CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- **Big thank you to UW CSE faculty member, Hal Perkins**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, …)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, …)

Agenda

- Overview of SSA IR – review and finish
  - Constructing SSA graphs
  - Sample of SSA-based optimizations
  - Converting back from SSA form
  - Sources: Appel ch. 19, also an extended discussion in Cooper-Torczon sec. 9.3, Mike Ringenburg's CSE 401 slides (13wi)
- Compiler back-end organization
- Instruction selection – tree pattern matching
- Sources:
  - Slides by Keith Cooper (Rice); Appel ch. 9; burg/iburg slides by Preston Briggs, CSE 501 Sp09
  - Burg/iburg paper: "Engineering a Simple, Efficient Code Generator", Fraser, Hanson, & Proebsting, ACM LOPLAS v1, n3 (Sept. 1992)
- Instruction scheduling issues – latencies
  - List scheduling
- Register allocation
  - Register allocation constraints
  - Local methods
  - Faster compile, slower code, but good enough for lots of things (JITs, …)
  - Global allocation – register coloring

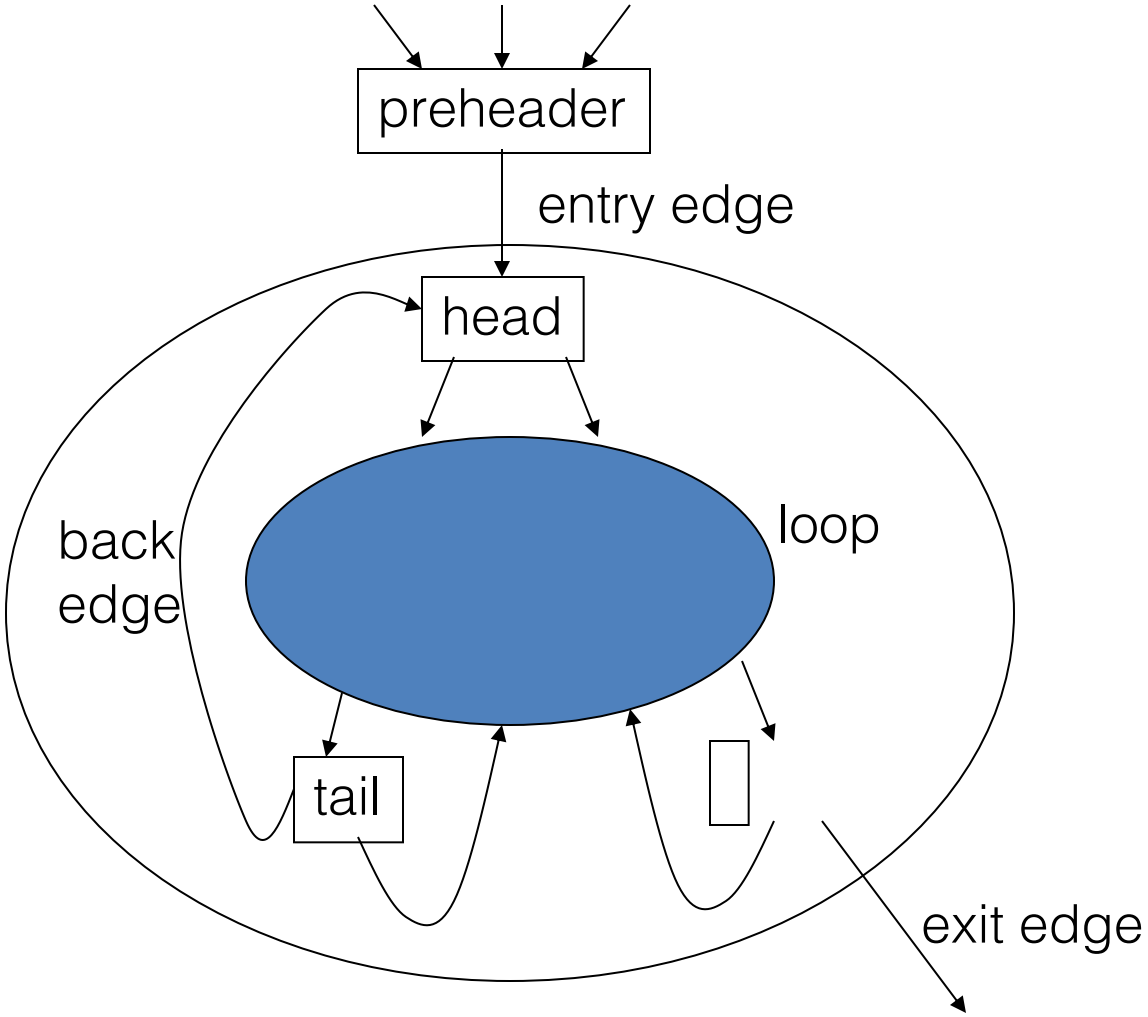
Review: What Is a Loop?

- In a control flow graph, a loop is a set of nodes *S* such that:
  - *S* includes a *header node* h
  - From any node in *S* there is a path of directed edges leading to h
  - There is a path from h to any node in *S*
  - There is no edge from any node outside *S* to any node in *S* other than h

Review: Entries and Exits

- In a loop
  - An *entry node* is one with some predecessor outside the loop
  - An *exit node* is one that has a successor outside the loop
- Corollary: A loop may have multiple exit nodes, but only one entry node

Review: Loop Terminology



Review: Reducible Flow Graphs

- In a reducible flow graph, any two loops are either nested or disjoint
- Roughly, to discover if a flow graph is reducible, repeatedly delete edges and collapse together pairs of nodes (x,y) where x is the only predecessor of y
- If the graph can be reduced to a single node it is reducible
  - Caution: this is the "powerpoint" version of the definition – see a good compiler book for the careful details

Review: Finding Loops in Flow Graphs

- We use *dominators* for this
- Recall
  - Every control flow graph has a unique start node s_0
  - Node x dominates node y if every path from s_0 to y must go through x
  - A node x dominates itself

Review: Calculating Dominator Sets

- D[n] is the set of nodes that dominate n
  - $D[s_0] = \{ s_0 \}$
  - $D[n] = \{ n \} \cup (\cap_{p \in pred[n]} D[p])$
- Set up an iterative analysis as usual to solve this
  - Except initially each D[n] must be all nodes in the graph – updates make these sets smaller if changed

Review: Immediate Dominators

- Every node n has a single *immediate dominator* idom(n)
  - idom(n) dominates n
  - idom(n) differs from n – i.e., strictly dominates
  - idom(n) does not dominate any other strict dominator of n
    - i.e., strictly dominates and is nearest dominator
- Fact (er, theorem): If a dominates n and b dominates n, then either a dominates b or b dominates a

  ∴ idom(n) is unique

Review: Dominator Tree

- A *dominator tree* is constructed from a flowgraph by drawing an edge form every node in n to idom(n)
  - This will be a tree. Why?

Basic and Derived Induction Variables

- Variable i is a *basic induction variable* in loop L with header h if the only definitions of i in L have the form i:=i±c where c is loop invariant
- Variable k is a *derived induction variable* in L if:
  - There is only one definition of k in L of the form k:=j*c or k:=j+d where j is an induction variable and c, d are loop-invariant, *and*
  - if j is a derived variable in the family of i, then:
    - The only definition of j that reaches k is the one in the loop, *and*
    - there is no definition of i on any path between the definition of j and the definition of k

Optimizing Induction Variables

- Strength reduction: if a derived induction variable is defined with j:=i*c, try to replace it with an addition inside the loop
- Elimination: after strength reduction some induction variables are not used or are only compared to loop-invariant variables; delete them
- Rewrite comparisons: If a variable is used only in comparisons against loop-invariant variables and in its own definition, modify the comparison to use a related induction variable

Loop Unrolling

- If the body of a loop is small, much of the time is spent in the "increment and test" code
- Idea: reduce overhead by *unrolling* – put two or more copies of the loop body inside the loop

Loop Unrolling

- Basic idea: Given loop L with header node h and back edges s_i->h
  1. Copy the nodes to make loop L' with header h' and back edges s_i'->h'
  2. Change all back edges in L from s_i->h to s_i->h'
  3. Change all back edges in L' from s_i'->h' to s_i'->h

Unrolling Algorithm Results

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1' else L2
L1':  x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1 else L2
L2:
```

Hmmmm….

- Not so great – just code bloat
- But: use induction variables and various loop transformations to clean up

After Some Optimizations

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1' else L2
L1':  x := M[i]
    s := s + x
    i := i + 4
  if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i<n goto L1 else L2
L2:
```

Still Broken…

- But in a different, better(?) way
- Good code, but only correct if original number of loop iterations was even
- Fix: add an epilogue to handle the “odd” leftover iteration

Fixed

- Before

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
  if i<n goto L1 else L2
L2:
```

- After

```
      if i<n-8 goto L1 else L2
L1:   x := M[i]
      s := s + x
      x := M[i+4]
      s := s + x
      i := i + 8
      if i<n-8 goto L1 else L2
L2:   x := M[i]
      s := s+x
      i := i+4
      if i < n goto L2 else L3
L3:
```

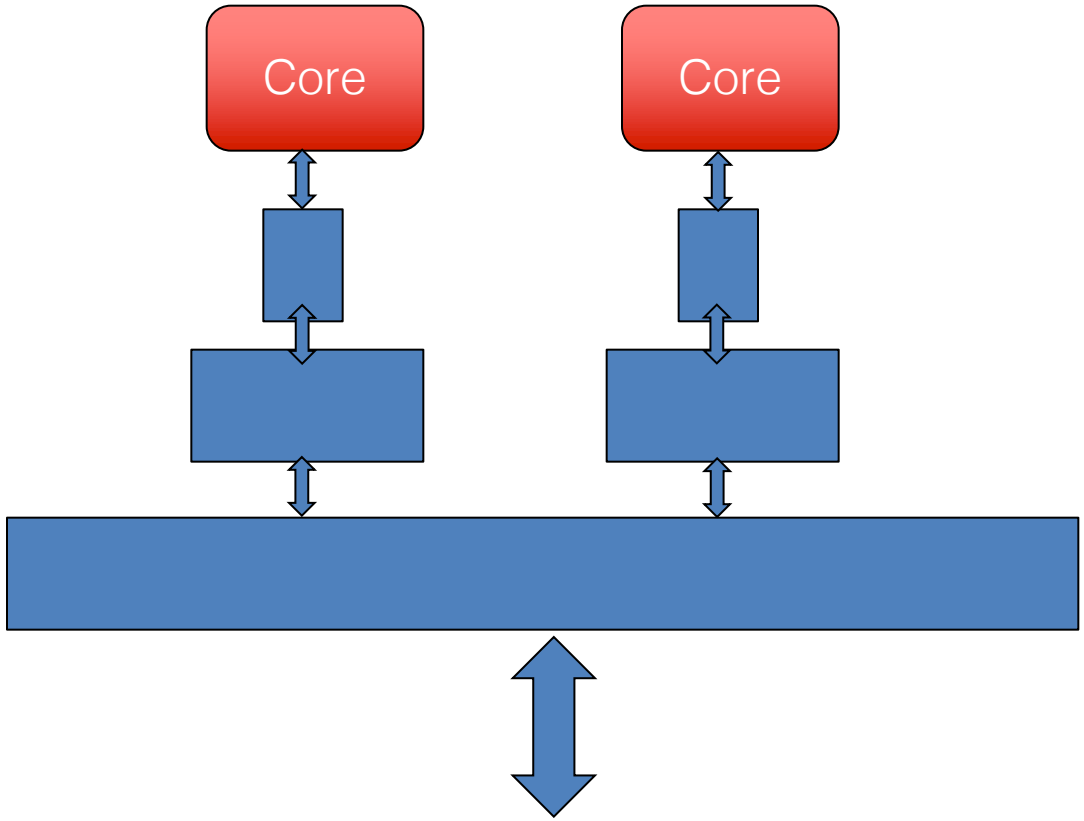
Postscript

- This example only unrolls the loop by a factor of 2
- More typically, unroll by a factor of K
  - Then need an epilogue that is a loop like the original that iterates up to K-1 times

Memory Hierarchies

- One of the great triumphs of computer design
- Effect is a large, fast memory
- Reality is a series of progressively larger, slower, cheaper stores, with frequently accessed data automatically staged to faster storage (cache, main storage, disk)
- Programmer/compiler typically treats it as one large store. (but not always the best idea)
- Hardware maintains cache coherency – most of the time

Intel Haswell Caches



L1 = 64 KB per core

L2 = 256 KB per core

L3 = 2-8 MB shared

Just How Slow is Operand Access?

- Instruction ~5 per cycle
- Register 1 cycle
- L1 CACHE ~4 cycles
- L2 CACHE ~10 cycles
- L3 CACHE (unshared line) ~40 cycles
- DRAM ~100 ns

Memory Issues

- Byte load/store is often slower than whole (physical) word load/store
  - Unaligned access is often extremely slow
- Temporal locality: accesses to recently accessed data will usually find it in the (fast) cache
- Spatial locality: accesses to data near recently used data will usually be fast
  - "near" = in the same cache block
- But – alternating accesses to blocks that map to the same cache block will cause thrashing

- CPU speed increases have out-paced increases in memory access times
- Memory access now often determines overall execution speed
- "Instruction count" is not the only performance metric for optimization

Data Alignment

- Data objects (structs) often are similar in size to a cache block (≈ 64 bytes)
  ∴ Better if objects don't span blocks
- Some strategies:
  - Allocate objects sequentially; bump to next block boundary if useful
  - Allocate objects of same common size in separate pools (all size-2, size-4, etc.)
- Tradeoff: speed for some wasted space

Instruction Alignment

- Align frequently executed basic blocks on cache boundaries (or avoid spanning cache blocks)
- Branch targets (particularly loops) may be faster if they start on a cache line boundary
  - Often see multi-byte nops in optimized code as padding to align loop headers
  - How much depends on architecture (current intel 16 bytes, current AMD 32 bytes)
- Try to move infrequent code (startup, exceptions) away from hot code
- Optimizing compiler may perform basic-block ordering

Loop Interchange

- Watch for bad cache patterns in inner loops; rearrange if possible
- Example

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < p; k++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

  - b[i,j+1,k] is reused in the next two iterations, but will have been flushed from the cache by the k loop

Loop Interchange

- Solution for this example: interchange j and k loops

```
for (i = 0; i < m; i++)
  for (k = 0; k < p; k++)
    for (j = 0; j < n; j++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

  - Now b[i,j+1,k] will be used three times on each cache load
  - Safe here because loop iterations are independent

Loop Interchange

- Need to construct a data-dependency graph showing information flow between loop iterations
- For example, iteration (j,k) depends on iteration (j',k') if (j',k') computes values used in (j,k) or stores values overwritten by (j,k)
  - If there is a dependency and loops are interchanged, we could get different results – so can't do it

Blocking

- Consider matrix multiply

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

- If a, b fit in the cache together, great!
- If they don't, then every b[k,j] reference will be a cache miss
- Loop interchange (i<->j) won't help; then every a[i,k] reference would be a miss

Blocking

- Solution: reuse rows of A and columns of B while they are still in the cache
- Assume the cache can hold 2*c*n matrix elements (1 < c < n)
- Calculate c × c blocks of C using c rows of A and c columns of B

Blocking

- Calculating c × c blocks of C

```
for (i = i0; i < i0+c; i++)
  for (j = j0; j < j0+c; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

Blocking

- Then nest this inside loops that calculate successive c × c blocks

```
for (i0 = 0; i0 < n; i0+=c)
  for (j0 = 0; j0 < n; j0+=c)
    for (i = i0; i < i0+c; i++)
      for (j = j0; j < j0+c; j++) {
        c[i,j] = 0.0;
        for (k = 0; k < n; k++)
          c[i,j] = c[i,j] + a[i,k]*b[k,j]
      }
```

SSA – Review and Finish

Def-Use (DU) Chains

- Common dataflow analysis problem: Find all sites where a variable is used, or find the definition site of a variable used in an expression
- Traditional solution: def-use chains – additional data structure on top of the dataflow graph
  - Link each statement defining a variable to all statements that use it
  - Link each use of a variable to its definition

DU-Chain Drawbacks

- Expensive: if a typical variable has N uses and M definitions, the total cost *per-variable* is O(N * M), i.e., $O(n^2)$
  - Would be nice if cost were proportional to the size of the program
- Unrelated uses of the same variable are mixed together
  - Complicates analysis – variable looks live across all uses even if unrelated

SSA: Static Single Assignment

- IR where each variable has only one definition in the program text
  - This is a single *static* definition, but that definition can be in a loop that is executed dynamically many times
- Makes many analyses (and associated optimizations) more efficient
- Separates values from memory storage locations
- Complementary to CFG/DFG – better for some things, but cannot do everything

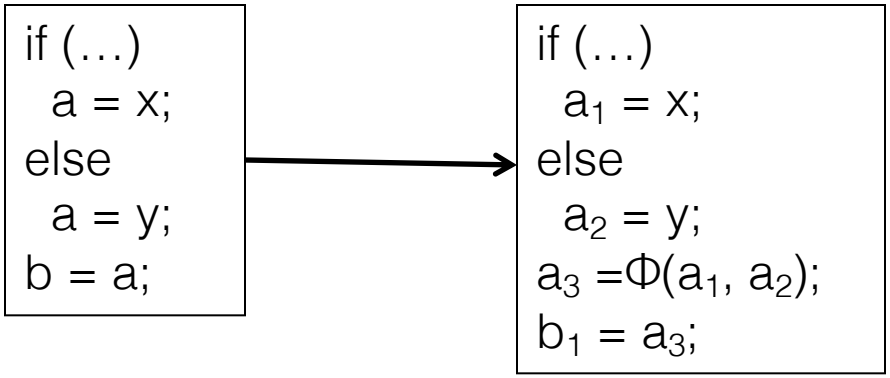
SSA in Basic Blocks

Idea: for each original variable x, create a new variable x_n at the n^{th} definition of the original x. Subsequent uses of x use x_n until the next definition point.

- Original
  - a := x + y
  - b := a – 1
  - a := y + b
  - b := x * 4
  - a := a + b
- SSA
  - a_1 := x + y
  - b_1 := a_1 – 1
  - a_2 := y + b_1
  - b_2 := x * 4
  - a_3 := a_2 + b_2

Merge Points

- The issue is how to handle merge points

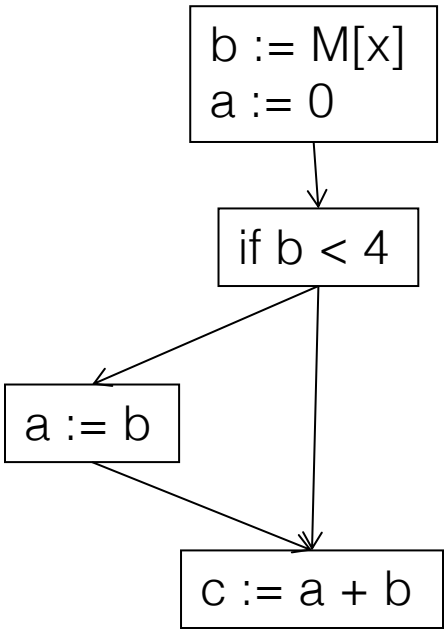


- Solution: introduce a Φ-function

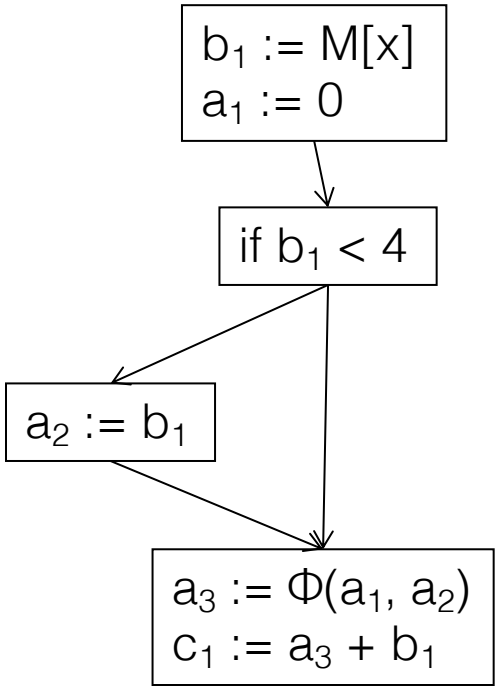
  $a_3 := \Phi(a_1, a_2)$
- Meaning: a_3 is assigned either a_1 or a_2 depending on which control path is used to reach the Φ-function

Another Example

Original



SSA

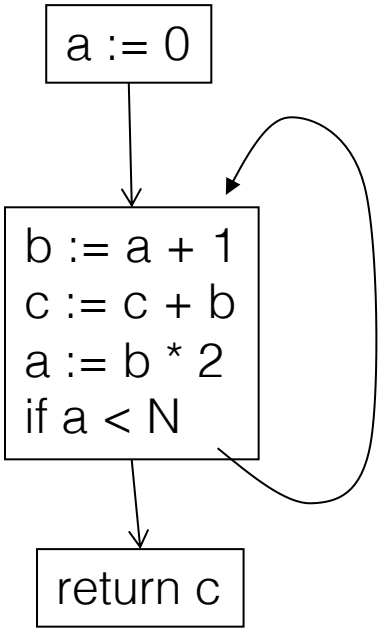


How Does Φ "Know" What to Pick?

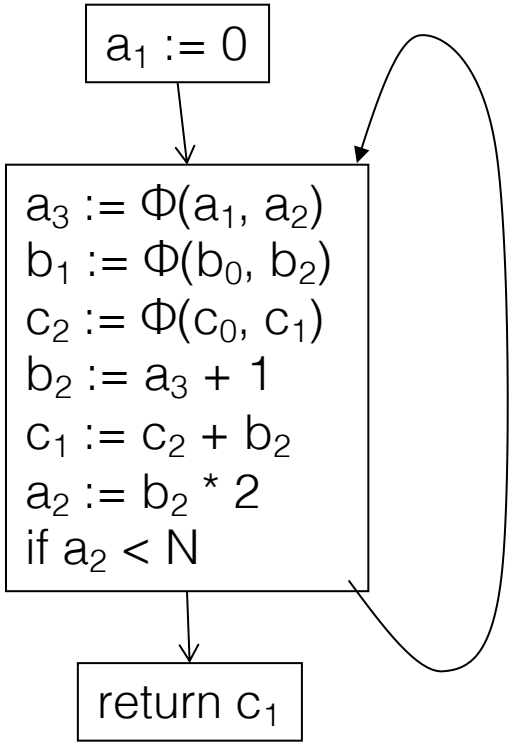
- It doesn't
- Φ-functions don't actually exist at runtime
  - When we're done using the SSA IR, we translate back out of SSA form, removing all Φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA, actual x
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to "execute" anything

Example With a Loop

Original



SSA



Notes:

- Loop back edges are also merge points, so require Φ-functions
- a_0, b_0, c_0 are initial values of a, b, c on block entry
- b_1 is dead – can delete later
- c is live on entry – either input parameter or uninitialized

What does SSA "get" us?

- No need for DU or UD chains – implicit in SSA
- Compact representation
- SSA is "recent" (i.e., 80s)
- Prevalent in real compilers for { } languages

Converting To SSA Form

- Basic idea
  - First, add Φ-functions
  - Then, rename all definitions and uses of variables by adding subscripts

Inserting Φ-Functions

- Could simply add Φ-functions for every variable at every join point(!)
- Called “maximal SSA”
- But
  - Wastes *way* too much space and time
  - Not needed in many cases

Path-convergence criterion

- Insert a Φ-function for variable a at point z when:
  - There are blocks x and y, both containing definitions of a, and $x \neq y$
  - There are nonempty paths from x to z and from y to z
  - These paths have no common nodes other than z

Details

- The start node of the flow graph is considered to define every variable (even if "undefined")
- Each Φ-function itself defines a variable, which may create the need for a new Φ-function
  - So we need to keep adding Φ-functions until things converge
- How can we do this efficiently?
  Use a new concept: dominance frontiers

Dominators - Review

- Definition: a block x *dominates* a block y if and only if every path from the entry of the control-flow graph to y includes x
- So, by definition, x dominates x

Dominators and SSA

- One property of SSA is that definitions dominate uses; more specifically:
  - If $x := \Phi(\ldots,x_i,\ldots)$ is in block B, then the definition of x_i dominates the i^{th} predecessor of B
  - If x is used in a non-Φ statement in block B, then the definition of x dominates block B

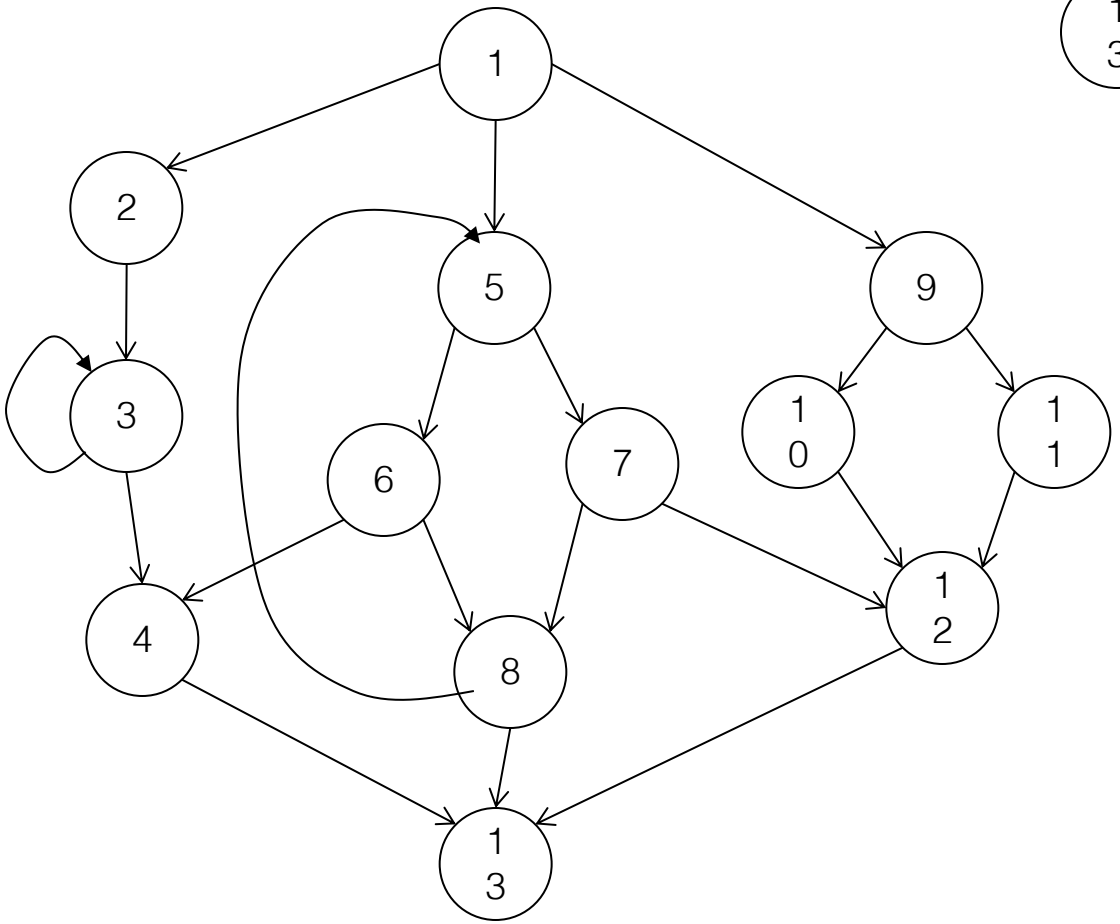
Dominance Frontier (1)

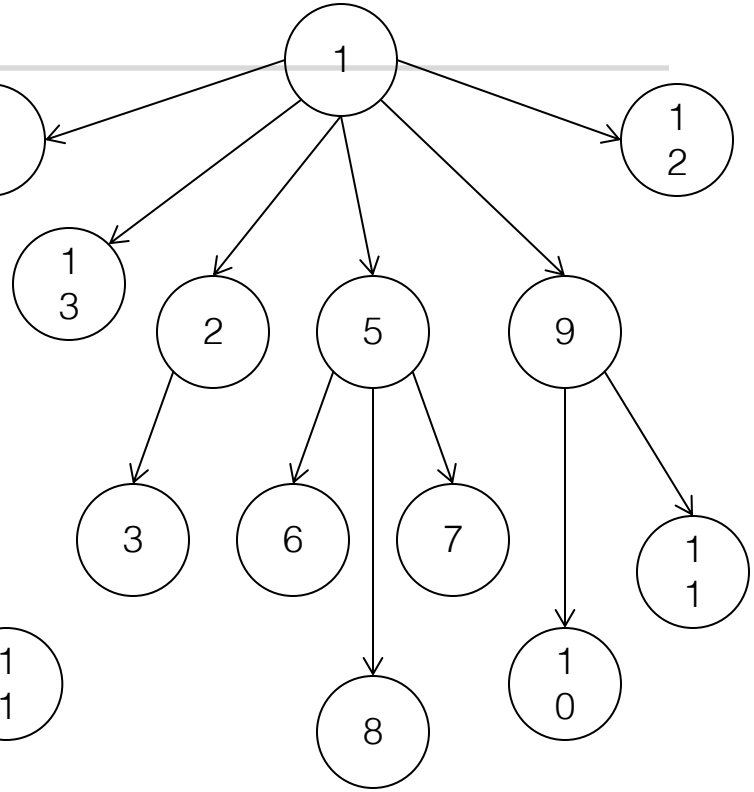
- To get a practical algorithm for placing Φ-functions, we need to avoid looking at all combinations of nodes leading from x to y
- Instead, use the dominator tree in the flow graph

Dominance Frontier (2)

- Definitions
  - x *strictly dominates* y if x dominates y and x ≠ y
  - The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but x does not strictly dominate w
    - This means that x can be in *it's own* dominance frontier! That can happen if there is a back edge to x (i.e., x is the head of a loop)
- Essentially, the dominance frontier is the border between dominated and undominated nodes

Example



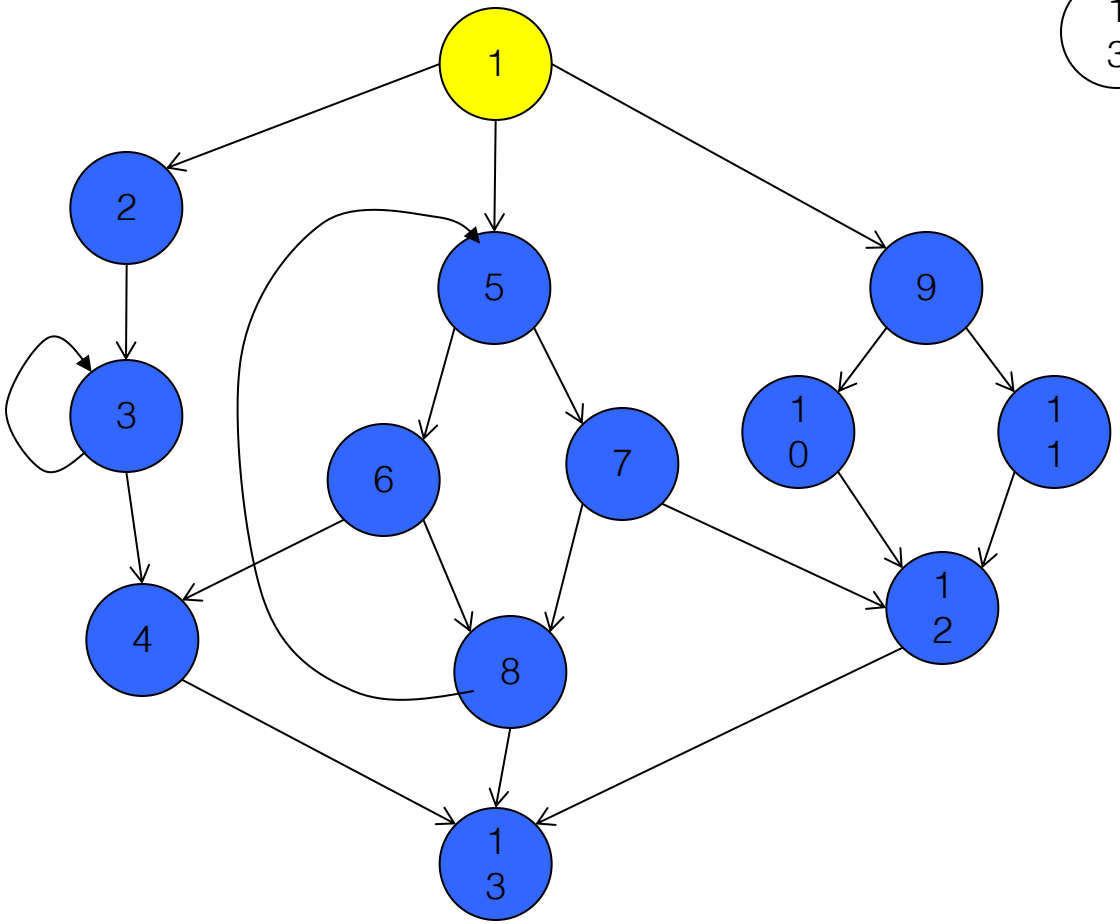


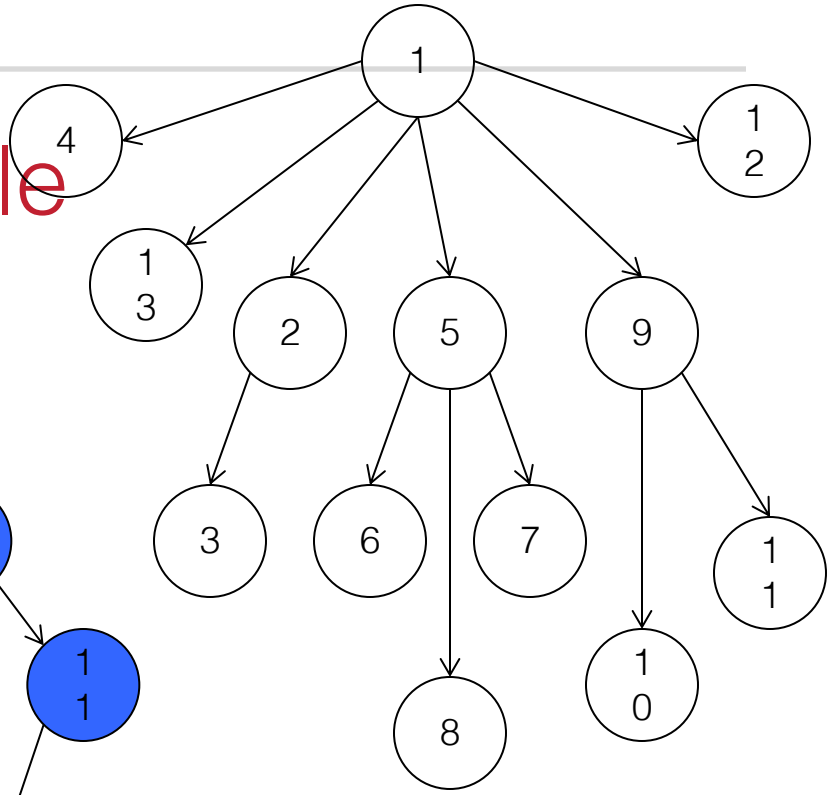
= x

= DomFrontier(x)

= StrictDom(x)

Example



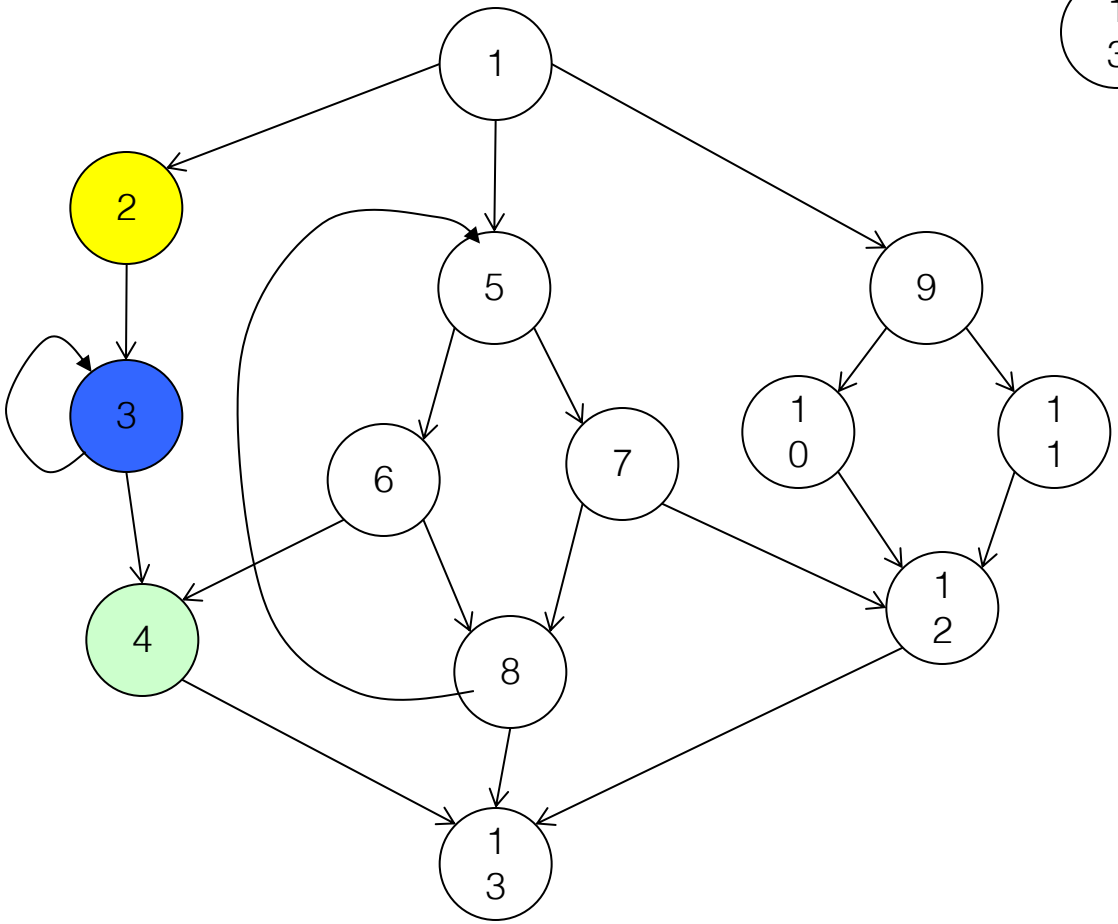


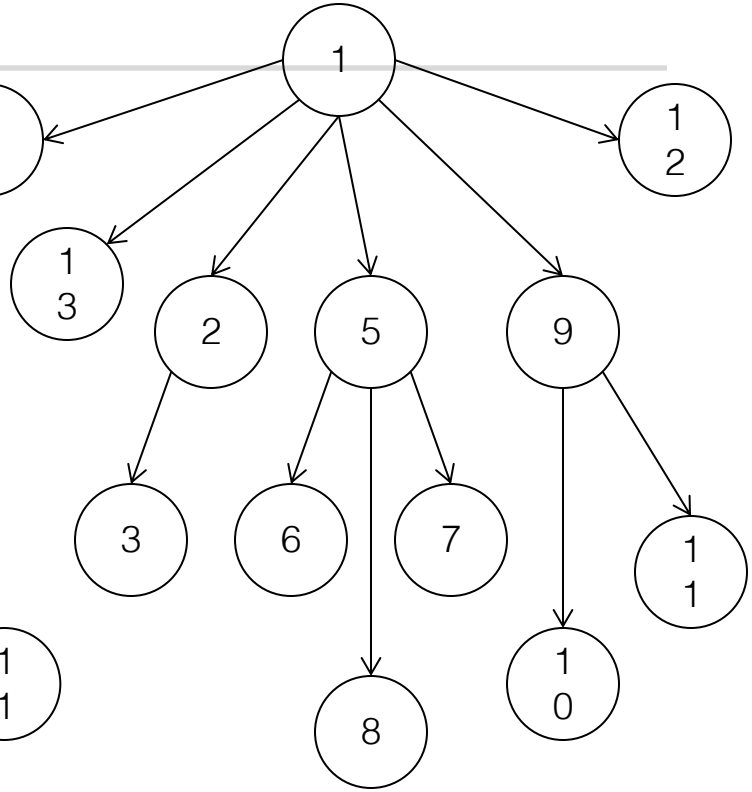
= x

= DomFrontier(x)

= StrictDom(x)

Example



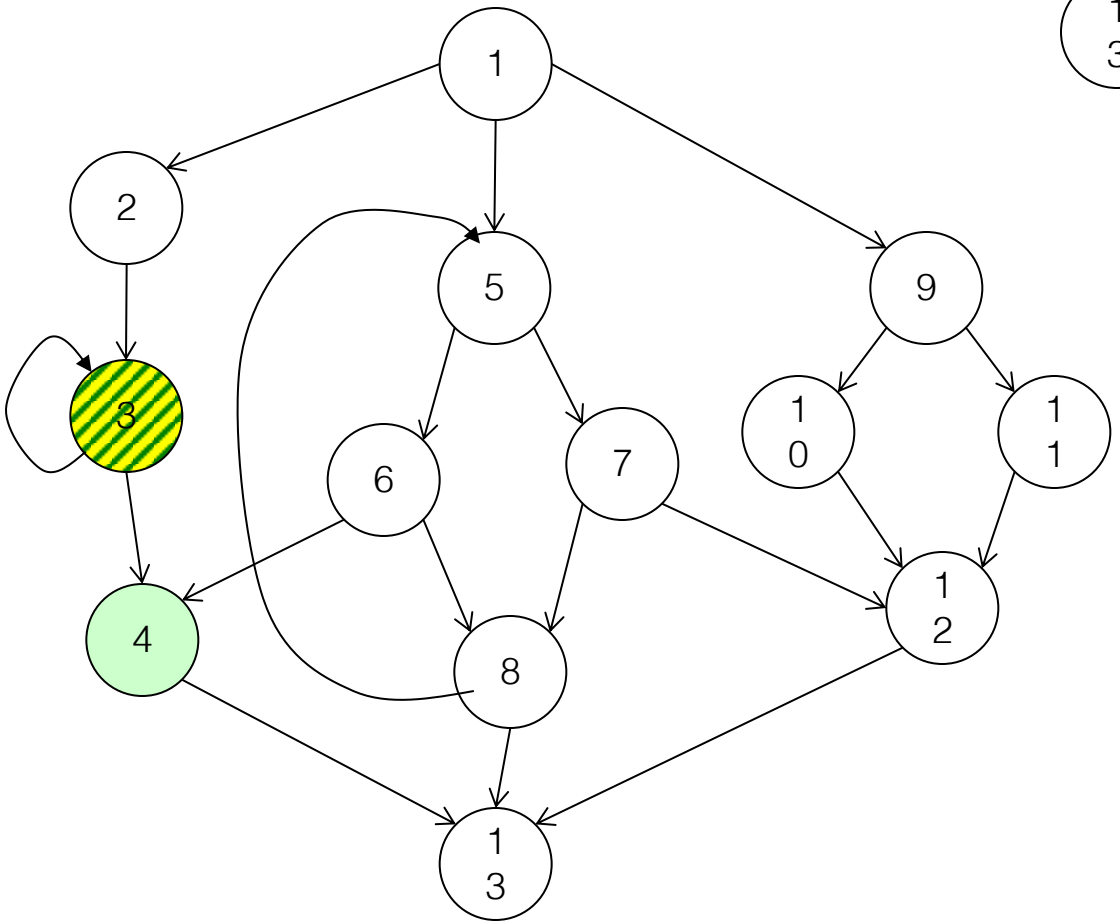


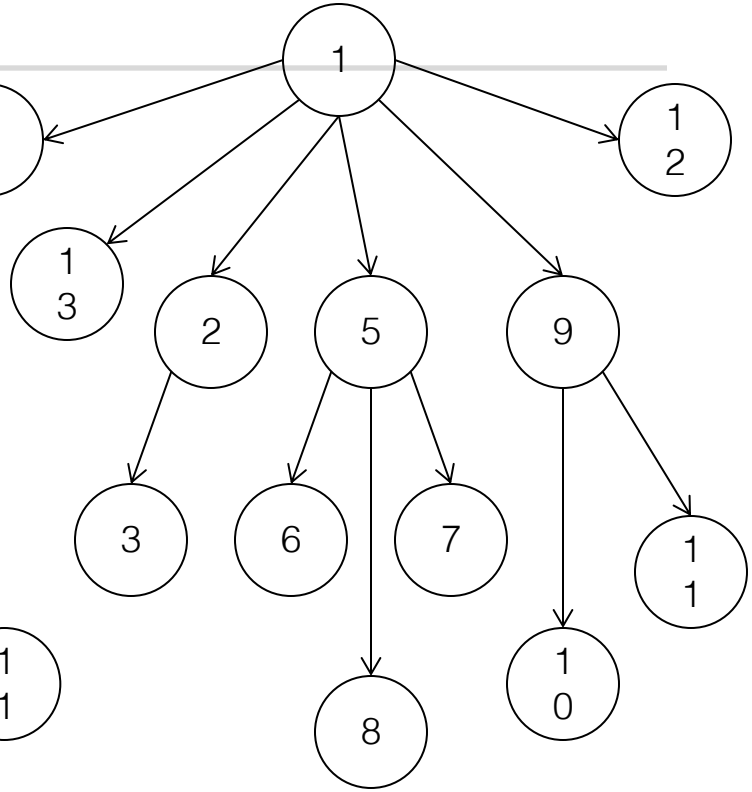
= x

= DomFrontier(x)

= StrictDom(x)

Example



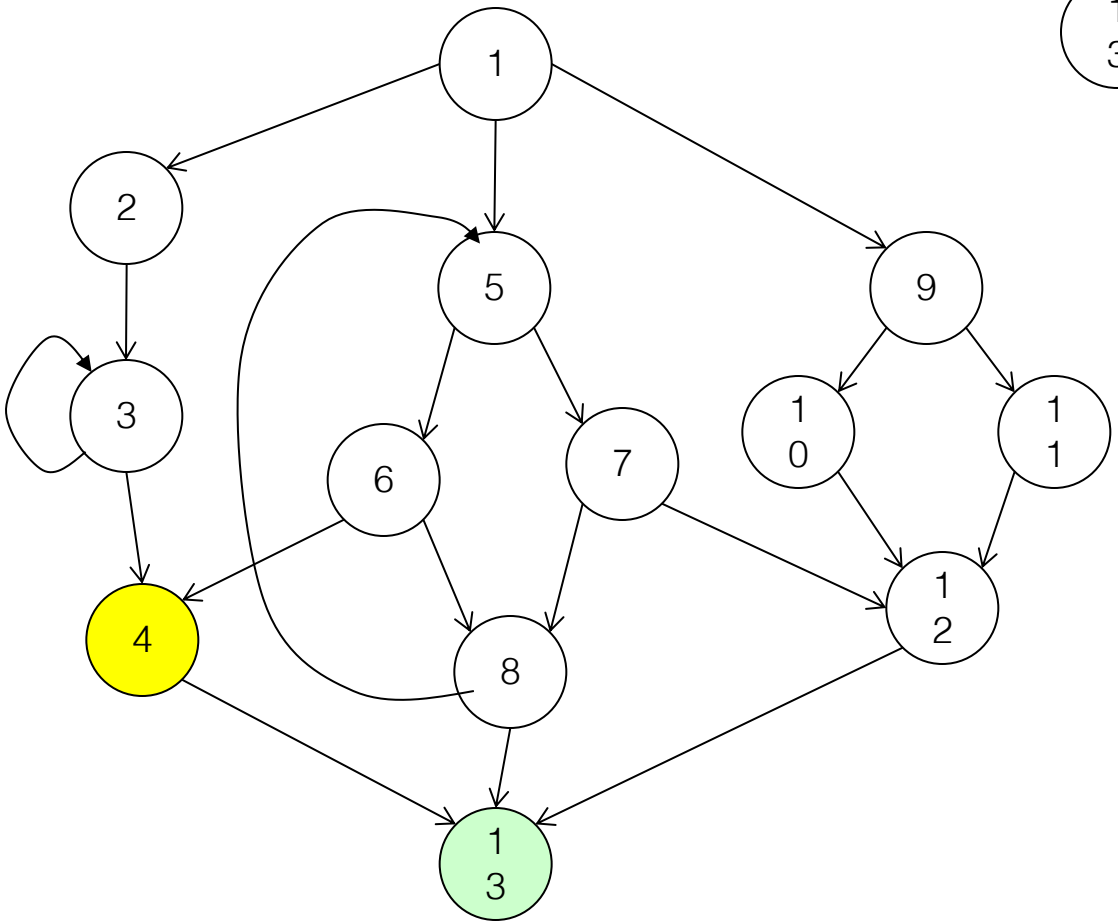


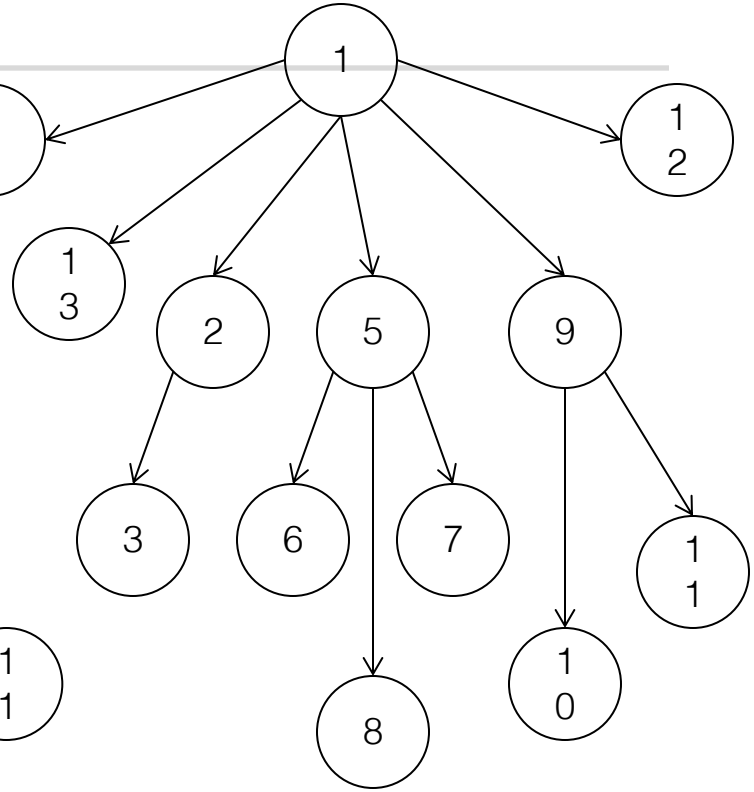
= x

= DomFrontier(x)

= StrictDom(x)

Example



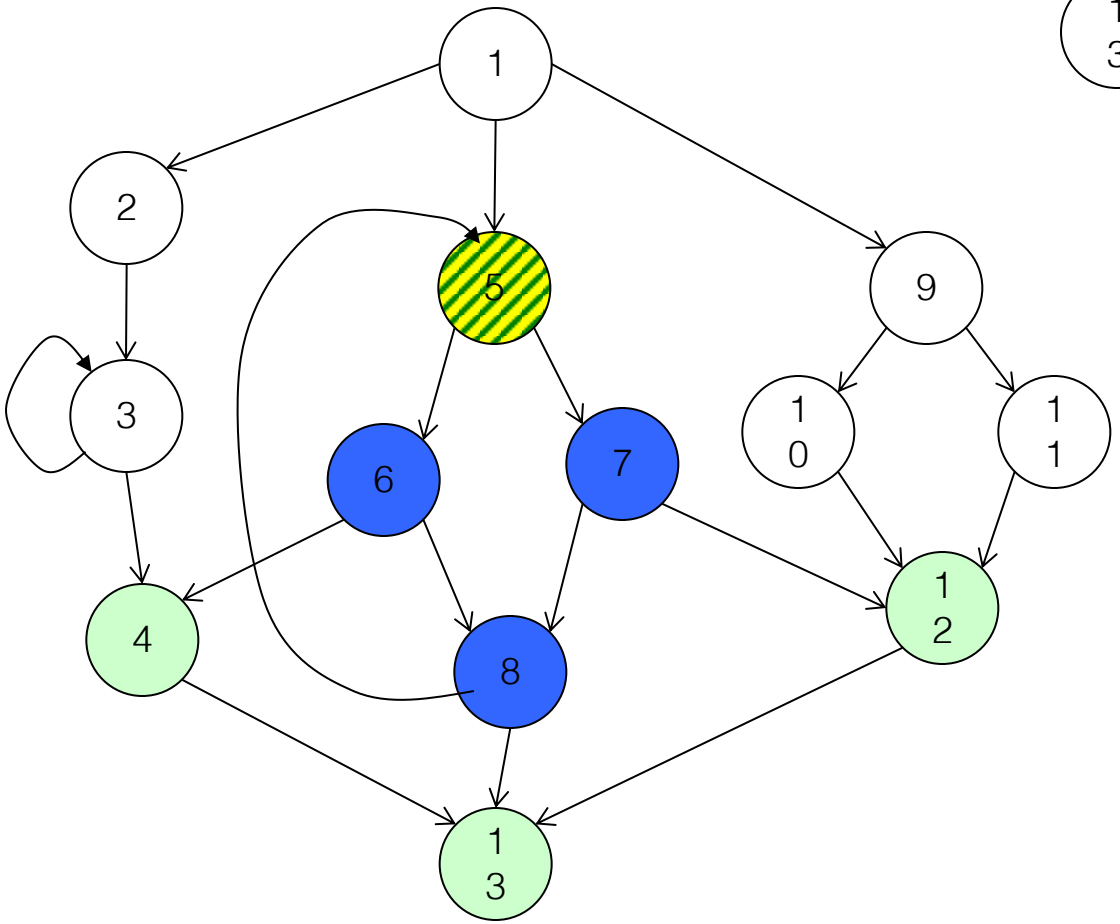


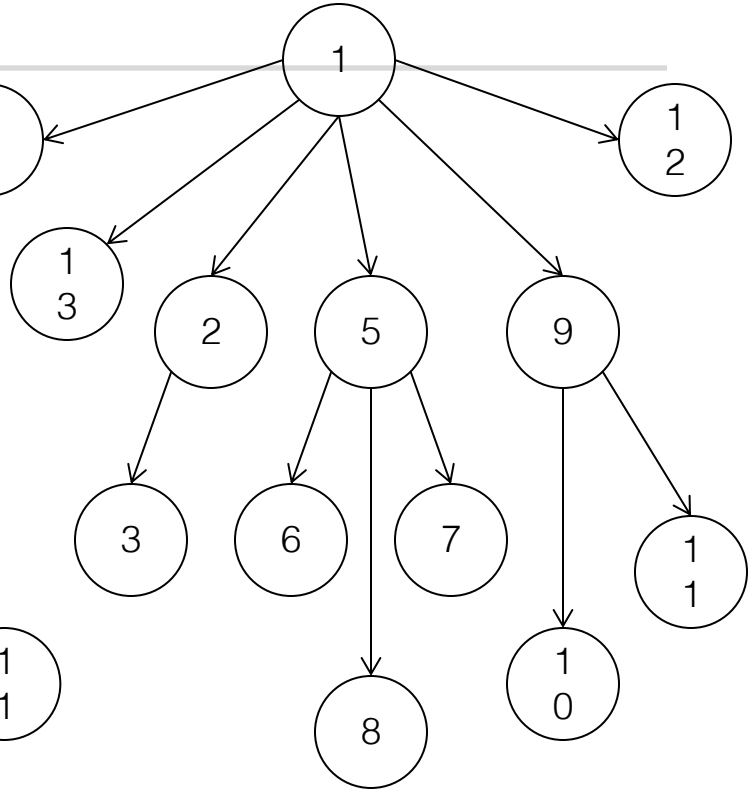
= x

= DomFrontier(x)

= StrictDom(x)

Example



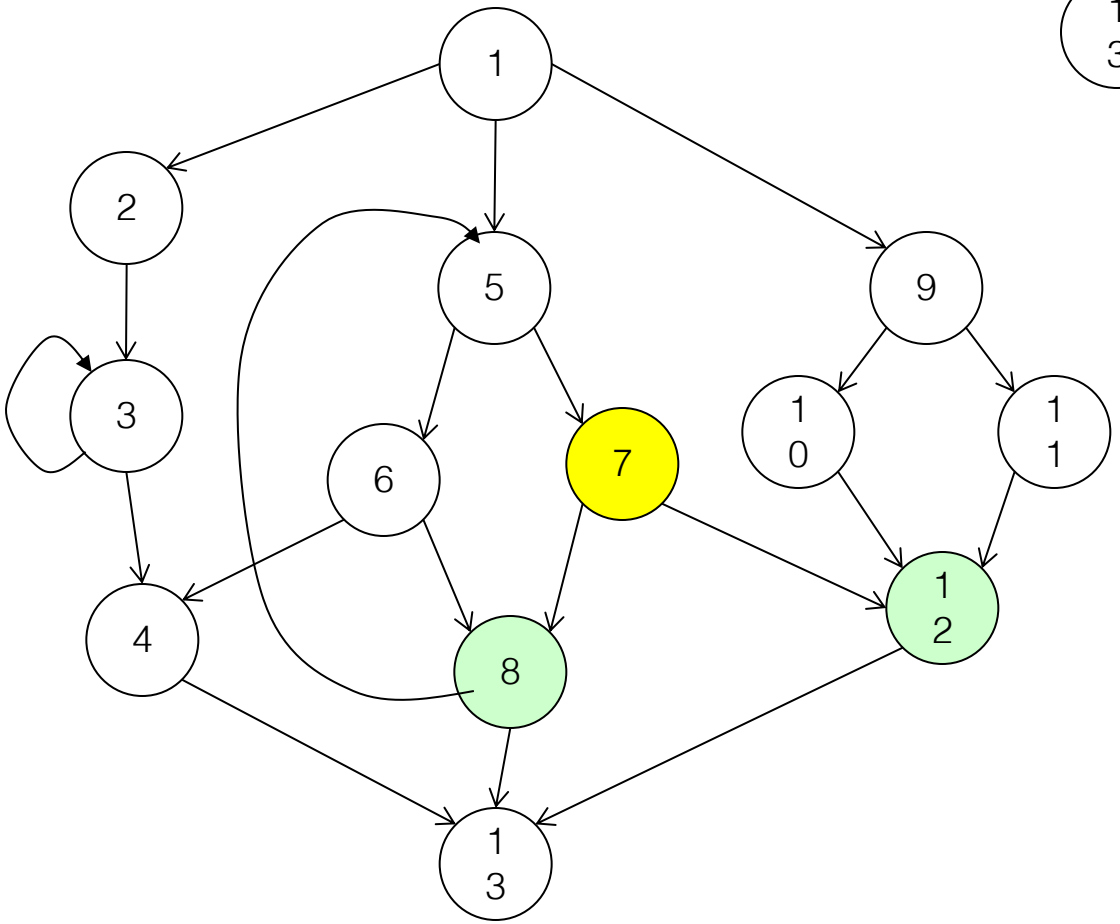


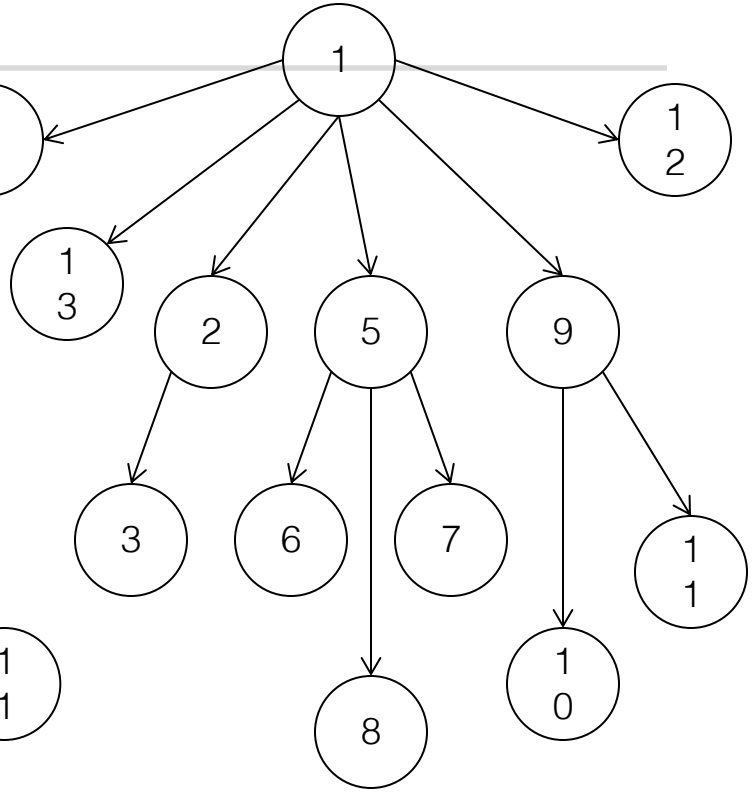
= X

= DomFrontier(x)

= StrictDom(x)

Example



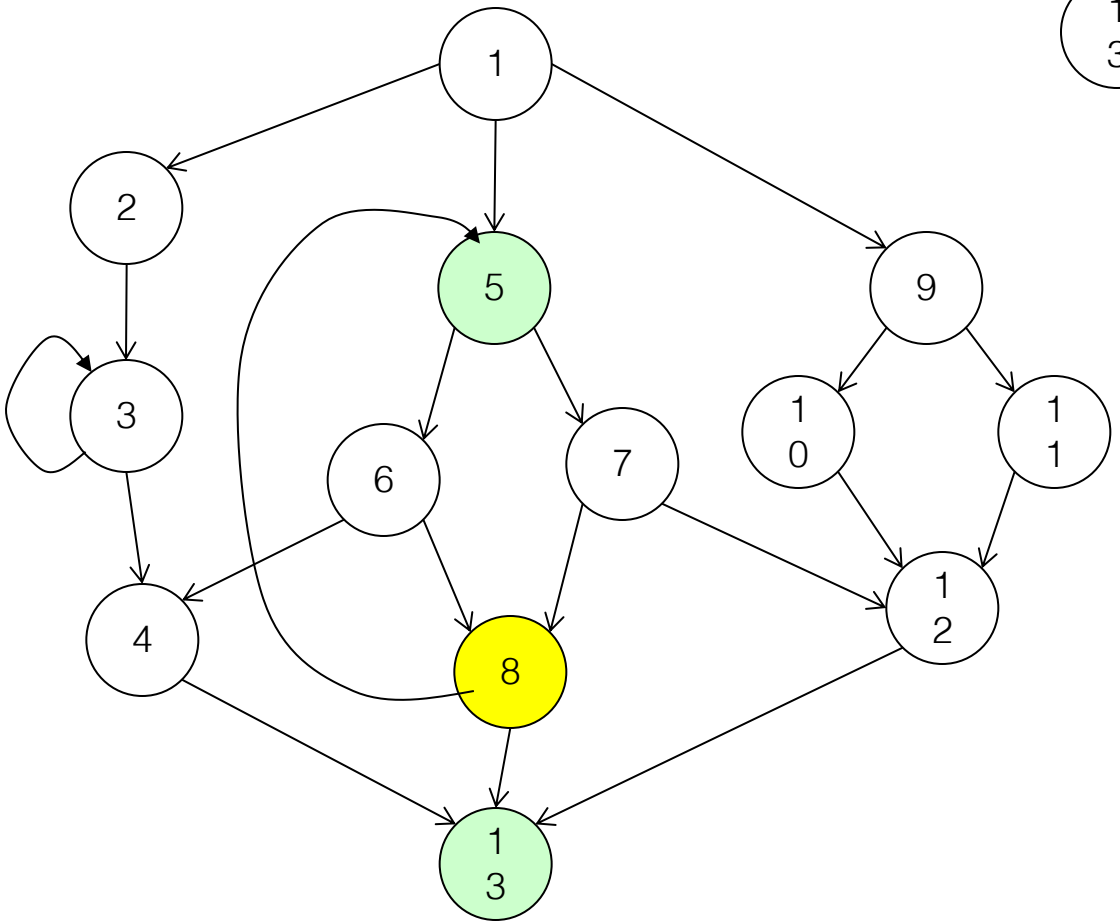


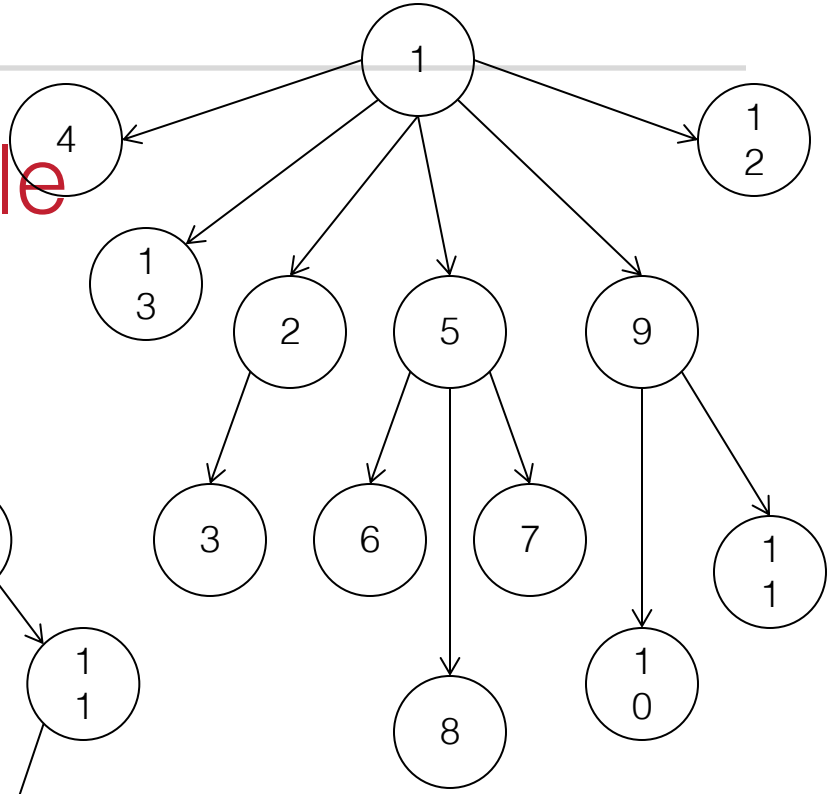
= x

= DomFrontier(x)

= StrictDom(x)

Example



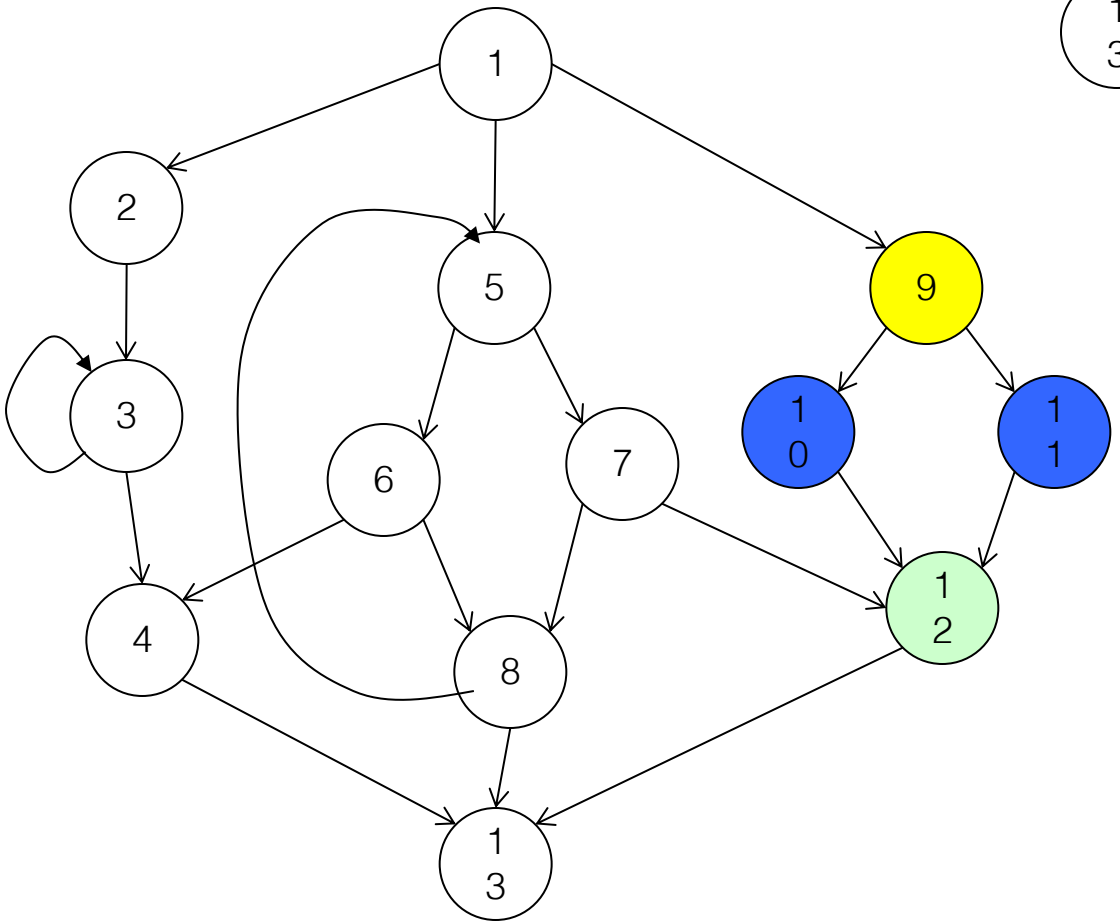


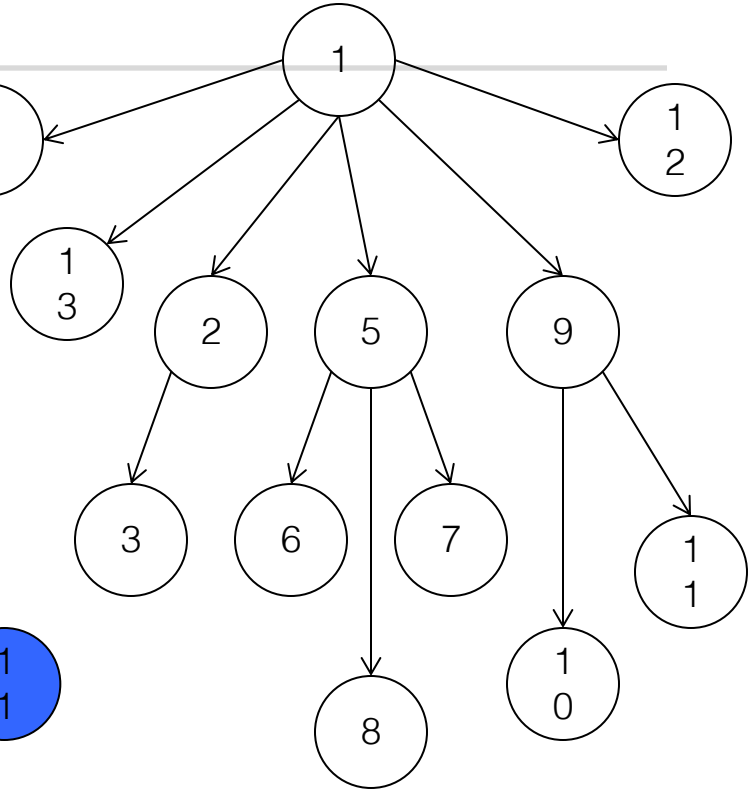
= x

= DomFrontier(x)

= StrictDom(x)

Example



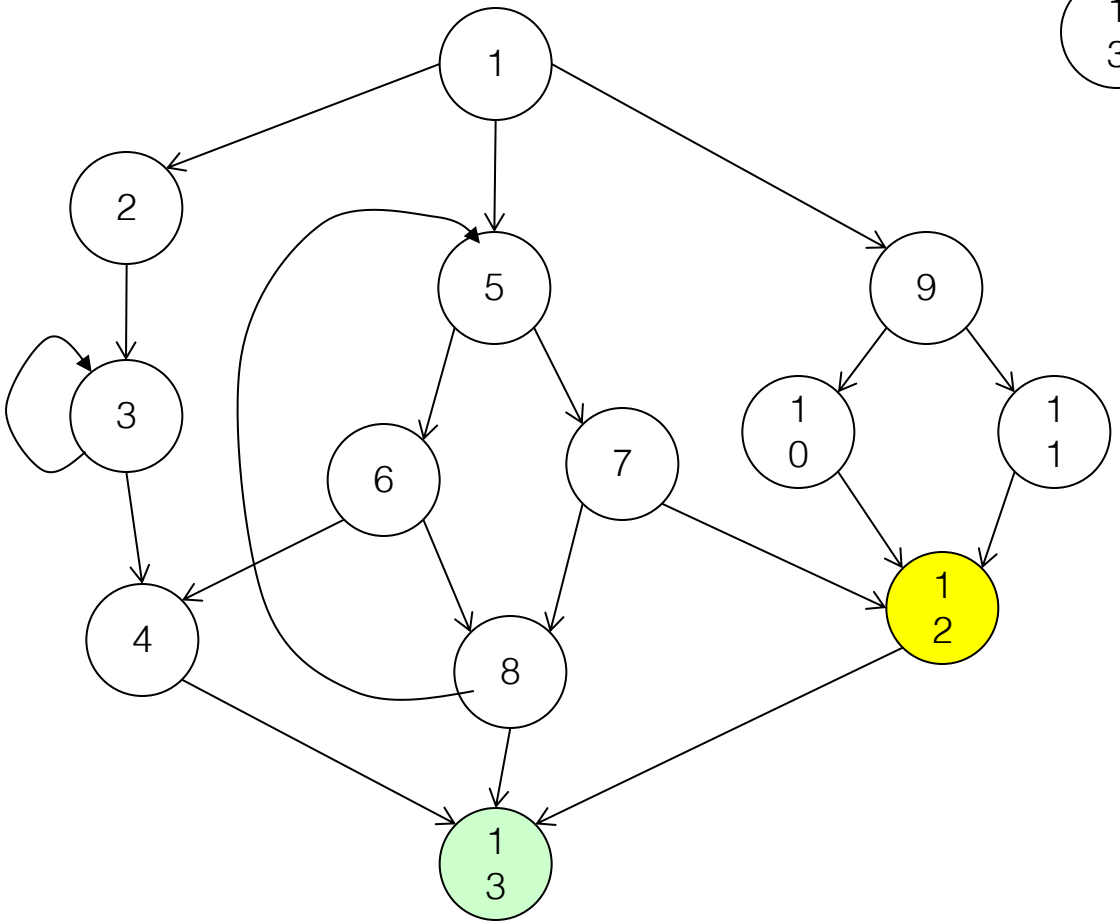


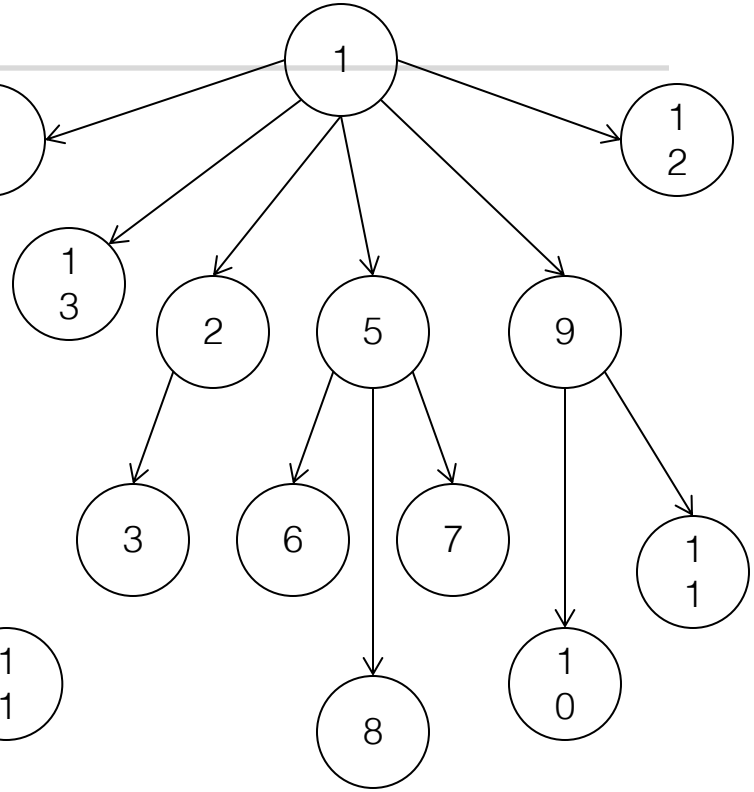
= X

= DomFrontier(x)

= StrictDom(x)

Example



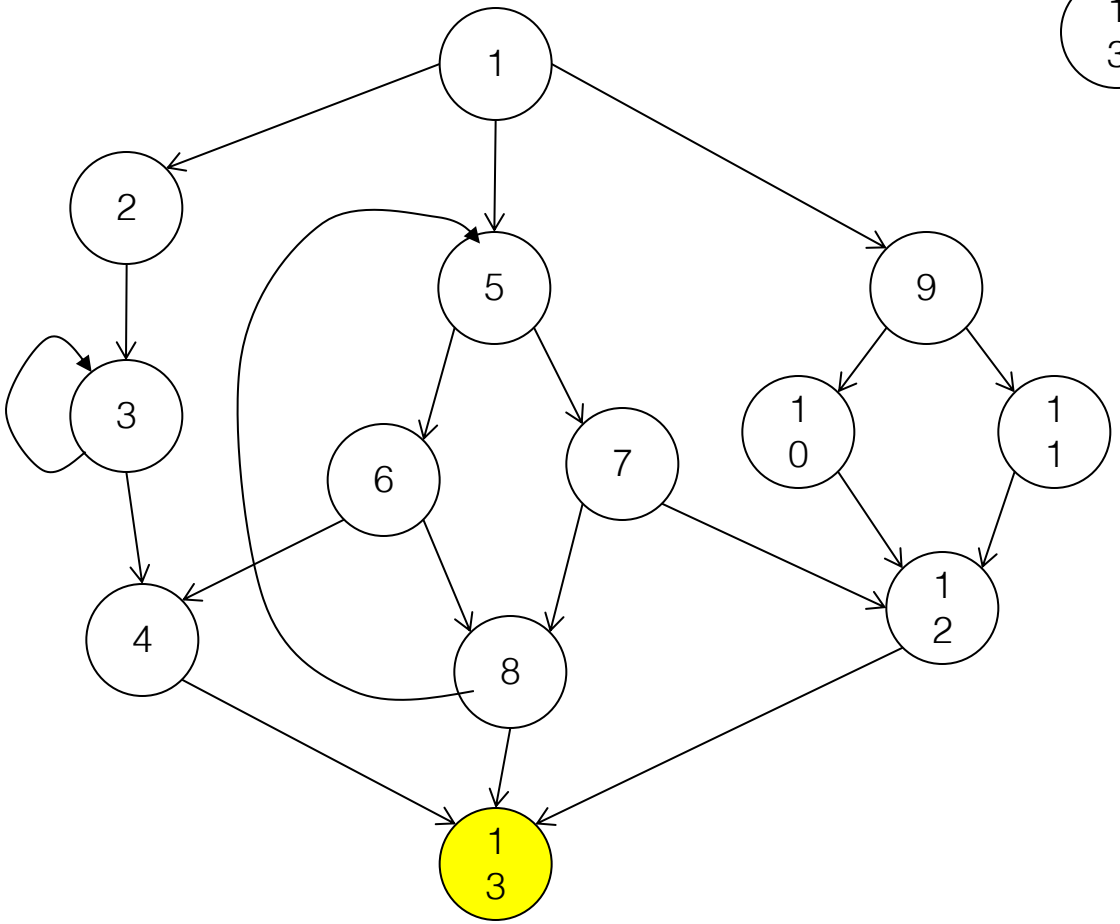


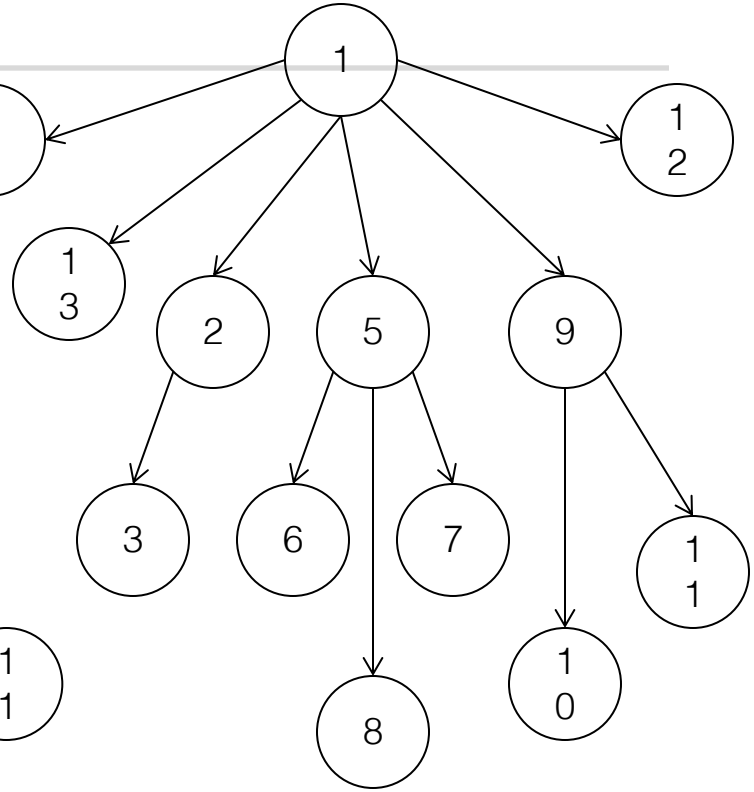
= x

= DomFrontier(x)

= StrictDom(x)

Example





= x

= DomFrontier(x)

= StrictDom(x)

Dominance Frontier Criterion for Placing Φ-Functions

- If a node x contains the definition of variable a, then every node in the dominance frontier of x needs a Φ-function for a
  - Idea: Everything dominated by x will see x's definition of a. The dominance frontier represents the first nodes we could have reached via an alternative path, which *will* have an alternate reaching definition (recall we say the entry node defines everything)
    - Why is this right for loops? Hint: strict dominance…
  - Since the Φ-function itself is a definition, this placement rule needs to be iterated until it reaches a fixed-point
- Theorem: this algorithm places exactly the same set of Φ-functions as the path criterion given previously

Placing Φ-Functions: Details

- See the book for the full construction, but the basic steps are:
  1. Compute the dominance frontiers for each node in the flowgraph
  2. Insert just enough Φ-functions to satisfy the criterion. Use a worklist algorithm to avoid reexamining nodes unnecessarily
  3. Walk the dominator tree and rename the different definitions of each variable a to be a_1, a_2, a_3, …

Efficient Dominator Tree Computation

- Goal: SSA makes optimizing compilers faster since we can find definitions/uses without expensive bit-vector algorithms
- So, need to be able to compute SSA form quickly
- Computation of SSA from dominator trees are efficient, but…

Lengauer-Tarjan Algorithm

- Iterative set-based algorithm for finding dominator trees is slow in worst case
- Lengauer-Tarjan is near linear time
  - Uses depth-first spanning tree from start node of control flow graph
  - See books for details

SSA Optimizations

- Why go to the trouble of translating to SSA?
- The advantage of SSA is that it makes many optimizations and analyses simpler and more efficient
  - We'll give a couple of examples
- But first, what do we know? (i.e., what information is kept in the SSA graph?)

SSA Data Structures

- Statement: links to containing block, next and previous statements, variables defined, variables used.
- Variable: link to its (single) definition and (possibly multiple) use sites
- Block: List of contained statements, ordered list of predecessors, successor(s)

Dead-Code Elimination

- A variable is live ⇔ its list of uses is not empty(!)
  - That's it! Nothing further to compute
- Algorithm to delete dead code:

  while there is some variable v with no uses
  if the statement that defines v has no
  other side effects, then delete it

  - Need to remove this statement from the list of uses for its operand variables – which may cause those variables to become dead

Sparse Simple Constant Propagation

- If c is a constant in v := c, any use of v can be replaced by c
  - Then update every use of v to use constant c
- If the c_i's in v := $\Phi(c_1, c_2, \ldots, c_n)$ are all the same constant c, we can replace this with v := c
- Incorporate copy propagation, constant folding, and others in the same worklist algorithm

Simple Constant Propagation

```
W := list of all statements in SSA program
while W is not empty
   remove some statement S from W
   if S is v:=Φ(c, c, …, c), replace S with v:=c
   if S is v:=c
      delete S from the program
      for each statement T that uses v
         substitute c for v in T
         add T to W
```

Converting Back from SSA

- Unfortunately, real machines do not include a Φ instruction
- So after analysis, optimization, and transformation, need to convert back to a "Φ-less" form for execution

Translating Φ-functions

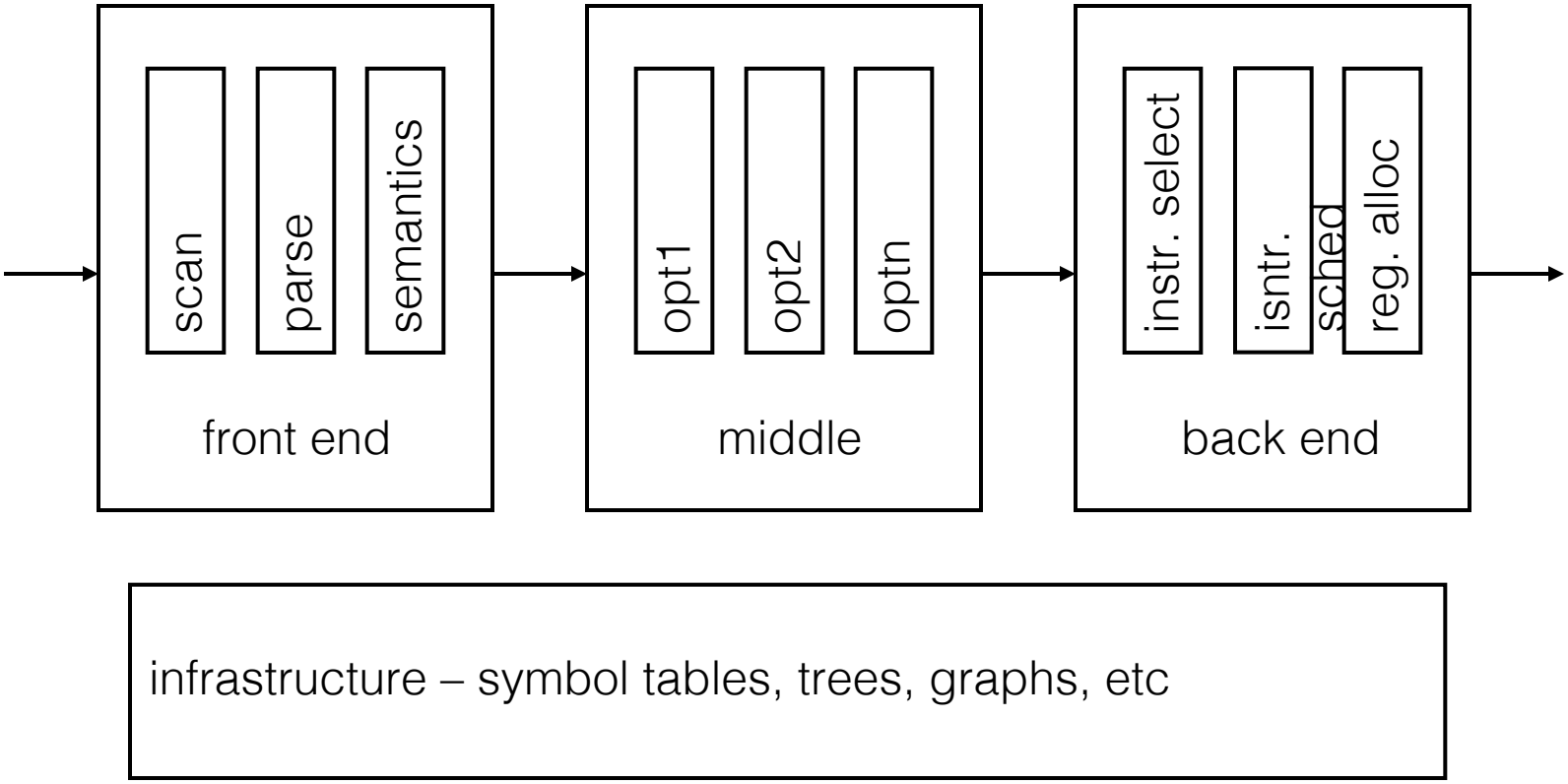
- The meaning of $x := \Phi(x_1, x_2, \ldots, x_n)$ is "set $x := x_1$ if arriving on edge 1, set $x := x_2$ if arriving on edge 2, etc."
- So, for each i, insert $x := x_i$ at the end of predecessor block i
- Rely on copy propagation and coalescing in register allocation to eliminate redundant copy instructions

SSA Wrapup

- More details needed to fully and efficiently implement SSA, but these are the main ideas
  - See recent compiler books (but not the new Dragon book!)
- Allows efficient implementation of many optimizations
- SSA is used in most modern optimizing compilers (llvm is based on it) and has been retrofitted into many older ones (gcc is a well-known example)
- Not a silver bullet – some optimizations still need non-SSA forms, but very effective for many

Instruction Selection

Compiler Organization



Big Picture

- Compiler consists of lots of fast stuff followed by hard problems
  - Scanner: O(n)
  - Parser: O(n)
  - Analysis & Optimization: ~ O(n log n)
  - Instruction selection: fast or NP-Complete
  - Instruction scheduling: NP-Complete
  - Register allocation: NP-Complete

IR for Code Generation

- Assume a low-level RISC-like IR
  - 3 address, register-register instructions + load/store

    r1 <- r2 op r3
  - Could be tree structure or linear
  - Expose as much detail as possible
- Assume "enough" (i.e., ∞) registers
  - Invent new temporaries for intermediate results
  - Map to actual registers later

Overview: Instruction Selection

- Map IR into assembly code
- Assume known storage layout and code shape
  - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions (take advantage of addressing modes, etc.)

Overview: Instruction Scheduling

- Reorder operations to hide latencies – processor function units; memory/cache
  - Originally invented for supercomputers (1960s)
  - Now important everywhere
    - Even non-RISC machines, i.e., x86
    - Even if processor reorders on the fly
- Assume fixed program

Overview: Register Allocation

- Map values to actual registers
  - Previous phases change need for registers
- Add code to spill values to temporaries as needed, etc.
- Usually worth doing another instruction scheduling pass afterwards if spill code inserted

How Hard?

- Instruction selection
  - Can make locally optimal choices
  - Global is undoubtedly NP-Complete
- Instruction scheduling
  - Single basic block – quick heuristics
  - General problem – NP Complete
- Register allocation
  - Single basic block, no spilling, interchangeable registers – linear
  - General – NP Complete

Conventional Wisdom

- We typically lose little by solving these independently
  - But not always, of course (iterating phases on x86[-64] can help because of limited registers, memory operands)
- Instruction selection
  - Use some form of pattern matching
  - ∞ virtual registers – create as needed
- Instruction scheduling
  - Within a block, list scheduling is close to optimal
  - Across blocks: EBBs or trace scheduling if list scheduling not good enough
- Register allocation
  - Start with unlimited virtual registers and map "enough" to K

An Simple Low-Level IR (1)

- Details not important for our purposes; point is to get a feeling for the level of detail involved
  - This example is from Appel
- Expressions
  - CONST(i) – integer constant i
  - TEMP(t) – temporary t (i.e., register)
  - BINOP(op,e1,e2) – application of op to e1,e2
  - MEM(e) – contents of memory at address e
    - Means value when used in an expression
    - Means address when used on left side of assignment
  - CALL(f,args) – apply function f to argument list args

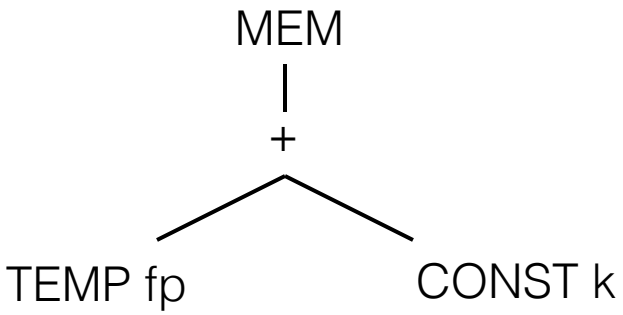
Simple Low-Level IR (2)

- Statements
  - MOVE(TEMP t, e) – evaluate e and store in temporary t
  - MOVE(MEM(e1), e2) – evaluate e1 to yield address a; evaluate e2 and store at a
  - EXP(e) – evaluate expressions e and discard result
  - SEQ(s1,s2) – execute s1 followed by s2
  - NAME(n) – assembly language label n
  - JUMP(e) – jump to e, which can be a NAME label, or more complex (e.g., switch)
  - CJUMP(op,e1,e2,t,f) – evaluate e1 op e2; if true jump to label t, otherwise jump to f
  - LABEL(n) – defines location of label n in the code

Low-Level IR Example (1)

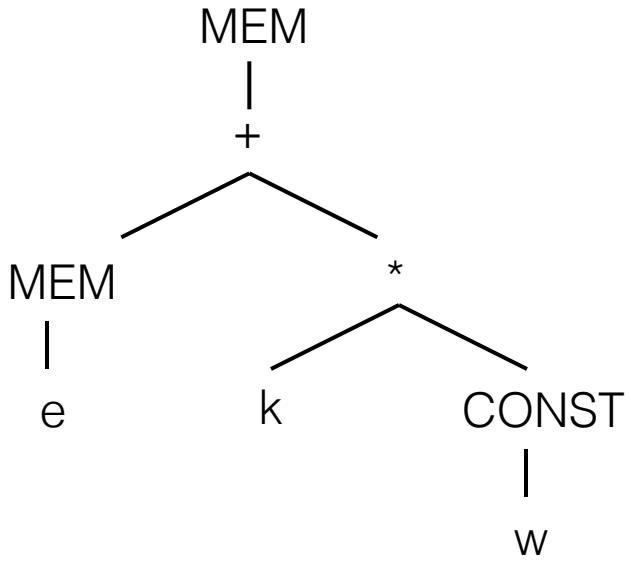
- Expr for a local variable at a known offset k from the frame pointer fp
  - Linear

    MEM(BINOP(PLUS, TEMP fp, CONST k))
  - Tree



Low-Level IR Example (2)

- For an array element e(k), where each element takes up w storage locations



Generating Low-Level IR

- Assuming initial IR is an AST, a simple treewalk can be used to generate the low-level IR
  - Can be done before, during, or after optimizations in the middle part of the compiler
    - Typically AST is lowered to some lower-level IR, but maybe not final lowest-level one used in instruction selection
- Create registers (temporaries) for values and intermediate results
  - Value can be safely allocated to a register when only 1 name can reference it
    - Trouble: pointers, arrays, reference parameters
  - Assign a virtual register to anything that can go into one
  - Generate loads/stores for other values

Instruction Selection Issues

- Given the low-level IR, there are many possible code sequences that implement it correctly
  - e.g. to set %rax to 0 on x86, among others…

```
movq  $0,%rax        xorq  %rax,%rax
subq   %rax,%rax     imulq  %rax,0
```

  - Many machine instructions do several things at once – e.g., register arithmetic and effective address calculation

```
leaq    offset(%rbase,%rindex,scale),%rdst
```

Instruction Selection Criteria

- Several possibilities:
  - Fastest
  - Smallest
  - Minimize power consumption (ex: don't use a function unit if leaving it powered-down is a win)
- Sometimes not obvious
  - e.g., if one of the function units in the processor is idle and we can select an instruction that uses that unit, it effectively executes for free, even if that instruction wouldn't be chosen normally
    - (Some interaction with scheduling here…)

Implementation

- Problem: We need some representation of the target machine instruction set that facilitates code generation
- Idea: Describe machine instructions using same low-level IR used for program
- Use pattern matching techniques to pick machine instructions that match fragments of the program IR tree
  - Want this to run quickly
  - Would like to automate as much as possible

Matching: How?

- Tree IR – pattern match on trees
  - Tree patterns as input
  - Each pattern maps to target machine instruction (or sequence)
    - and at least one simple target match for each kind of tree node so we can always generate something
  - Various algorithms – max. munch, dynamic programming, …
- Linear IR – some sort of string matching
  - Strings as input
  - Each string maps to target machine instruction sequence
  - Use text matching or peephole matching (parsing!, but with a way, way ambiguous grammar for target machine)
- Both work well in practice; algorithms are quite different

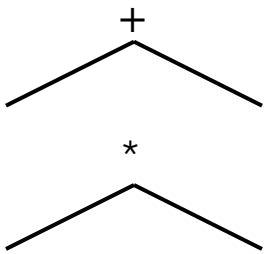
An Example Target Machine (1)

- Arithmetic Instructions – result in reg.

  (unnamed) ri TEMP

  ADD ri <- rj + rk

  MUL ri <- rj * rk

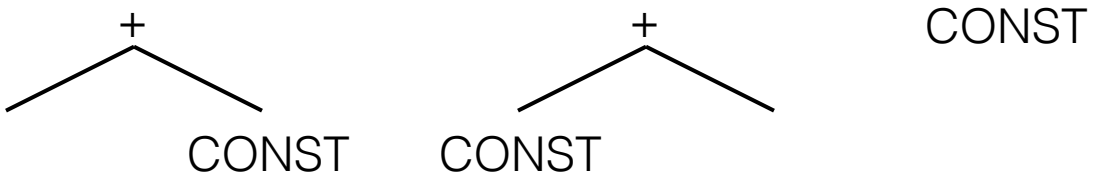


  SUB and DIV are similar

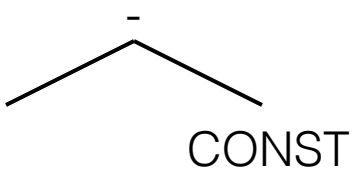
An Example Target Machine (2)

- Immediate Instructions

ADDI ri <- rj + c



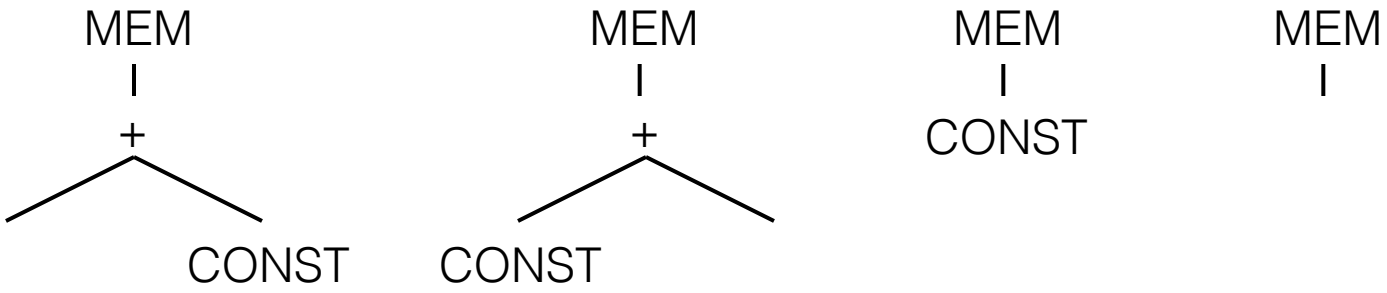
SUBI ri <- rj - c



An Example Target Machine (3)

- Load

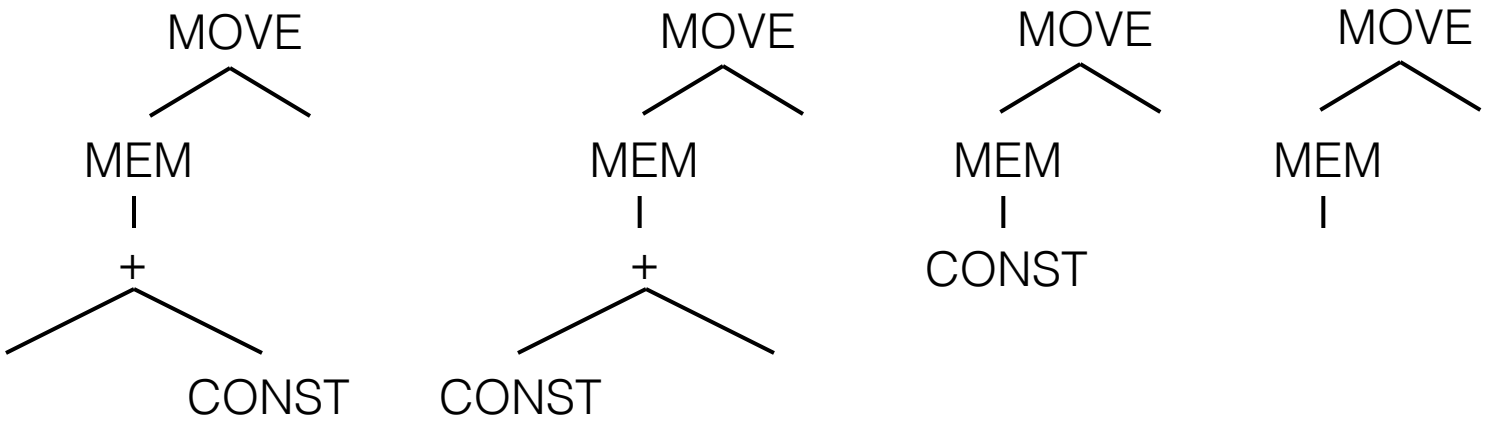
  LOAD ri <- M[rj + c]



An Example Target Machine (4)

- Store – not an expression; no result

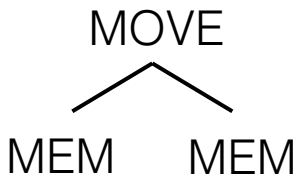
  STORE M[rj + c] <- ri



An Example Target Machine (5)

- mem->mem copy – also not an expression

  MOVEM M[rj] <- M[ri]



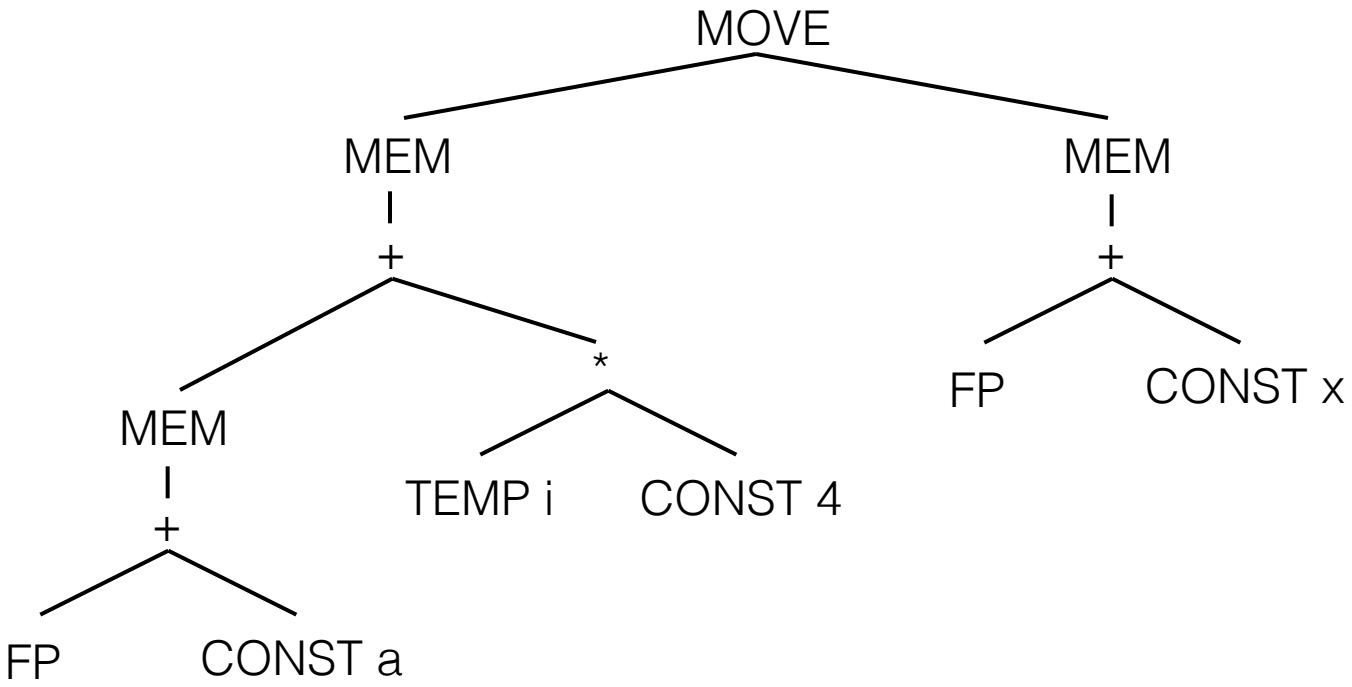
Tree Pattern Matching (1)

- Goal: Tile the low-level tree with operation (instruction) trees
- A *tiling* is a collection of <node,op> pairs
  - node is a node in the tree
  - op is an operation tree
  - <node,op> means that op could implement the subtree at node

Tree Pattern Matching (2)

- A tiling "implements" a tree if it covers every node in the tree and the overlap between any two tiles (trees) is limited to a single node
  - If <node,op> is in the tiling, then node is also covered by a leaf in another operation tree in the tiling – unless it is the root
  - Where two operation trees meet, they must be compatible (i.e., expect the same value in the same location)

Example – Tree for a[i]:=x



Generating Code

- Given a tiled tree, to generate code
  - Postorder treewalk; node-dependant order for children
  - Emit code sequences corresponding to tiles in order
  - Connect tiles by using same register name to tie boundaries together

Tiling Algorithms (1)

- Maximal Munch
  - Start at root of tree, find largest tile that fits. Cover the root node and possibly other nearby nodes. Then repeat for each subtree
  - Generate instruction as each tile is placed
    - Generates instructions in reverse order
  - Generates an optimum tiling – tiles sum to lowest possible cost
    - But not optimal – there may be another tiling where two adjacent tiles can be combined into one of lower cost

Tiling Algorithms (2)

- Dynamic Programming
  - There may be many tiles that could match at a particular node
  - Idea: Walk the tree and accumulate the set of all possible tiles that could match at that point – Tiles(n)
  - Then: Select minimal cost for subtrees (bottom up), and go top-down to select and emit lowest-cost instructions

Tile(Node n)

```
Tiles(n) <- empty;
if n has two children then
     Tile(left child of n)
     Tile(right child of n)
     for each rule r that implements n
       if (left(r) is in Tiles(left(n)) and right(r) is in Tiles(right(n)))
       Tiles(n) <- Tiles(n) + r
else if n has one child then
     Tile(child of n)
     for each rule r that implements n
        if(left(r) is in Tiles(child(n)))
       Tiles(n) <- Tiles(n) + r
else /* n is a leaf */
     Tiles(n) <- { all rules that implement n }
```

Tools

- iburg and burg and others use a combination of dynamic programming and tree pattern matching to find optimal translations
- Product of years of research going back to peephole optimizers
- Not really a research area now (like parsing), but still room for newer/better tools

Peephole Optimization

- Idea: Find pairs of instructions (not necessarily adjacent) and replace them with something better
- Instead of

```
t2 = &c
t3 = *t2
```

use

```
t3 = c
```

Which Pairs?

- Chris Fraser noticed that useful pairs were connected by UD chains, so
- Plan: consider each instruction together with instructions that feed it
- UD chains reach across blocks, so instruction selector can work globally
  - Works great with SSA too

Patterns

- Reduce pairs of instructions to schematics

  t5 = 4

  t6 = t4 + t5

  becomes

  %reg1 = %con

  %reg3 = %reg2 + %reg1

- Find in hash table; if found, replace

  %reg3 = %reg2 + %con

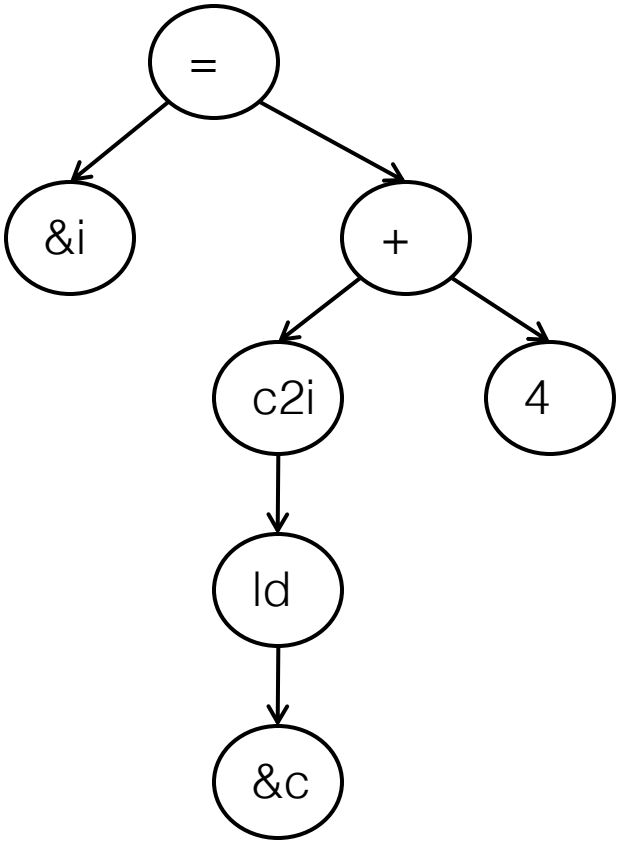
Tree-Based Representation

- The tree makes the UD chains explicit
- Each definition in the tree is used once
- Typically a basic block would have a sequence of expression trees

Example

Code for i = c + 4
[c a char; i an int]

```
t1 = &i
t2 = &c
t3 = *t2
t4 = cvci(t3)
t5 = 4
t6 = t4+t5
*t1 = t6
```



Tree Patterns

- An iburg spec is a set of rules. BNF:

  rule = nonterm : tree = integer (cost)

  tree = term (tree , tree)

  | term (tree)

  | term

  | nonterm

- Terminals are IL operations
- Nonterminals name sets of rules

Rules

- Rules are numbered to the right of the = sign
- The cost of a rule is its cost (often 0) plus the cost of any subtrees
- A rule may be nested to define a pattern that matches more than one level in a tree

Example

```
stmt    : ASSIGN(addr, reg) = 1 (1)
stmt    : reg = 2 (0)
reg     : ADD(reg, rc) = 3 (1)
reg         : ADD(rc, reg) = 4 (1)
reg         : LD(addr) = 5 (1)
reg         : C2I(LD(addr)) = 6 (1)
reg         : addr = 7 (1)
reg         : con = 8 (1)
addr    : ADD(reg, con) = 9 (0)
addr    : ADD(con, reg) = 10 (0)
addr    : ADDRLP = 11 (0)   // addr of local var
rc      : con = 12 (0)
rc      : reg = 13 (0)
con     : CNST = 14 (0)
```

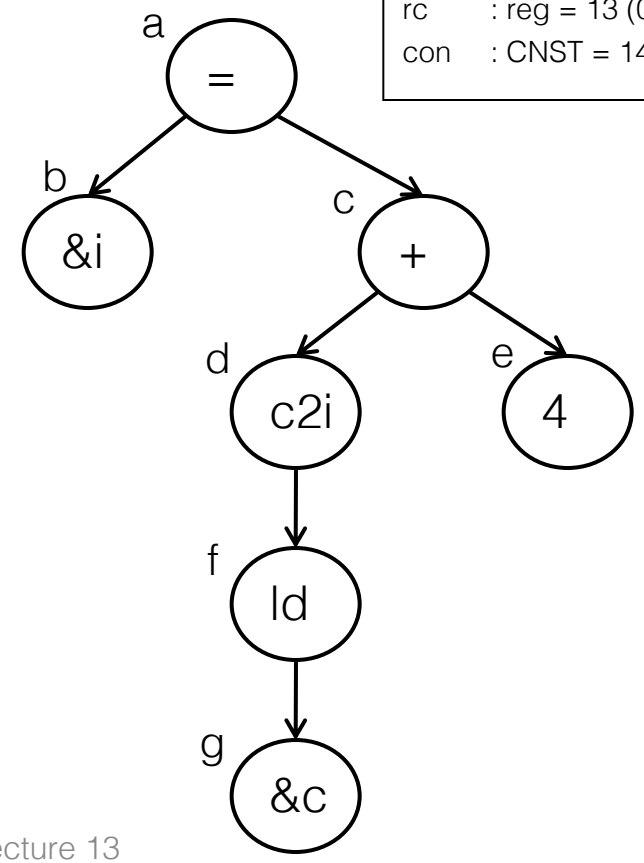
Algorithm

- Pass I: bottom up
  - Label each node with lowest cost rule to produce result from available operands (cost = cost of rule + cost of subtrees)
  - Label is: (r, c) = best is rule r, cost is c
- Pass II: top down
  - Find cheapest node that generates result needed by parent tree
  - Emit instructions in reverse order as choices are made

Bottom-up (Labeling)

```
stmt : ASSIGN(addr, reg) = 1 (1)
stmt : reg = 2 (0)
reg  : ADD(reg, rc) = 3 (1)
reg  : ADD(rc, reg) = 4 (1)
reg  : LD(addr) = 5 (1)
reg  : C2I(LD(addr)) = 6 (1)
reg  : addr = 7 (1)
reg  : con = 8 (1)
addr : ADD(reg, con) = 9 (0)
addr : ADD(con, reg) = 10 (0)
addr : ADDRLP = 11 (0)
rc   : con = 12 (0)
rc   : reg = 13 (0)
con  : CNST = 14 (0)
```

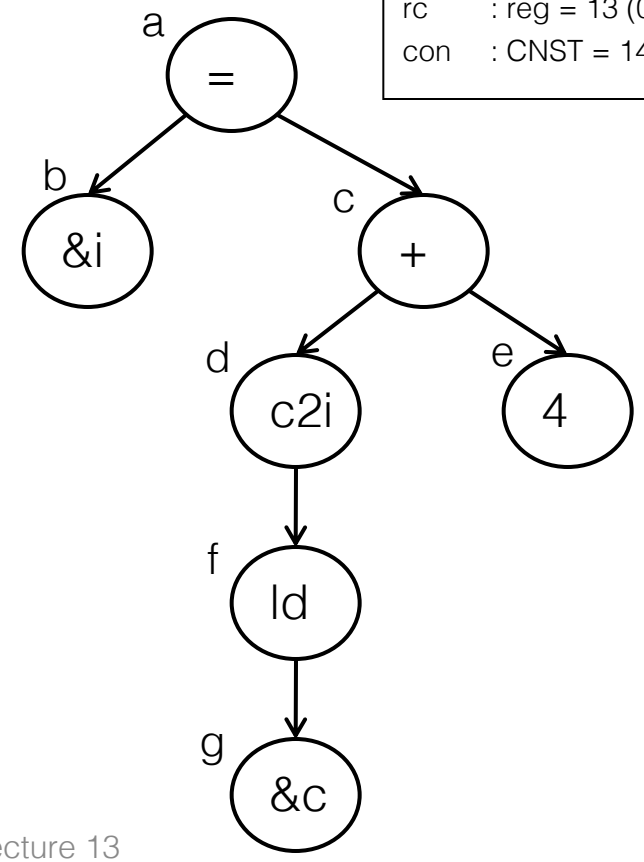
op	stmt	reg	addr	rc	con
a					
b					
c					
d					
e					
f					
g					



Top-Down (Reduction)

```
stmt : ASSIGN(addr, reg) = 1 (1)
stmt : reg = 2 (0)
reg  : ADD(reg, rc) = 3 (1)
reg  : ADD(rc, reg) = 4 (1)
reg  : LD(addr) = 5 (1)
reg  : C2I(LD(addr)) = 6 (1)
reg  : addr = 7 (1)
reg  : con = 8 (1)
addr : ADD(reg, con) = 9 (0)
addr : ADD(con, reg) = 10 (0)
addr : ADDRLP = 11 (0)
rc   : con = 12 (0)
rc   : reg = 13 (0)
con  : CNST = 14 (0)
```

op	stmt	reg	addr	rc	con
a	(1,3)				
b	(2,1)	(7,1)	(11,0)	(13,1)	
c	(2,2)	(3,2)	(9,1)	(13,2)	
d	(2,1)	(6,1)		(13,1)	
e	(2,1)	(8,1)		(12,0)	(14,0)
f	(2,1)	(5,1)		(13,1)	
g	(2,1)	(7,1)	(11,0)	(13,1)	



Instruction Selection

Issues (1)

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
  - Want to take advantage of multiple function units on chip
- Loads & Stores may or may not block
  - may be slots after load/store for other useful work

Issues (2)

- Branch costs vary
- Branches on some processors have delay slots
- Modern processors have heuristics to predict whether branches are taken and try to keep pipelines full

- GOAL: Scheduler should reorder instructions to hide latencies, take advantage of multiple function units and delay slots, and help the processor effectively pipeline execution

Latencies for a Simple Example Machine

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1
BRANCH	0 TO 8

Example: w = w*2*x*y*z;

Simple schedule

```
1  LOAD    r1 <- w
4  ADD     r1 <- r1,r1
5  LOAD       r2 <- x
8  MULT       r1 <- r1,r2
9  LOAD       r2 <- y
12 MULT    r1 <- r1,r2
13 LOAD       r2 <- z
16 MULT    r1 <- r1,r2
18 STORE    w <- r1
21 r1 free
```

2 registers, 20 cycles

Loads early

```
1  LOAD       r1 <- w
2  LOAD       r2 <- x
3  LOAD       r3 <- y
4  ADD     r1 <- r1,r1
5  MULT       r1 <- r1,r2
6  LOAD       r2 <- z
7  MULT       r1 <- r1,r3
9  MULT       r1 <- r1,r2
11 STORE   w <- r1
14 r1 is free
```

3 registers, 13 cycles

Instruction Scheduling

- Problem
  - Given a code fragment for some machine and latencies for each operation, reorder to minimize execution time
- Constraints
  - Produce correct code (required)
  - Minimize wasted cycles
  - Avoid spilling registers if possible
  - Do this efficiently

Precedence Graph

- Nodes *n* are operations
- Attributes of each node
  - type – kind of operation
  - delay – latency
- If node n_2 uses the result of node n_1, there is an edge e = (n_1,n_2) in the graph

Example Graph

- Code

```
a  LOAD    r1 <- w
b  ADD     r1 <- r1,r1
c  LOAD       r2 <- x
d  MULT       r1 <- r1,r2
e  LOAD       r2 <- y
f  MULT       r1 <- r1,r2
g  LOAD       r2 <- z
h  MULT       r1 <- r1,r2
i  STORE   w <- r1
```

Schedules (1)

- A correct schedule S maps each node n into a non-negative integer representing its cycle number, and
  - S (n) >= 0 for all nodes n (obvious)
  - If (n_1,n_2) is an edge, then S(n_1)+delay(n_1) <= S(n_2)
  - For each type t there are no more operations of type t in any cycle than the target machine can issue

Schedules (2)

- The *length* of a schedule S, denoted L(S) is

$$L(S) = \max_n (S(n) + delay(n))$$

- The goal is to find the shortest possible correct schedule
  - Other possible goals: minimize use of registers, power, space, …

Constraints

- Main points
  - All operands must be available
  - Multiple operations can be ready at any given point
  - Moving operations can lengthen register lifetimes
  - Moving uses near definitions can shorten register lifetimes
  - Operations can have multiple predecessors
- Collectively this makes scheduling NP-complete
- Local scheduling is the simpler case
  - Straight-line code
  - Consistent, predictable latencies

Algorithm Overview

- Build a precedence graph P
- Compute a priority function over the nodes in P (typical: longest latency-weighted path)
- Use list scheduling to construct a schedule, one cycle at a time
  - Use queue of operations that are ready
  - At each cycle
    - Chose a ready operation and schedule it
    - Update ready queue
- Rename registers to avoid false dependencies and conflicts

List Scheduling Algorithm

```
Cycle = 1;  Ready = leaves of P;  Active = empty;
while (Ready and/or Active are not empty)
    if (Ready is not empty)
     remove an op from Ready;
     S(op) = Cycle;
     Active = Active ∪ op;
    Cycle++;
    for each op in Active
     if (S(op) + delay(op) <= Cycle)
          remove op from Active;
          for each successor s of op in P
               if (s is ready – i.e., all operands available)
                    add s to Ready
```

Example

- Code

```
a  LOAD    r1 <- w
b  ADD     r1 <- r1,r1
c  LOAD       r2 <- x
d  MULT       r1 <- r1,r2
e  LOAD       r2 <- y
f  MULT       r1 <- r1,r2
g  LOAD       r2 <- z
h  MULT       r1 <- r1,r2
i  STORE   w <- r1
```

Forward vs Backwards

- Backward list scheduling
  - Work from the root to the leaves
  - Schedules instructions from end to beginning of the block
- In practice, compilers try both and pick the result that minimizes costs
  - Little extra expense since the precedence graph and other information can be reused
  - Different directions win in different cases

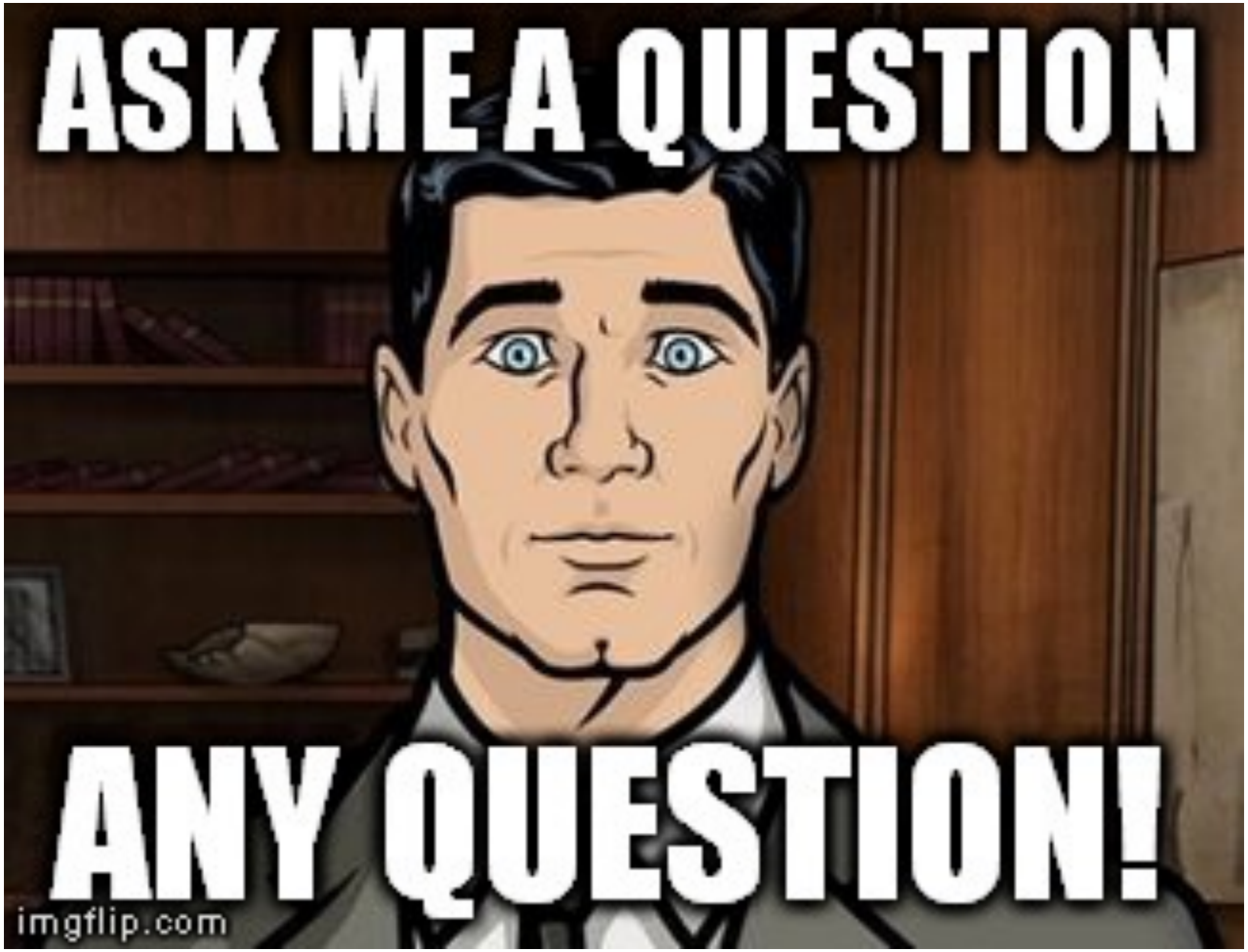
Beyond Basic Blocks

- List scheduling dominates, but moving beyond basic blocks can improve quality of the code. Some possibilities:
  - Schedule extended basic blocks
    - Watch for exit points – limits reordering or requires compensating
  - Trace scheduling
    - Use profiling information to select regions for scheduling using traces (paths) through code

And That's It

We covered:

- Compilers organization
- Front-end compiler architecture:
  - Scanners
  - Parsers
  - AST
- Intermediate representations
- Compiler optimization
  - Analysis
    - Dataflow
    - SSA
  - Transformations
- Code shape and code generation
- Back-end compiler architecture



[Meme credit: imgflip.com]