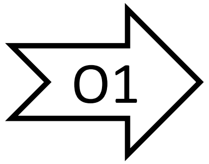
0
k
2k
3k
4k
…
100k

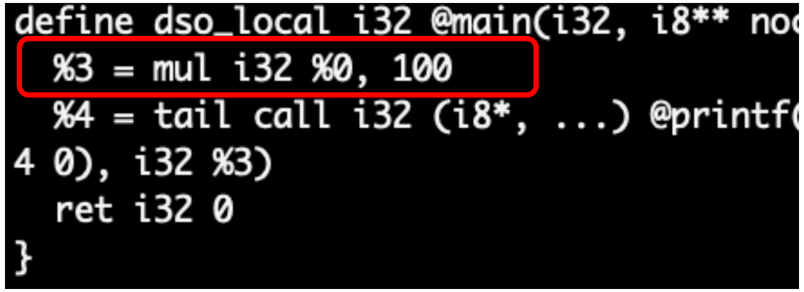
Code example

```
int myF (int k){
  int i;
  int s = 0;
  s = k * 100;



  return s;
}
```

Code example

```
int myF (int k){
  int i;
  int s = 0;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```





```
define dso_local i32 @main(i32, i8** noc
  %3 = mul i32 %0, 100
  %4 = tail call i32 (i8*, ...) @printf(
4 0), i32 %3)
  ret i32 0
}
```

```
int myF (int k){
  int i;
  int s = 0;
  s = k * 100;


  return s;
}
```

Code example 2

```
int myF (int k){
  int i;
  int s = 5;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

O0

```
; <label>:3:                                    ; preds = %7, %2
  %.01 = phi i32 [ 5, %2 ], [ %6, %7 ]
  %.0 = phi i32 [ 0, %2 ], [ %8, %7 ]
  %4 = icmp slt i32 %.0, 100
  br i1 %4, label %5, label %9

; <label>:5:                                    ; preds = %3
  %6 = add nsw i32 %.01, %0
  br label %7

; <label>:7:                                    ; preds = %5
  %8 = add nsw i32 %.0, 1
  br label %3
```

Code example 2

```
int myF (int k){
  int i;
  int s = 5;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

Value of k

5

5 + k

5 + 2k

5 + 3k

5 + 4k

…

5 + 100k

Code example 2

```
int myF (int k){
  int i;
  int s ;
  s = k * 100;
  s = s + 5;

  return s;
}
```

Code example 2

```
int myF (int k){
  int i;
  int s = 5;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

O1

```
define dso_local i32 @main(i32, i8** noc
  %3 = mul i32 %0, 100
  %4 = add i32 %3, 5
  %5 = tail call i32 (i8*, ...) @printf(i
4 0), i32 %4)
  ret i32 0
}
```

```
int myF (int k){
  int i;
  int s ;
  s = k * 100;
  s = s + 5;

  return s;
}
```

Code example 3

```
int myF (int k, int iters){
  int i;
  int s = 5;
  for (i=0; i < iters; i++){
    s = s + k;
  }
  return s;
}
```

O0

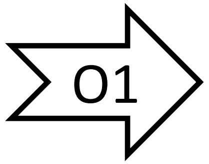
```
; <label>:3:                                      ; preds = %7, %2
  %.01 = phi i32 [ 5, %2 ], [ %6, %7 ]
  %.0 = phi i32 [ 0, %2 ], [ %8, %7 ]
  %4 = icmp slt i32 %.0, %1
  br i1 %4, label %5, label %9

; <label>:5:                                      ; preds = %3
  %6 = add nsw i32 %.01, %0
  br label %7

; <label>:7:                                      ; preds = %5
  %8 = add nsw i32 %.0, 1
  br label %3
```

Code example 3

```
int myF (int k, int iters){
  int i;
  int s ;
  s = k * iters;
  s = s + 5;

  return s;
}
```

Code example 3

```
int myF (int k, int iters){
  int i;
  int s = 5;
  for (i=0; i < iters; i++){
    s = s + k;
  }
  return s;
}
```




```
define dso_local i32 @myF(i32, i32)
  %4 = mul i32 %1, %0
  %5 = add i32 %4, 5
  ret i32 %5
}
```

```
int myF (...){
  int i;
  int s ;
  s = k * iters;
  s = s + 5;

  return s;
}
```

Important information about variable evolution

```
int myF (int k){
  int i;
  int s = 0;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

```
int myF (int k){
  int i;
  int s = 5;
  for (i=0; i < 100; i++){
    s = s + k;
  }
  return s;
}
```

```
int myF (int k, int iters){
  int i;
  int s = 5;
  for (i=0; i < iters; i++){
    s = s + k;
  }
  return s;
}
```

- It is important to understand the evolution of variables
- Important transformations are possible
  only when variable evolutions are analyzed
- Variables with a specific type of evolution (described next)
  are called "induction variables"
  - "s" was an induction variable in all prior examples

Induction variable observation

- **Observation**:
  Some variables change by a constant amount on each loop iteration
  - x initialized at 0; increments by 1
  - y initialized at N; increments by 2
  - These are all induction variables

```
x = 0 ; y = N;
While (…){
   x++;
   y = y + 2;
}
```

- **Definition of induction variable (IV):**
  An IV is a variable that
  - increases or decreases by a fixed amount on every iteration of a loop or
  - it is a linear function of another IV
- How can we identify IVs automatically?

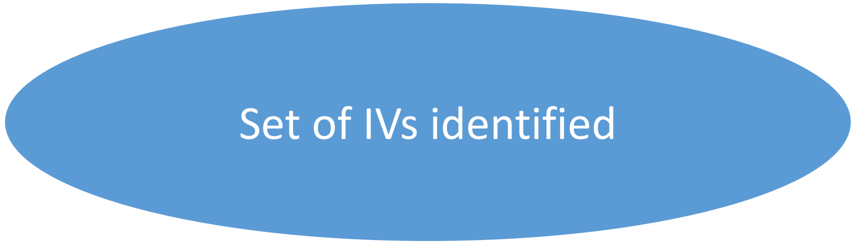
Identify induction variables

Idea

We find induction variables incrementally.

First: we identify the basic cases.

Second: we identify the complex cases.



Iterate the analysis until
we cannot add new IVs

Induction variables

- Basic induction variables
  - i = i op c
  - c is loop invariant

What is a loop-invariant?

  - a.k.a. independent induction variable

- Derived induction variables

Loop-invariant computations

- Let *d* be the following definition

(d) t = x

- d is a loop-invariant of a loop L if (assuming x does not escape)
  - x is constant or
  - All reaching definitions of x are outside the loop, or
  - Only one definition of x reaches d, and that definition is loop-invariant

Loop-invariant computations

- Let *d* be the following definition

(d) t = x op y

- d is a loop-invariant of a loop L if (assuming x, y do not escape)
  - x and y are constants or
  - All reaching definitions of x and y are outside the loop, or
  - Only one definition of x (or y) reaches d, and that definition is loop-invariant

Loop-invariant computations

- Let *d* be the following definition

(d) t = load(x)

- d is a loop-invariant of a loop L if (assuming x does not escape)
  - The memory location pointed by x, mem[x], is constant or
  - All reaching definitions of mem[x] are outside the loop, or
  - Only one definition of mem[x] reaches d, and that definition is loop-invariant

Loop example

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

d is a loop-invariant of a loop L if

x and *y* are constants or

all reaching definitions of *x* and *y* are outside the loop, or

only one definition reaches *x* (or *y*),

and that definition is loop-invariant

??

Loop-invariant computations in LLVM

```
for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi){
  BasicBlock *bb = *bbi;
  for (auto& instr_iter : *bb){
    auto instr = &instr_iter;









  }
}
```

Loop-invariant computations in LLVM

```
for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi){
  BasicBlock *bb = *bbi;
  for (auto& instr_iter : *bb){
    auto instr = &instr_iter;
    if (loop->isLoopInvariant(instr)){
      errs() << prefix << " ";
      instr->print(errs());
      errs() << "\n";
    }






  }
}
```

Loop-invariant computations in LLVM

```
for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi){
  BasicBlock *bb = *bbi;
  for (auto& instr_iter : *bb){
    auto instr = &instr_iter;
    if (loop->isLoopInvariant(instr)){
      errs() << prefix << " ";
      instr->print(errs());
      errs() << "\n";
    }
    if (loop->hasLoopInvariantOperands(instr)){
      errs() << prefix << "     Operand invariants";
      instr->print(errs());
      errs() << "\n";
    }
  }
}
```

Induction variables

- Basic induction variables
  - i = i op c
  - c is loop invariant
  - this definition is executed exactly once per iteration
  - a.k.a. independent induction variable

- Derived induction variables
  - $j = i * c_1 + c_2$
  - c_1 and c_2 are loop invariants
  - this definition is executed exactly once per iteration
  - i is an IV
  - a.k.a. dependent induction variable

Identify induction variables: step 1

Find the basic IVs

① Scan loop body for defs of the form

x = x + c

where c is loop-invariant and
this definition is executed exactly once per iteration

How can we do?
Can we exploit SSA?

② Record these basic IVs as

x = (x, 1, c)

this represents the IV: x = x * 1 + c

Identify induction variables: step 2

Find derived IVs

① Scan for derived IVs of the form

k = i * c1 + c2

where i is an IV and
this is the only definition of k in the loop and
this definition is executed exactly once per iteration

② Record as k = (i, c1, c2)
We say k is in the family of i

Code example

```
int myF1 (int start, int end){
  int i = start;
  while (i < end){
    j = i * 8 + 4;
    i++;
  }
  return j;
}
```

(i, 8, 4)

(i, 1, 1)

```
int myF2 (int start, int end){
  int i = start;
  while (i < end){
    j = i * 8;
    while (j > 0){
      k = j * 42 + i;
      j--;
    }
    i++;
  }
  return j;
}
```

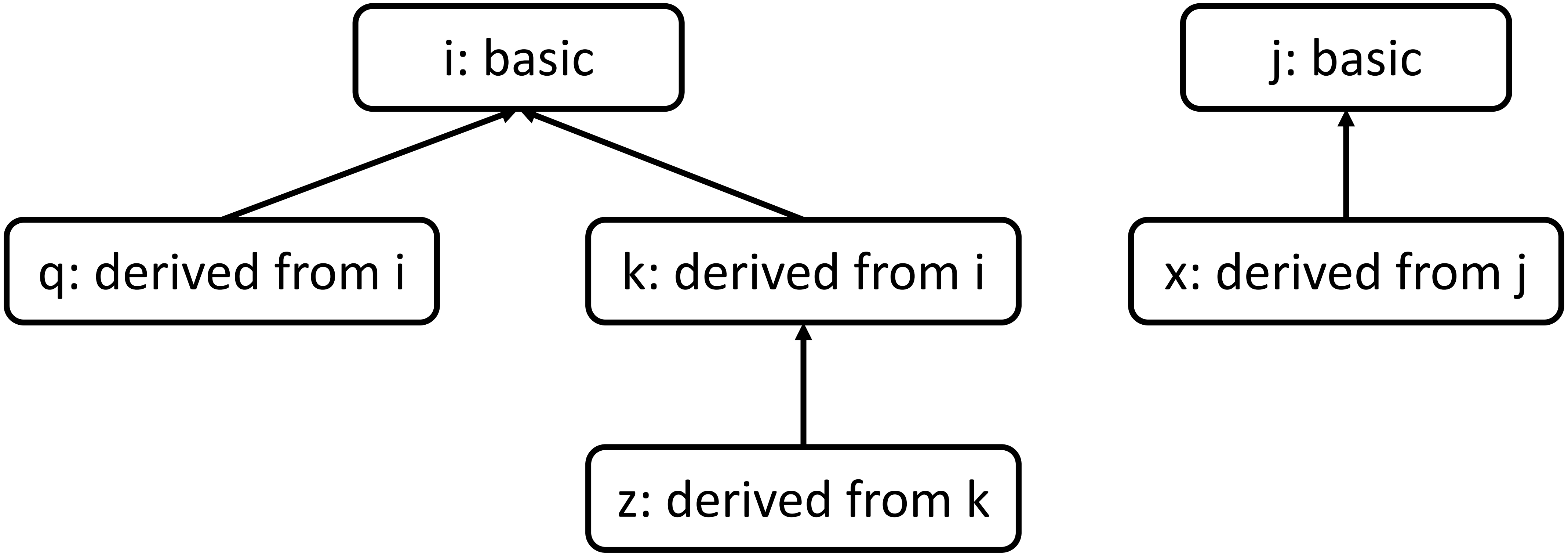
(j, 42, i)

(j, 1, -1)

(i, 8, 0)

(i, 1, 1)

Identified induction variables



A forest of induction variables

Induction variables in LLVM

- You have up to 1 IV per loop
  - This is the IV that control the number of iterations of the loop

```
int j=0;
for (int i=0; i < N; i++){
    j = j + 42;
}
```

  - An IV that starts from 0 and it increments by 1 is called **canonical**

- Potentially many IVs that do not control the #iterations
  - They are called **auxiliary** IVs

Induction variables in LLVM

```
bool runOnFunction(Function &F) override {
  auto &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
  ScalarEvolution *SE =  &getAnalysis<ScalarEvolutionWrapperPass>().getSE();

  errs() << "Function: " << F.getName() << "\n";
  for (auto i = LI.begin(); i != LI.end(); ++i){
    auto loop = *i;




















    }
  }
  return false;
}
```

Induction variables in LLVM

```
bool runOnFunction(Function &F) override {
  auto &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
  ScalarEvolution *SE =  &getAnalysis<ScalarEvolutionWrapperPass>().getSE();

  errs() << "Function: " << F.getName() << "\n";
  for (auto i = LI.begin(); i != LI.end(); ++i){
    auto loop = *i;
    errs() << " Loop\n";
    errs() << "   Header = " << *loop->getHeader() << "\n";
    auto IV = loop->getInductionVariable(*SE);
    if (IV != nullptr){
      errs() << "   IV = " << *IV << "\n";
    }

    for (auto bb : loop->getBlocks()){
      for (auto &inst : *bb){
        auto phi = dyn_cast<PHINode>(&inst);
        if (phi == nullptr){
          continue ;
        }
        if (loop->isAuxiliaryInductionVariable(*phi, *SE)){
          errs() << "   Auxiliary IV = " << *phi << "\n";
        }
      }
    }
  }
  return false;
}
```

Induction variables in LLVM

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();
  AU.setPreservesAll();
}
```

Identification of Induction variables in LLVM

- Based on the analysis called scalar-evolution:
  - Scalar evolution:
    change in the value of scalar variables over iterations of the loop
  - It represents scalar expressions (e.g., x = y op z)
    - It supports induction variables (e.g., x = x + 1)
  - It lowers the burden of explicitly handling the composition of expressions

- LLVM implementation: ScalarEvolutionWrapperPass

Induction variable vs. scalar evolution

- Basic IV (BIV):
  It increases or decreases by a fixed amount
  on every iteration of a loop

- IV:
  A BIV or a linear function of another IV

- Generalized IV (GIV):
  It increases or decreases by a given amount
  It can depend non-linearly on other BIVs/GIVs
  It can have multiple updates

Chain of recurrences

It is a formalism to analyse expressions in BIV and GIV expressing them as **Recurrences**

n! = 1 x 2 x … x n ⟺ n! = (n-1)! x n

f(n) = 1 x 2 x … x n ⟺ f(n) = f(n-1) * n

Basic recurrences

```
int f = k0;
for (int j=0; j < n ; j++){
  ... = f;
  f = f + k1
}
```

Assuming k1 to be a loop invariant

$$f(i) = \begin{cases} k0 & \text{if } i == 0 \\ f(i-1) + k1 & \text{if } i > 0 \end{cases}$$

i-th value

Starts with k0, and it increments by k1 every time

Basic recurrence = {k0, +, k1}

So what is a chain of references?

Chain of recurrences

```
int f = g = k0;
for (int j=0; j < n ; j++){
   ... = f;
   g = g + f;
   f = f + k1
}
```

$$f(i) = \begin{cases} k0 & \text{if } i == 0 \\ f(i-1) + k1 & \text{if } i > 0 \end{cases}$$

This is an IV

Basic recurrence = {k0, +, k1}

$$g(i) = \begin{cases} k0 & \text{if } i == 0 \\ g(i-1)+f(i-1) & \text{if } i > 0 \end{cases}$$

This is not an IV

Chain of recurrence = {k0, +, {k0, +, k1}}

⇩

{k0, +, k0, +, k1}

Chain of recurrences

```
for (int x=0; x < n ; x++){
   p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

How can be compute it?

Chain of recurrence for p[x] = {7, +, 6, +, 10, +, 6}

x	0	1	2	3	4	5

What is the value of p[x] when x is equal to 0? 7

What is the value of p[x] when x is equal to 1? 13

What is the value of p[x] when x is equal to 2? 29

What is the value of p[x] when x is equal to 3? 61

What is the value of p[x] when x is equal to 4? 115

What is the value of p[x] when x is equal to 5? 197

Chain of recurrences

```
for (int x=0; x < n ; x++){
  p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

How can be compute it?

Chain of recurrence for p[x] = {7, +, 6, +, 10, +, 6}

x	0	1	2	3	4	5
	7	13	29	61	115	197

Chain of recurrences

```
for (int x=0; x < n ; x++){
   p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

x	0	1	2	3	4	5
p[x]	7 →	13	29	61	115	197
What is the increment between iterations?		6	16	32	54	82

Chain of recurrences

```
for (int x=0; x < n ; x++){
   p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

x	0	1	2	3	4	5
p[x]	7	13	29	61	115	197
D	-	6 →	16	32	54	82

Chain of recurrences

```
for (int x=0; x < n ; x++){
   p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

x	0	1	2	3	4	5
p[x]	7	13	29	61	115	197
D	-	6	16	32	54	82
D^2	-	-	10 →	16	22	28

Chain of recurrences

```
for (int x=0; x < n ; x++){
   p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

x	0	1	2	3	4	5
p[x]	7	13	29	61	115	197
D	-	6	16	32	54	82
D^2	-	-	10	16	22	28
D^3	-	-	-	6	6	6

Chain of recurrence = {7, +, 6, +, 10, +, 6}

Chain of recurrences

Chain of recurrence = {7, +, 6, +, 10, +, 6}

```c
void myF (int *p, int n){
  for (int x=0; x < n; x++){
    p[x] = x*x*x + 2*x*x + 3*x + 7;
  }
}
```

And if you run scalar evolution of LLVM:

Instruction   %16 = add nsw i32 %15, 7 is SCEVAddRecExpr

SCE: {7,+,6,+,10,+,6}<%7>

LLVM scalar evolution example

```c
#include <stdio.h>

int main (int argc, char *argv[]){

  for (int i=0; i < argc; i++){
    printf("ciao\n");
  }
  return 0;
}
```

- SCEV: {A, B, C}<flag>*<%D>
  - A: Initial; B: Operator; C: Operand; D: basic block where it get defined

```llvm
define i32 @main(i32 %argc, i8** %argv) #0 {
  br label %1

; <label>:1                                       ; preds = %5, %0
  %i.0 = phi i32 [ 0, %0 ], [ %6, %5 ]
  %2 = icmp slt i32 %i.0, %argc
  br i1 %2, label %3, label %7

; <label>:3                                       ; preds = %1
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6 x i8]* @.str, i32 0, i32 0))
  br label %5

; <label>:5                                       ; preds = %3
  %6 = add nsw i32 %i.0, 1
  br label %1

; <label>:7                                       ; preds = %1
  ret i32 0
}
```

LLVM scalar evolution example

```c
#include <stdio.h>

int main (int argc, char *argv[]){

  for (int i=0; i < argc; i++){
    printf("ciao\n");
  }
  return 0;
}
```

- SCEV: {A, B, C}<flag>*<%D>
  - A: Initial; B: Operator; C: Operand; D: basic block where it get defined

```
cat-c program.bc -c -emit-llvm -o loop_0.bc
Function: main
 New loop
   Instruction   %i.0 = phi i32 [ 0, %0 ], [ %6, %5 ] is SCEVAddRecExpr
     SCE: {0,+,1}<nuw><nsw><%1>
   Instruction   %6 = add nsw i32 %i.0, 1 is SCEVAddRecExpr
     SCE: {1,+,1}<nuw><nsw><%1>
```

LLVM scalar evolution example: pass deps

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();
  AU.setPreservesAll();
}
```

```
bool runOnFunction(Function &F) override {
  LoopInfo& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
  ScalarEvolution *SE =  &getAnalysis<ScalarEvolutionWrapperPass>().getSE();

  errs() << "Function: " << F.getName() << "\n";
  for (auto i = LI.begin(); i != LI.end(); ++i){
    Loop *loop = *i;
    errs() << " New loop\n";
    for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi){
      BasicBlock *bb = *bbi;
      for (BasicBlock::iterator i = bb->begin(); i != bb->end(); ++i){
        Instruction *inst = &*i;
        const SCEV *S = SE->getSCEV(inst);
        if (auto *AR = dyn_cast<SCEVAddRecExpr>(S)){
          errs() << "   Instruction ";
          i->print(errs());
          errs() << " is SCEVAddRecExpr\n" ;

          errs() << "     SCE: " ;
          AR->print(errs());
          errs() << "\n";
        }
      }
    }
  }
  return false;
}
```

Scalar evolution in LLVM

- Analysis used by
  - Induction variable analysis
  - Strength reduction
  - Vectorization
  - …
- SCEVs are modeled by the llvm::SCEV class
  - There is a sub-class for each kind of SCEV (e.g., llvm::SCEVAddExpr)
- A SCEV is a tree of SCEVs
  - Leafs:
    - Constant : llvm:SCEVConstant (e.g., 1)
    - Unknown: llvm:SCEVUnknown (e.g., %v = call rand())
  - To iterate over a tree: llvm:SCEVVisitor

Always have faith in your ability

Success will come your way eventually

Best of luck!