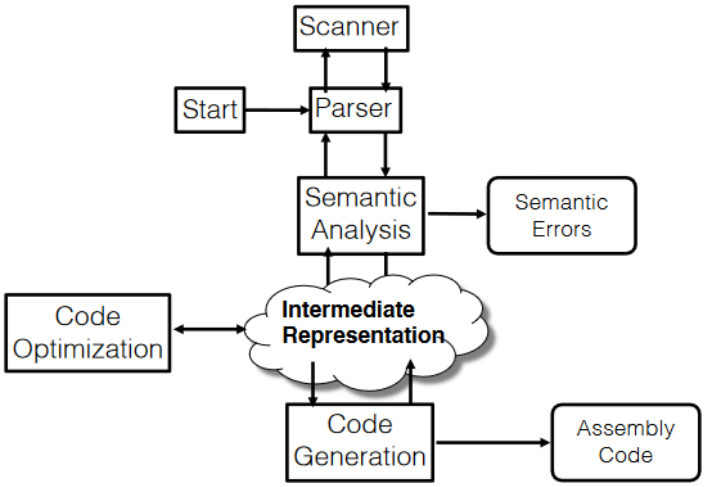
Lecture 1: Compiler Structure, Intermediate Representations

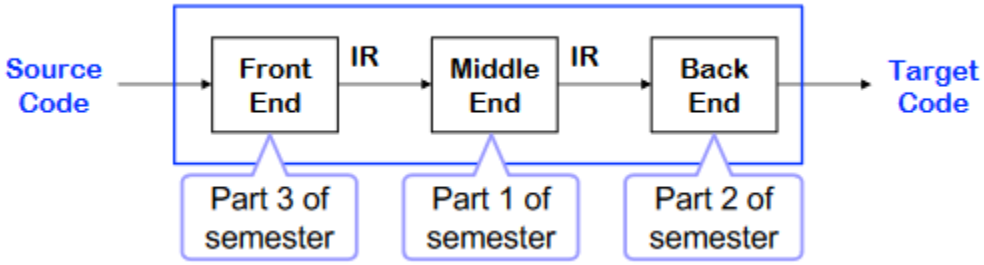
- A **compiler** is a program that translates an executable program in one language into an executable program in another language. The compiler should improve the program, in some way
    - *Important* - Responsible for many aspects of system performance
    - *Interesting* - Include many applications of theory to practice, and writing a compiler exposes practical algorithmic & engineering issues
    - *Everywhere* - Many practical applications have embedded languages and have input formats that look like languages



- An **interpreter** is a program that reads an executable program and produces the results of executing that program
    - Ex. Java is compiled into bytecodes (which is code for the Java VM), which is then interpreted

Role of Intermediate Representations in Code Generation

- There are three "parts" to a compiler…
    - Front End - Produces an intermediate representation (IR)
    - Middle End - Transforms the IR into an equivalent IR that runs more efficiently
    - Back End - Transforms the IR into native code



- IR encodes the compiler's knowledge of the program
- The middle end usually consists of several passes

Intermediate Representations

- Decisions in IR design affect speed & efficiency of compiler
- Important properties include…
    - Ease of generation
    - Ease of manipulation

    - Procedure size
    - Freedom of expression
    - Level of abstraction
- The importance of different properties varies between compilers, and therefore selecting an appropriate IR for a compiler is critical
- There are three major types of intermediate representations…
    - *Structural* - Graphically oriented, heavily used in source-to-source translators, tend to be large. Ex. Abstract Syntax Tree (AST)
    - *Linear* - Pseudo-code for an abstract machine, level of abstraction varies, simple and compact data structures, easier to rearrange. Ex. 3 address code, Stack machine code
    - *Hybrid* - Combination of graphs and linear code. Ex. Control-flow graph (CFG)

Stack Machine Code

- Originally used for stack-based computers, now Java

» Example:

x - 2 * y becomes

```
push x
push 2
push y
multiply
subtract
```

- Advantages include…
    - Compact form (implicit names take up no space, but explicit ones do)
    - Introduced names are implicit, not explicit
    - Simple to generate and execute code
- Useful where code is transmitted over slow communication links (e.g., to mobile phones)

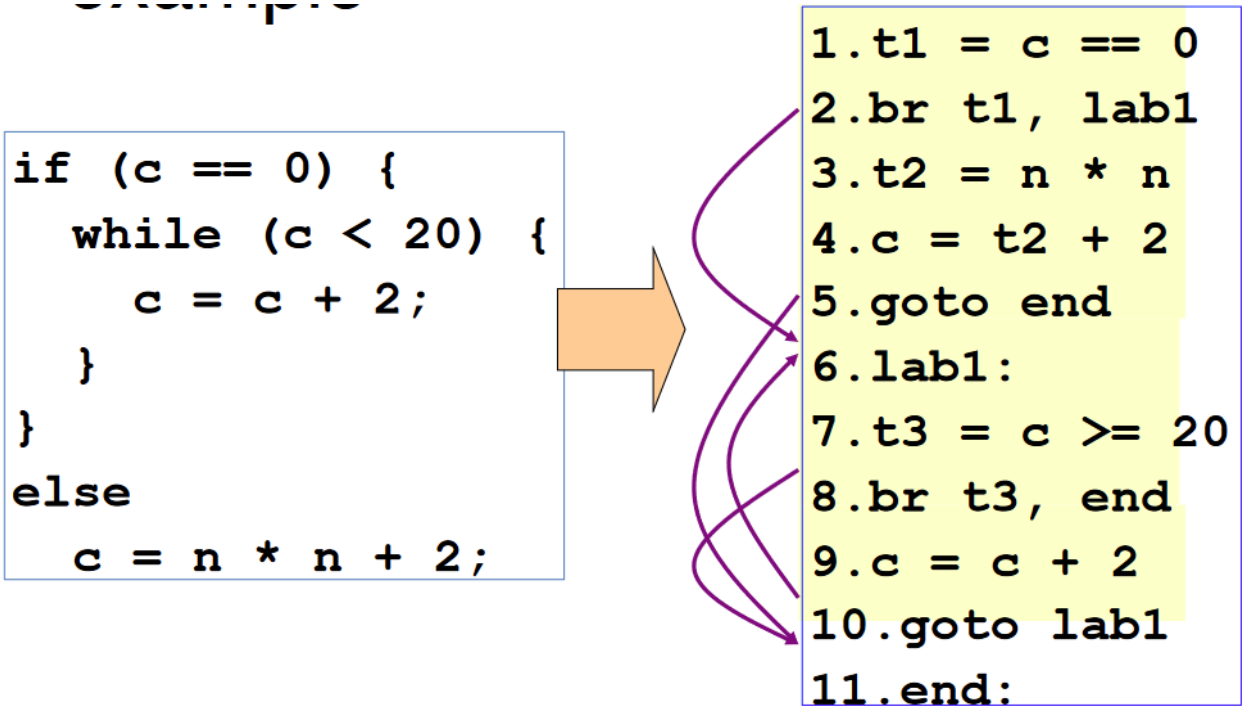
Three Address Code

- There exist several representations of three address code, however in general: three address code has statements in the form…

$$x \leftarrow y \underline{\text{op}}\ z$$

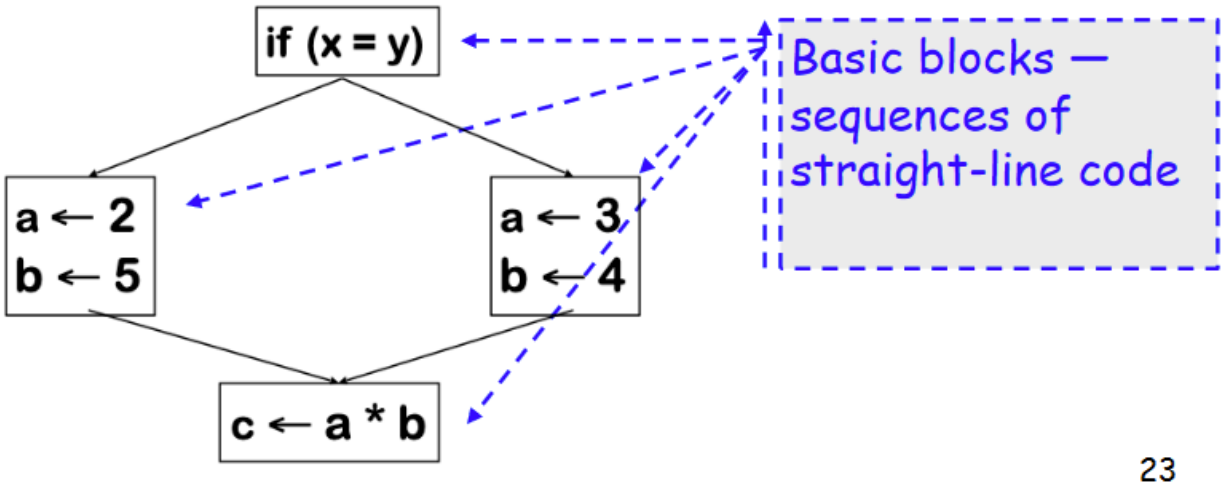
With 1 operator (op) and, at most, 3 names (x, y, z)

- Advantages include…
  - Resembles many real machines
  - Introduces a new set of temporary variables (virtual registers) visible only to the compiler, e.g., t1, r1, r2, …
- Example of generating a 3-address code…



Control-Flow Graph

- A **control-flow graph** models the transfer of control in the procedure
- Nodes in the graph are **basic blocks**, which are sequences of straight-line code
- Edges represent the flow of control
- Potential for exceptions can reduce basic block size in some languages, e.g. NullPointerException in Java



Tiger-IR - The IR of CS 4240

- We will be using three address code and follow the following instruction format…

  op, x, y, z

  Which is equivalent to "x ← y op z" (this is three address code)
- Example…

  2 * a + (b - 3)

  mult, t2, a, 2
  sub, t1, b, 3
  add, t3, t1, t2
- t1, t2, t3 are temporary variables generated by the compiler when generating the IR

Op	Example source	Example IR
goto	break;	goto, after_loop, ,

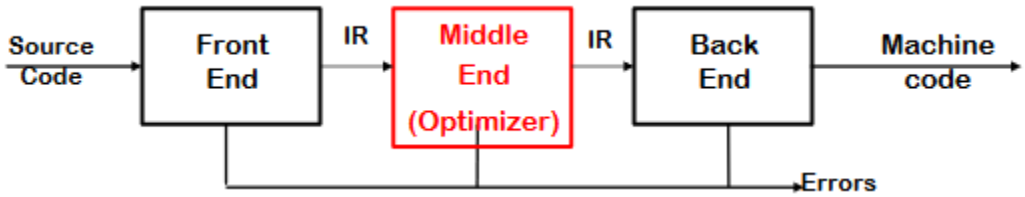
Op	Example source	Example IR
breq	if(a <> b) then	breq, a, b, after_if_part
brneq	if(a = b) then	brneq, a, b, after_if_part
brlt	if(a >= b) then	brlt, a, b, after_if_part
brgt	if(a <= b) then	brgt, a, b, after_if_part
brgeq	if(a < b) then	brgeq, a, b, after_if_part
brleq	if(a > b) then	brleq, a, b, after_if_part

- Functions start with a function signature IR statement denoting the return type, name of the function, and the function parameters

  `int foo(int first, float second)`
- Function signature is followed by a data segment consisting of type declarations for all variables (including arras and temporaries), e.g.
  - int-list: t1, t2, t3, a, b[100]
  - float-list: t4, t3, t6
- Function call instruction is an exception to the 3-address rule
  - **`call, func_name, param1, param2, …, paramn`**
- Function calls with return values have a similar structure
  - **`callr, x, func_name, param1, param2, …, paramn`**

Lecture 2 - Control Flow Graphs, Reaching Definitions

- Contemporary compiler contains 3 components
    - Front End: Reads in the source code
    - Middle End: Focuses on optimizing the IR
    - Back End: Generate a machine executable code



- The middle end is very important…
    - Analyzes IR And rewrites (or transforms) IR
    - Goal is to reduce running time of compiled code
    - Must preserve the "meaning" of the code

Dead Code Elimination

- Similar to mark-sweep garbage collection; mark useful operations, and everything not marked is useless
- Need an efficient way to find and mark useful operations
    - Start with critical operations
    - Work back up data flow edges to find their antecedents
- **Critical Operations**: I/O statements, linkage code (entry & exit blocks), return values, calls to other procedures, global variables that can be visible on program exit

Control Flow Graphs

- Assists us with creating a dead code elimination algorithm
- **Control flow** pertains to transfer of execution across program statements/instructions
    - Default execution is sequential
    - Can be altered by unconditional/conditional branch instructions & call instructions in intermediate code & machine code
- **Control flow graphs** are an abstract representation of control flow in programs
- A program is a sequence of instructions executed from top to bottom. This sequence of execution has special *control* instructions
    - Ex. Branches, loops, gotos, etc.
- This pattern of executions is the program's **control flow**
- These *control flow graphs* contain **basic blocks** (sequences of instructions), while edges indicate control flow from one basic block to another
- Reasons for CFGs
    - Abstract away the different control flow mechanisms in the language

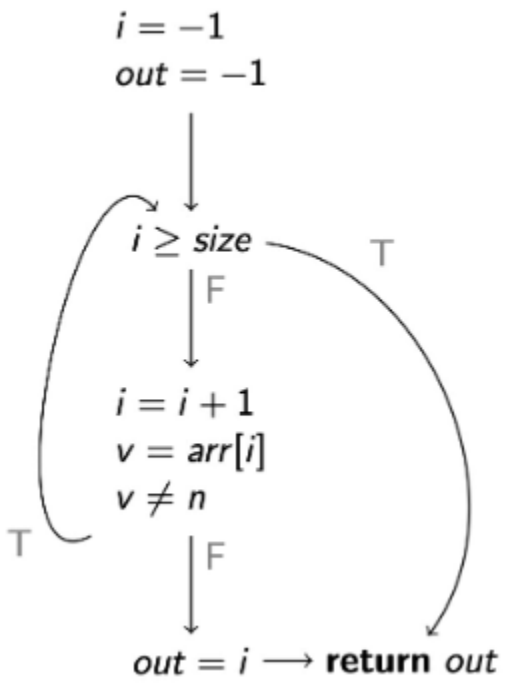
    - Graphical representation allows us to use general knowledge of graphs to manipulate programs
    - Useful *visual* representation of a program, which can aid in understanding
- **Basic blocks** are contiguous sequences of program instructions such that if the first instruction in the block is executed, so are the rest

Building a CFG

- The basic idea is to…
    - Determine where all the basic blocks are
    - Add a vertex for each basic block
    - Add the appropriate instructions to each basic block
    - Draw edges between the vertices
- In order to translate a sequence of instructions to a CFG, we need to know what the **leaders** are; every basic block has a leader, which is the first instruction. An instruction is a leader if it is…
    - The first instruction in the procedure
    - The target of a **goto** or **branch** instruction
    - The successor of a **branch** instruction
- The line numbers in the set to the right indicate which lines are leaders
- The below is an example of converting to a CFG

```
1:  SEARCH'(arr, size, n)
2:      i = −1
3:      out = −1
4:      branch (i ≥ size) 11
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10
8:      out = i
9:      goto 11
10:     goto 4
11:     return out
```

$\{2, 4, 5, 8, 10, 11\}$



Reaching Definitions

- There are two sites for a variable, a *def* site and a *use* site
- The declaration of a variable is **not** a definition, it is the instantiation of a variable to a value that is the definition of a variable.
    - Ex. *int* x is not a definition, however x = 2 is a definition
- **Program point** is the location before/after a specific IR instruction
- A definition *d* reaches program point *u* if there is a control-flow path from *d* to *u* that does **not** contain an intervening definition of the same variable as *d*
    - Ex.
    - x = 1 (*d*) ← definition site
    -
    - z = x (*u*) ← use site
    - ---
    - x = 1
    - x = 2 (*d*) ← definition site
    - z = x (*u*) ← use site
- Reaching definition is the last write of a variable, because it will always reach the using site

Applications of Reaching Definitions

- Used to improve the precision of dead code elimination
    - You only have to mark statements based on reaching definitions, instead of all definitions

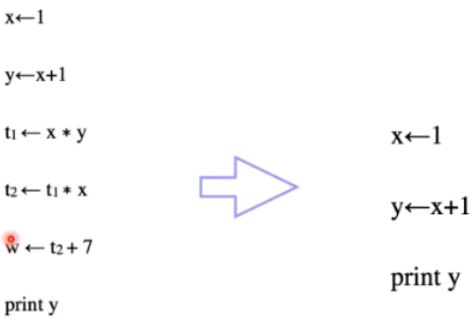
Lecture 3

General Roadmap for IR Optimizations

1. Identify an opportunity to modify the IR to improve performance
    a. Dead code elimination
2. Create an algorithm for performing the code transformation
    a. Worklist algorithm
3. Identify analyses needed by the algorithm in step 2
    a. Reaching definitions
4. Define data flow equations to formalize the desired solution to the analyses in step 3
5. Create an algorithm to solve the data flow equations in step 4
6. Use the analysis results from step 5 in the algorithm in step 2

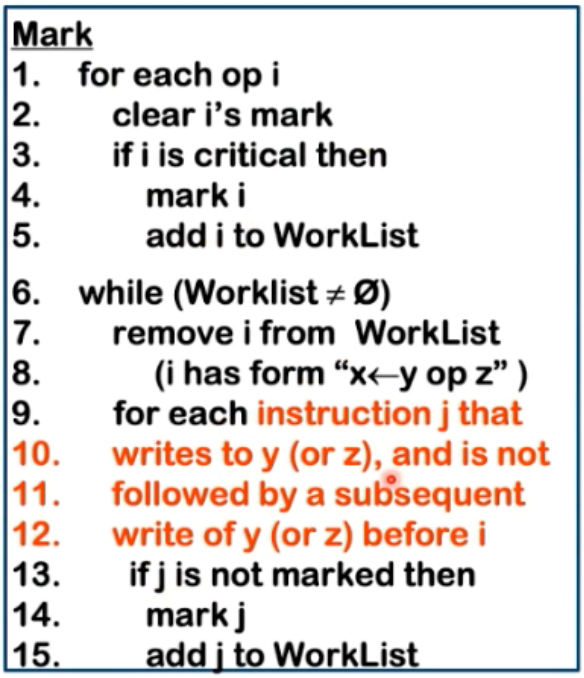
Step 1 Example - Dead Code Elimination (Lecture 1)

- Conceptually similar to *mark-sweep garbage collection*
    - Mark useful operations
    - Everything not marked is useless
- To do this…
    - Start with critical operations (i.e. I/O statements, branches)
    - Work back up data flow edes to find their antecedents



Step 2 & 3 Example - Dead-Code Elimination Algorithm (Lecture 2)

- Here, we are iterating through the operations and marking them depending on whether or not they are useful
- Using reaching definitions, we know which certain operations are critical; we only care about definitions that reach a particular use
    - If something is defined but not used, it's useless



Step 3: Analysis needed

Identification of instruction j that
writes to y (or z), and is not
followed by a subsequent
write of y (or z) before i

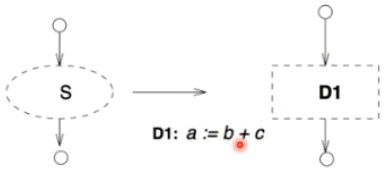
- The stuff in red corresponds to us looking for the reaching definitions
    - If *y* has a use, then there must be an instruction *j* that writes to it. If an instruction *j* writes to *y* and is not followed by a subsequent operation that writes to *y*, then that instruction is the definition in the reaching definition and the use of *y* is the use.

Step 4: Formalizing a Solution to the Reaching Definitions Problem

- Given a statement/instruction/basic block *S*, there exist two types of sets…
    - *Local Sets*, which are directly extracted from *S*. The sets obtained here are solely obtained by looking at the single basic block *S*, and therefore it cannot be changed in later iterations. These sets are not dependent on the interactions it has with other basic blocks.
        - GEN[*S*] = Set of definitions in *S* ("generated" by *S*)
        - KILL[*S*] = A set of definitions that *never* reach the end of *S* even if they reach the beginning
    - *Global Sets*, which are computed using a CFG.
        - IN[*S*] = set of definitions that reach the entry point of *S*
            - Formula: $IN[S] = \cup_{p \in predecessors} OUT[p]$
        - OUT[*S*] = set of definitions in *S* as well as definitions from IN[*S*] that go beyond *S* (are not "killed" by *S*)
            - Formula: $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$

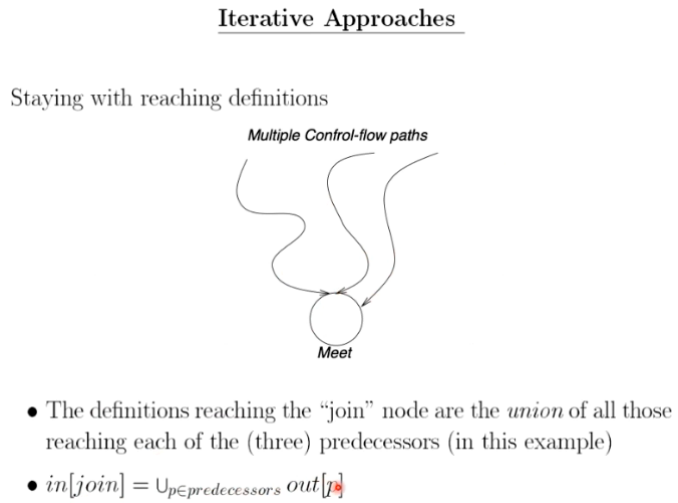
- *gen* A set of definitions that the reach the end of statement S whether they reach its beginning or not
- *kill* A set of definitions that *never* reach the end of S even if they reach the beginning
- *in* A set of definitions that reach the entry to statement S in the obvious way
- *out* A set of definitions that go past a statement S which include those that reach it and are not killed, and those in gen
  - Let's take a look at an example to see how we can create these sets

Single Statements



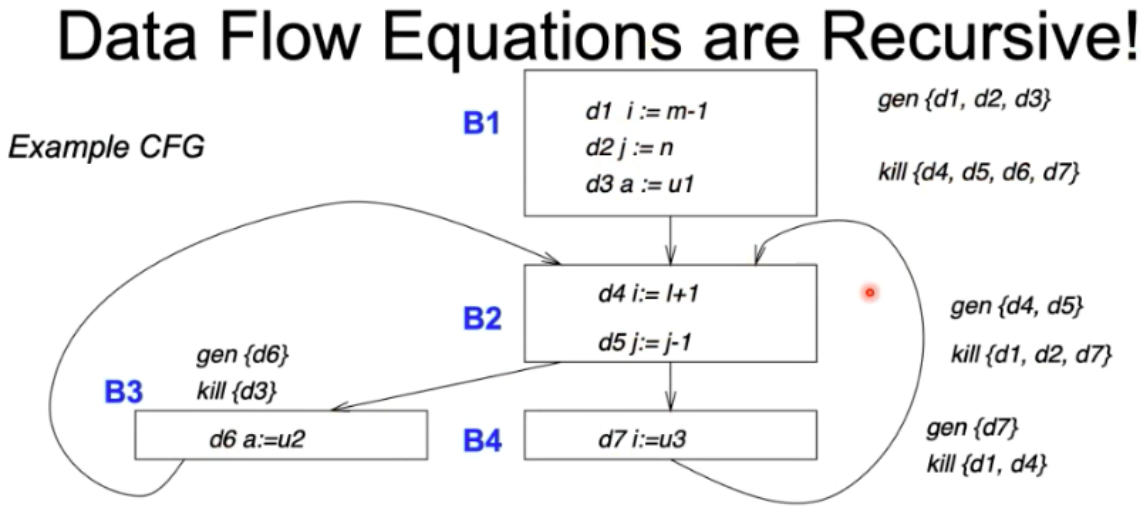
- $gen[S] = \{\mathbf{D1}\}$
- $kill[S] = \{D_a - \{\mathbf{D1}\}\}$ where D_a is the set of *all* definitions of a in the program
- $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$

- Here, we have a basic block *S* which generates a definition *D1*, and that's it. Since it generates this definition, the *gen[S]* consists of the instruction *D1*. However, it's not so simple. D1 is specifically an instruction that also writes to a variable *a* (we're assuming before this block *S* that there exists another definition of *a*.) Since this new definition overrides the previous definition, we need to kill all the previous definitions, which is exactly what *kill[S]* contains. Specifically, *kill[S]* contains every definition of *a* in the program minus D1, the most current definition to *a*.



- This one is quite easy to interpret. Since there are multiple branching paths which all meet at an eventual basic block, the *in* set is the *out* set of all the predecessors, which is just the definition of the *in* set as previously described

Step 5: Create an Algorithm to Solve the Data Flow Equations from Step 4



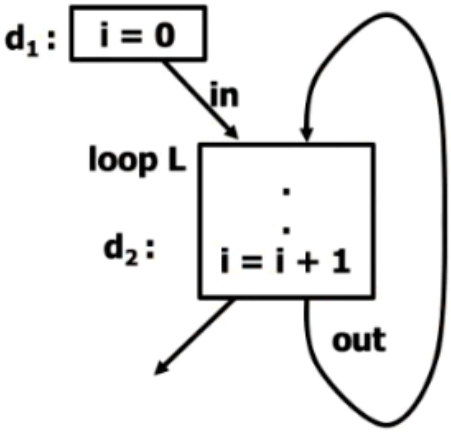
OUT[B1] = GEN[B1] U (IN[B1] - KILL[B1]) IN[B1] = { } // empty set	OUT[B3] = GEN[B3] U (IN[B3] - KILL[B3]) IN[B3] = OUT[B2]
OUT[B2] = GEN[B2] U (IN[B2] - KILL[B2]) IN[B2] = OUT[B1] U OUT[B3] U OUT[B4]	OUT[B4] = GEN[B4] U (IN[B4] - KILL[B4]) IN[B4] = OUT[B2]

- What's **incredibly** important to notice here is that even though the *gen* and *kill* sets seem local, the information in the *kill* sets pertains to the definitions from **all other basic blocks**. For example, in B2, notice how it's kill set contains definitions that are from B1

(d1 and d2) and definitions from B4 (d7). This goes back to the idea that on a local level of a basic block, the block doesn't know what's truly a predecessor or a successor. It's not until we start looking at a CFG do we understand what is and isn't.

Loops & Recursion, and the Fixed Point Iteration Method

- Consider the following…



Question:
What is the set of reaching definitions at the exit of the loop L?

$in[L] = \{d_1\} \cup out[L]$
$gen[L] = \{d_2\}$
$kill[L] = \{d_1\}$
$out[L] = gen[L] \cup \{in[L] - kill[L]\}$

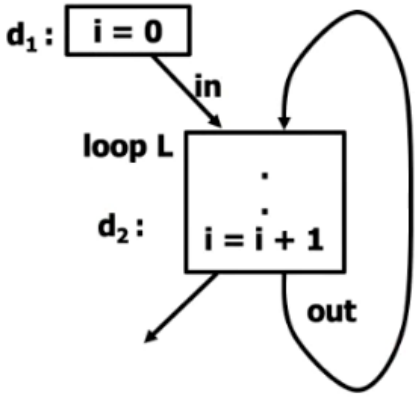
- For our loop here, we can easily find the *gen* and *kill* sets. However, we run into trouble when finding the *in* and *out* sets. Here, *in* depends on *out*, but *out* also depends on *in*. So how can we find them?
- Using the **fixed point iteration method**…

Initialization

$out[L] = \varnothing$

First iteration

$in[L] = \{d_1\} \cup out[L]$
$= \{d_1\}$
$out[L] = gen[L] \cup (in[L] - kill[L])$
$= \{d_2\} \cup (\{d_1\} - \{d_1\})$
$= \{d_2\}$



$in[L] = \{d_1\} \cup out[L]$
$gen[L] = \{d_2\}$
$kill[L] = \{d_1\}$
$out[L] = gen[L] \cup \{in[L] - kill[L]\}$

- Basically, take each iteration separately and start with *out* as an empty set. Eventually, you should reach a *fixed point*.

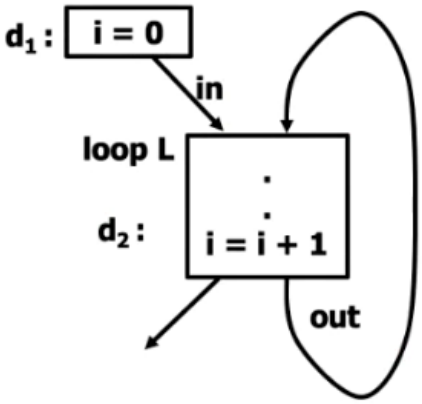
First iteration

$out[L] = \{d_2\}$

Second iteration

$in[L] = \{d_1\} \cup out[L]$
$= \{d_1, d_2\}$

$out[L] = gen\ [L] \cup (in\ [L] - kill\ [L])$
$= \{d_2\} \cup \{\{d_1, d_2\} - \{d_1\}\}$
$= \{d_2\} \cup \{d_2\}$
$= \{d_2\}$



We reached the fixed point!

$in\ [L] = \{d_1\} \cup out[L]$
$gen\ [L] = \{d_2\}$
$kill\ [L] = \{d_1\}$
$out\ [L] = gen\ [L] \cup \{in\ [L] - kill[L]\}$

- Notice how here, *out* from the 2nd iteration is the same as the *out* from the 1st iteration, so if we continued we'd keep getting the same thing, meaning we found the fixed point

Algorithm Summary: Overall Steps

Reaching Definitions:

- // Initialize *out* under the assumpt1¡ion that $in = \emptyset$ by setting $out[B] := gen[B]$ for all the blocks //
- $change :=$ **true**

  // This initiates the iteration and if there is a change after the iteration in *any* of the *out* sets, then it remains true//
- While *change* remains **true** compute
  - $in[B] = \cup_{p \in P}\, out[p]$ where P is the set of all predecessors of block B
- $tempout := out[B]$
- $out[B] := gen[B] \cup (in[B] - kill[B])$
- if $out[B] \neq temoput$ $change :=$ **true**

Lecture 4 - Redundancy Elimination, Value Numbering, Dominators

- Reaching definitions are incredibly useful for the optimization step of the compiler…
    - Identify dead/useless code
    - Identify uses that can be replaced by constants (constant propagation)
    - Identify uses that can be replaced by a def's rval (copy propagation)
    - Identify uses of initialized variables (uses that are not reached by any def)

Redundancy Elimination

- An expression x + y is **redundant** if, and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have <u>not</u> been re-defined
    - Ex

      a = x + y
      b = x + y
      b = t
      x + y is redundant in the 2nd statement b/c it's already defined above it; x + y = a

      a = x + y
      y = 3
      b = x + y
      t = b
      a is redundant

- If the compiler can prove that an expression is redundant…
    - It can preserve the results of earlier evaluations
    - It can replace the current evaluation with a reference
- Two pieces to the problem…
    - Proving x + y is redundant, or <u>available</u>
    - Rewriting the code to eliminate the redundant evaluation
- A technique for accomplishing both of these is **value numbering**

Value Numbering

- Here's how value numbering works: Assign an arbitrary, identifying "value number", *V(i)*, to each expression (rval) in IR instruction *i*
    - If two expressions have the same number, they will have the same value, so we can replace them.
    - If two expressions have different numbers, they may or may not always be the same
- This replacement means we can avoid recomputing the same expression, thus optimizing the program

- Here's the algorithm for **local value numbering**…

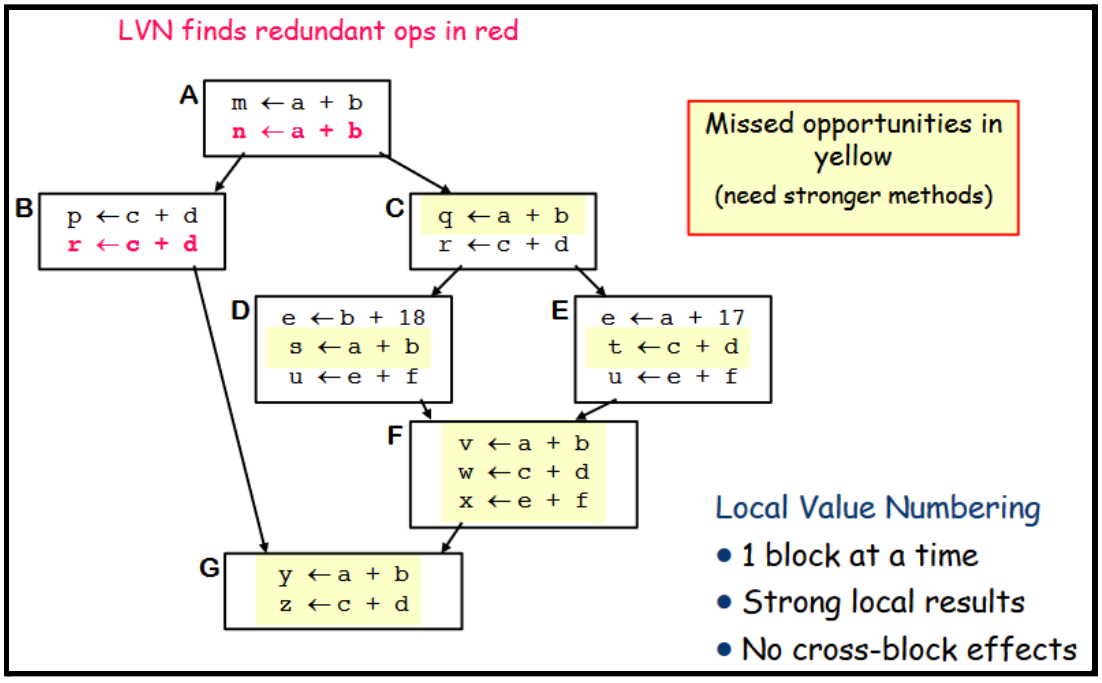
  For each operation **o = <operator, o_1, o_2>** in a basic block…

1. Obtain the value numbers for operands (o_1, o_2) from hash lookup
2. Hash the entire operation **<operator, o_1, o_2>** to get a value number for **o**
   a. If **o** already had a value number, replace it with a reference
   b. If **o_1 & o_2** are constant, evaluate it & replace with **load1**

- Note that this algorithm is for *local value numbering*, meaning it's limited to a single basic block and runs in linear time in practice
- Here's an example of the algorithm in practice…

Original Code	With VNs	Optimized
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$t \leftarrow x^1 + y^2$
* $b \leftarrow x + y$	* $b^3 \leftarrow x^1 + y^2$	$a \leftarrow t$
$a \leftarrow 17$	$a^4 \leftarrow 17$	* $b^3 \leftarrow t$
* $c \leftarrow x + y$	* $c^3 \leftarrow x^1 + y^2$	$a^4 \leftarrow 17$
NOTES:		* $c^3 \leftarrow t$

- The left-most code is normal code, the middle column code is the code with assigned value numbers, and the right-most code is the code that has been optimized using those value numbers
- However, because we are using *local value numbering*, you can imagine that this isn't very efficient if we run into cases where the same operation occurs in other basic blocks...

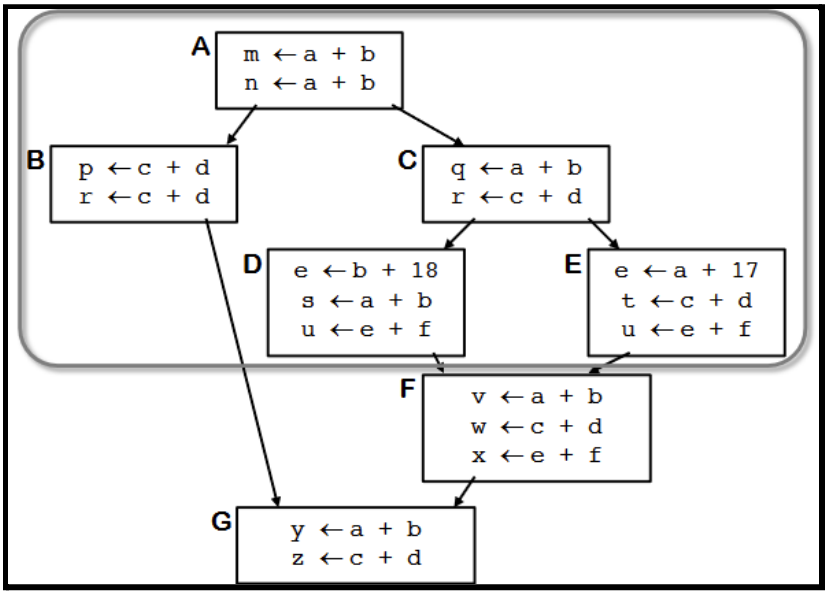


Scope of Optimization

- There are four scopes of optimization…
    - *Local Optimization* - Operates entirely within a single basic block. Properties of basic block lead to strong optimizations.
    - *Regional Optimization* - Operates on a *region* in the CFG that contains multiple blocks. Loops, trees, paths, extended basic blocks (EBBs)
    - *Whole Procedure Optimization* (in**tra**procedural) - Operates on the entire CFG for a procedure. Presence of cyclic paths forces analysis then transformation.
    - *Whole Program Optimization* (in**ter**procedural/multiple procedures) - Operate on some or all of the call graph. Must contend with call/return & parameter binding.
        - NOTE: The problem with doing whole program optimization is that it's a very *expensive* process and functions beyond the fundamentals of CS 4240. As such, we will not be concerning ourselves with it and therefore, for the sake of simplicity, we only need to know the first three scopes of optimization.

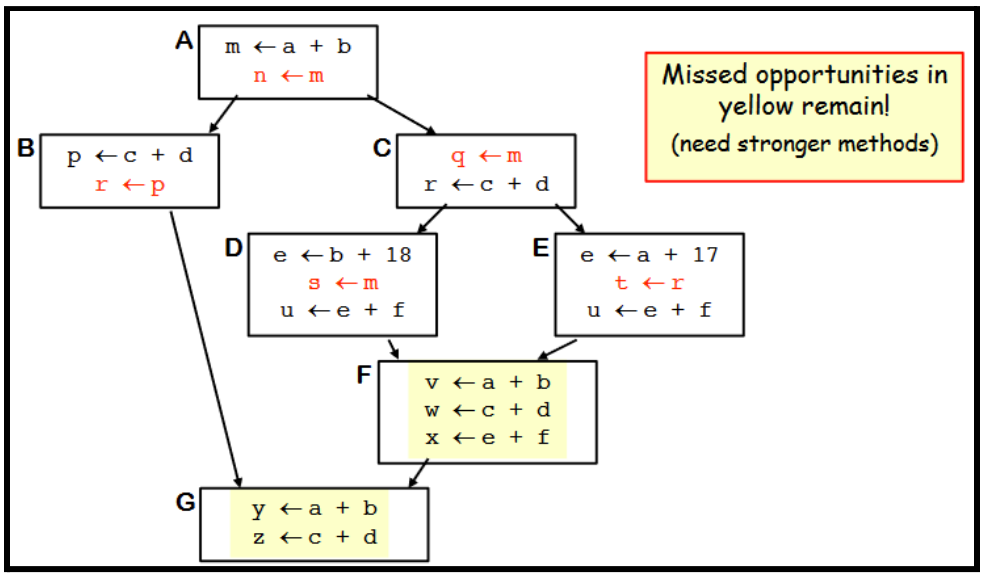
Regional Optimization & Extended Basic Blocks (EBBs)

- In *local optimization*, we only concerned ourselves with one basic block. In *superlocal optimization* or **regional optimization**, we concern ourselves with multiple basic blocks, something called an *extended basic block*.
- An **extended basic block** is a maximal set of blocks b_1, b_2, …, b_n where each b_1, *except* b_1, has only exactly one predecessor and that block is in the EBB.



- Here, we can see that we've made an extended basic block out of A, B, C, D, and E. Notice that F nor G are in this EBB because both of them have two predecessors. G has B and F & F has D and E

- However, the astute among you have probably noticed that we once again run into the same problem as with *local optimizations*; block F and G both utilize the same operations as previous blocks, however we are unable to utilize the optimization.



- It would be convenient if we could reach F and check to see if F can be optimized. In order to do this, we need to introduce a new idea: dominators.

Dominators

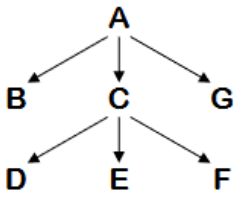
- As we discussed above, dominators are used to solve this problem with superlocal value numbering. For further convincing, let's look at an example from above…
    - Look at block C. Notice we have the assignment $r = c + d$. With this definition, so long as it's not redefined, we can use it in any superseding basic block. Take a look at block F. Block F uses the operation $c + d$, but because of the rules for EBBs, we are unable to extend the optimization to F. This is the exact problem that dominators attempt to solve.
- To begin our discussion with dominators, looking at blocks F and G, **which blocks always lie on the path to that block?**
    - For block F, you must go through block A and block C, however we can use either block D or E, so neither are *essential*. Thus, the set of required blocks is {A, C, F} (we include the block itself)
    - For block G, you must go through block A. However, you can either go left or right from A, therefore the only *essential* block is A, making the set {A, G}.
- We say those blocks are the *dominators*, or rather, A *dominates* G. Similarly, we'd say C *dominates* F.
- In general, we say x **dominates** if, and only if, every acyclic path from the entry of the CFG to the node for y includes x.
    - By the definition, a block will always dominate itself. Therefore, x *dominates* x.

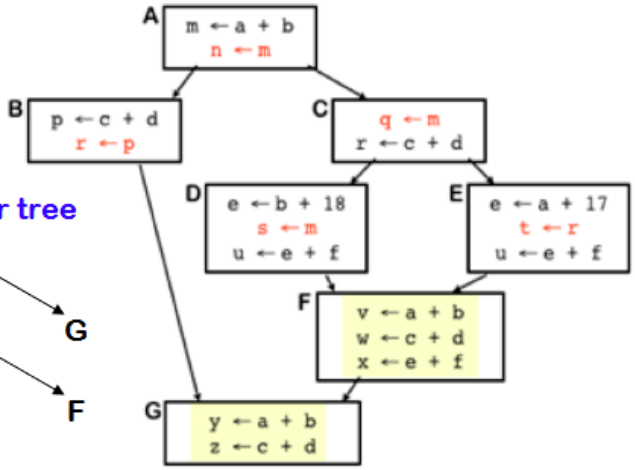
    - In layman's terms, when traversing through the CFG, if a block y is *required* to be traversed in order to reach some block x, we say that block y *dominates* block x.
- Each block has its own set Dom() containing the dominators of that block.
    - Ex. Dom(x) is the dominator set for block X
- Since each block x must have at least one dominator, we say that the closest dominator in dom(x) is the **immediate dominator**, written as IDom(x).
- *Immediate dominators* are useful for creating **dominator trees**, which are a graphical representation of the dominators in a CFG.

Dominator sets

Block	Dom	IDom
A	A	–
B	A,B	A
C	A,C	A
D	A,C,D	C
E	A,C,E	C
F	A,C,F	C
G	A,G	A

Dominator tree



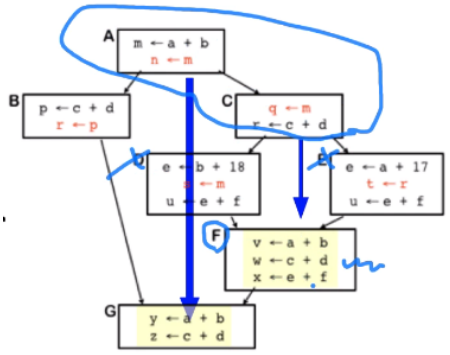


- To assist in generating the graph and chart, keep in mind the following…
    - CFG entry node n_0 dominates all CFG nodes
    - If d1 and d2 dominate n, then either…
        - d1 dominates d2, or
        - d2 dominates d1

Lecture 5 - Lazy Code Motion, Available Expression Analysis

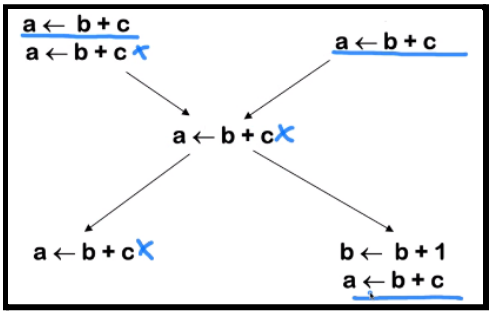
Redundancy Elimination

- Solved via **value numbering**
- Uses hashing in order to determine what arithmetic is redundant.
- Temporary variables can create dead code; will be solved in a later optimization
- *Local value numbering* (Block A, Block B) and *superlocal value numbering* can only get us so far; they don't cover every single basic block
- To overcome this, we use **dominators**. This way, we can reuse the hash tables from the blocks that dominate the blocks outside the EBB
    - Ex. We use the hash table from A to determine redundant arithmetic in G because A is a dominator of G.



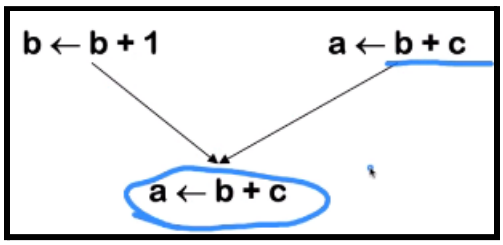
Dominator Value Numbering

- Works very well for linear programs, but how do we work with loops?
- An expression is **redundant** at a program point *p* if, on every path *p*
    - It is evaluated before reaching *p*, and
    - None of its constituent values is redefined before *p*



The last expression on the bottom right is **not** redundant because a constituent value is redefined (*b* is redefined before the arithmetic)

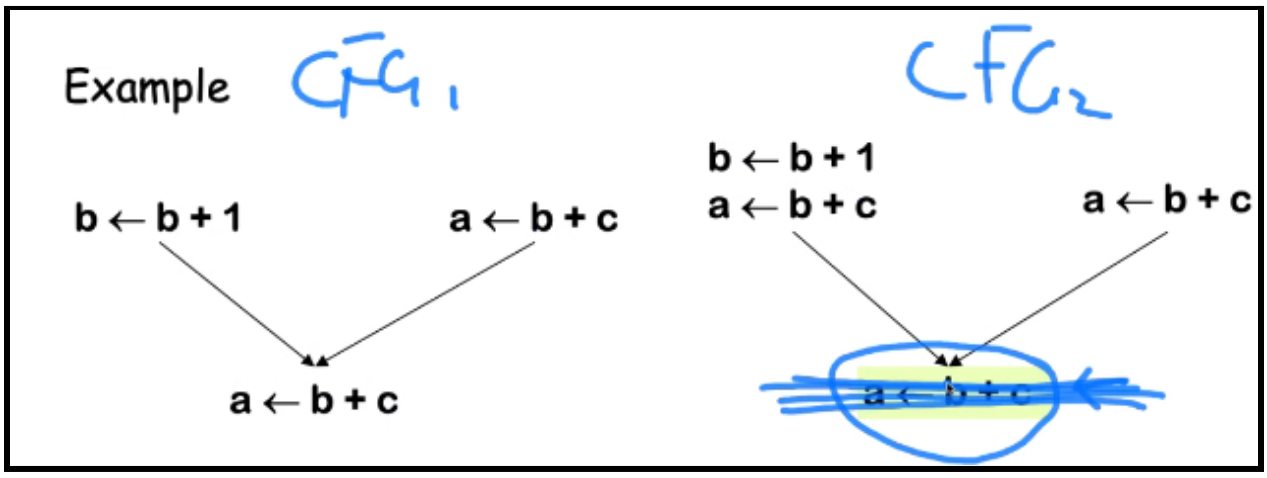
- An expression is **partially redundant** at *p* if it is redundant along some, but not all, paths reaching p



The circled expression is partially redundant b/c its only redundant if we do the right block first

- In order to handle a partially redundant expression…

- Insert the a copy of the operation on the side where it's not redundant
- Delete the original



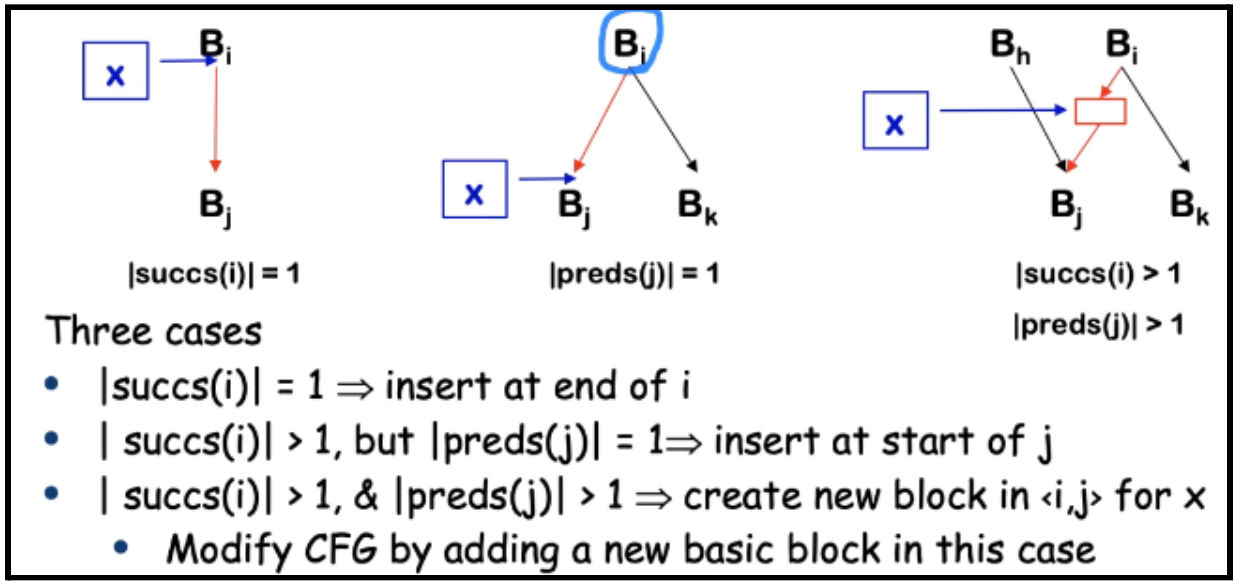
- Partial redundancies demonstrate an important concept: *code motion*

Lazy Code Motion

- Lazy code motion is an extension of the previously discussed topic of partial redundancy elimination
- For lazy code motion, we need to compute the sets of INSERT and DELETE via data flow analysis

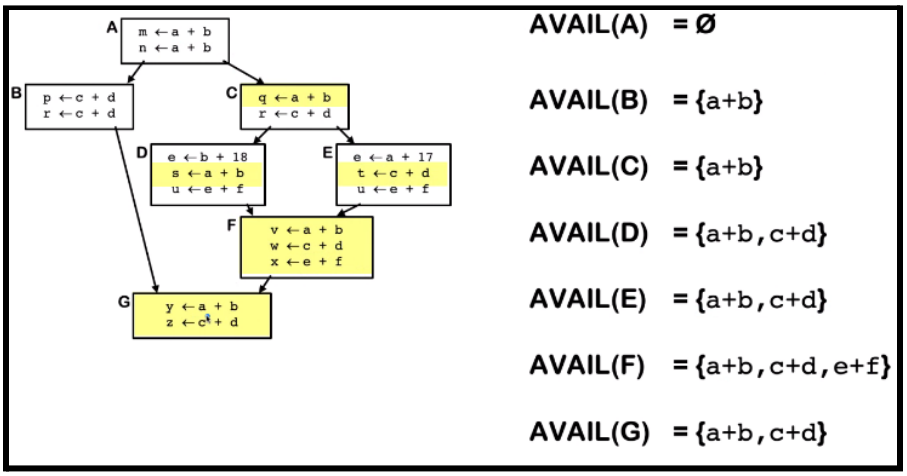
- Linear pass to rewrite code using INSERT & DELETE sets
  - x ∈ INSERT(i,j) ⇒ insert x at start of j, end of i, or new block
  - x ∈ DELETE(k) ⇒ delete first evaluation of x in k

- Insert Set



Available Expressions

- An expression x + y is **available** if, and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have <u>not</u> been redefined



For each block b, let

» Avail(b) be the set of expressions available on entry to b

» DEExpr(b) be the set of expressions computed in b and available on exit (Downward Exposed Expressions)

» ExprKill(b) be these set of expressions that are killed in b

» An expression is killed one of its inputs is assigned a value

Now, Avail(b) can be defined as:

$Avail(b) = \cap_{x \in pred(b)} (DEExpr(x) \cup (Avail(x) - ExprKill(x)))$

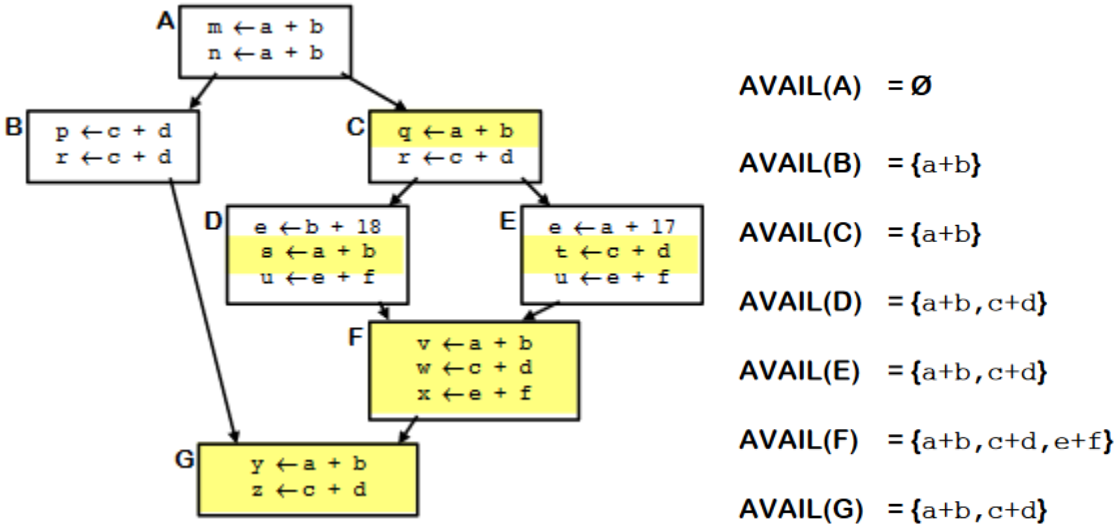
pred(b) is the set of b's predecessors in the control-flow graph

- This system of simultaneous equations forms a data-flow problem, and can be solved as past data-flow problems that we've seen (reaching definitions, dominators)

Lecture 7: Available Expressions Analysis & Constant Propagation

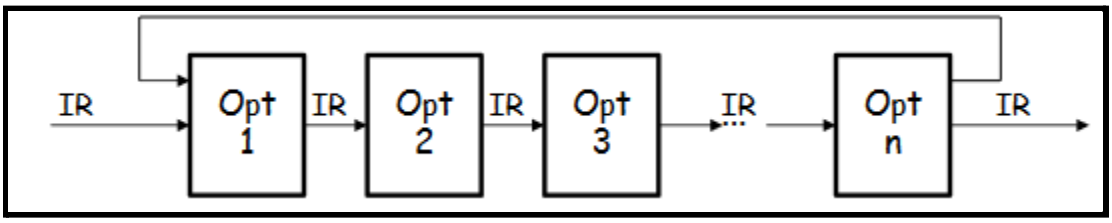
Review

- An expression x+y is **available** if and only if along every path from the procedure's entry, it has been evaluated, and it's constituent subexpressions (x & y) have <u>not</u> been redefined



A Review of the Middle of Compilers

- The middle-end performs multiplied optimization passes on an IR; <u>not just one</u>
  - The same optimization can be performed multiple times
- Combining optimizations to guarantee an optimal fixpoint can be challenging



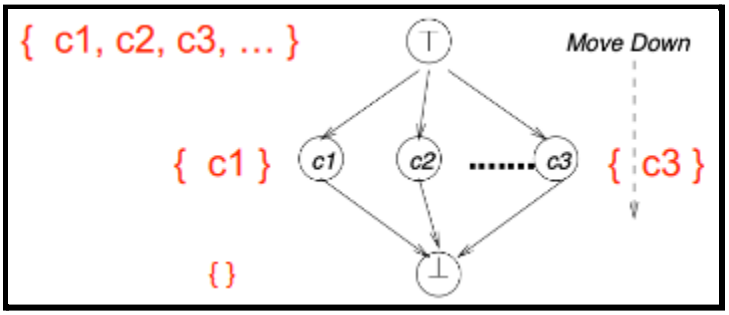
- The order of the optimizations do matter; some optimization strategies that we've learned are not as effective unless we have certain information (reaching definitions), which can not be obtained unless we pass the IR through an optimization
- We've learned of many optimizations and analysis thus far, here's a review…
  - *Reaching Definition Analysis*
    - OUT[S] and IN[S]
  - *Computation of Dominator Sets*
    - Dominator sets & dominator trees
    - The basic blocks that *must* be traveled through in order to reach a certain BB
  - *Available Expressions Analysis*
    - Avail(b) formula
  - *Constant Propagation* ← new one

Constant Propagation

- The *goal* of this analysis is to produce an algorithm that will propagate all constants in a procedure, replacing constant expressions with the result of evaluating the expression at compile time
- Strategy…
    - Construct **def-use chains** (reaching definitions) to map from definitions to uses within a procedure
    - Propagate constants forward from points of constant definitions along def-use chains
    - Evaluate new constant expressions whenever they are identified
    - Stop when no more constants are available
- Challenges…
    - Constructing **def-use chains** (we know this from def. of reaching definitions)
    - Identifying constant expressions in the presence of multiple reaching definitions
- In order to use these *def-use* chains, we need to learn of a new type of structure: a lattice structure

The Lattice Structure

- "A set equipped with a binary operation"
- A lattice value denotes a *set* of possible constant values. We are interested in the case when the *set* is a singleton

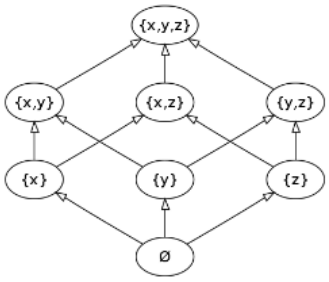


    - The perpendicular symbol (⊥) indicates that a constant value *cannot* be guaranteed; indicates that you cannot go any farther down (not a constant)
    - Several (potentially unlimited count of) constant symbols c_i that denote the space of all possible constants and that dominate ⊥
    - The constants are dominated by a unique T (upside perpendicular symbol) that represents the fact that corresponding variable/expression to which it is assigned *may* potentially be reducible to a constant (safe, but useless; all constants)
- To begin our lattice structure, begin with all nodes being assigned T
- Move down the lattice towards ⊥ and see whether the analysis stabilizes at a constant *c* in between or whether it reaches ⊥
- The rules for combination are as follows where *Anysymbol* denotes T, (perpendicular), or one of the constants C

1. $Anysymbol \sqcap \top = Anysymbol$
2. $Anysymbol \sqcap \bot = \bot$
3. $\mathcal{C}_i \sqcap \mathcal{C}_i = \mathcal{C}_i$
4. $\mathcal{C}_i \sqcap \mathcal{C}_j = \bot, i \neq j$

Demystifying Lattice

- Partial order: subset
- Meet: set intersection
- Join: set union



The meet operator, $\sqcap$ or $\wedge$ corresponds to the set intersection operation on sets denoted by the lattice values. It is performed at any point with two or more reaching definitions for the same variable.

Algorithm for Constant Propagation Analysis

```
// Initialization
Compute Def-Use "chains" (links from
each def to all uses that it reaches)
Worklist ← Ø
For i ← 1 to number of operations
   if in1 of operation i  is a constant ci
      then Value(in1,i ) ← ci
      else Value(in1,i ) ← Top
   if in2 of operation i  is a constant cj
      then Value(in2,i ) ← cj
      else Value(in2,i ) ← Top
   if (Value(in1,i ) is a constant &
       Value(in2,i ) is a constant)
      then Value(out,i ) ← evaluate op i
           Worklist ← Worklist ∪ {i }
```

```
// Iteration using meet operator, ∏ or ∧, //
in a semi lattice
while ( Worklist ≠ Ø)
   remove a definition i from WorkList
   for each j ∈ USES(out,i )
      set x so that out of i is inx of j
      Value(inx,j ) ← Value(out,i )
      for each k ∈ DEFS(inx,j ), k ≠ i
         Value(inx,j ) ← Value(inx,j )
                       ∧ Value(out,k )
      if (Value(in1,j ) is a constant &
          Value(in2,j ) is a constant)
         then Value(out,j ) ← evaluate op j
              Worklist ← Worklist ∪ {j }
      else if (Value(in1,j ) is ⊥ or
               Value(in2,j ) is ⊥)
         then Value(out,j ) ← ⊥
      else   Value(out,j ) ← ⊤
```

Lecture 8: Copy Propagation, Constant Propagation Revisited

Copy Propagation

- Given an assignment $x = y$, replace later uses of x with uses of y, provided there are no intervening assignments to x or y
- Performed as a clean-up pass after each optimization that introduces one or more copy statements
    - e.g. redundancy elimination after value number or available expression analysis
- Results in smaller, possibly more efficient, code depending on target machine & interaction w/ other optimizations

Local Copy Propagation

- Performed within basic blocks
- Algorithm outline…
    - Traverse BB from top to bottom
    - Maintain hashtable of copies encountered so far
    - Modify applicable instructions as you go

step	instruction	updated instruction	table contents
1	b = a	b = a	{(b,a)}
2	c = b + 1	c = a + 1	{(b,a)}
3	d = b	d = a	{(b,a), (d,a)}
		a = ….	
4	b = d + c	b = a + c	{(d,a)}
5	b = d	b = a	{(d,a), (b,a)}

    - Notice on line 4 how *b* gets redefined. Since it gets redefined, we can no longer simply replace occurrences of *b* with *a*, and so we have to remove it from the table.
    - On line 5, *b* gets redefined to *d*, which we already know is *a*, so we re-add *b* to the hashtable.
    - If we were to redefine *a*, we would need to clear the table of all key-value pairs where the value is *a*
        - So in summary, any time a *key* or *value* in the hashtable is redefined, we need to remove it from the table

```
for instr = i1 to in
    if instr is of the form 'res = opd1 op opd2'
        opd1 = REPLACE(opd1, copytable)
        opd2 = REPLACE(opd2, copytable)
    else if instr is of the form 'res = var'
        var = REPLACE(var, copytable)
    if instr has a lhs res,
        REMOVE from copytable all pairs involving res.
    if instr is of the form 'res = var' /* i.e. a copy */
        insert {(res, var)} in the copytable
endfor
```

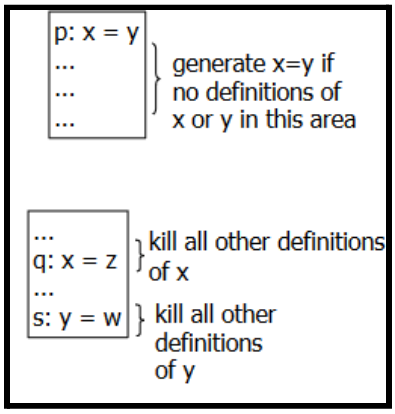
copytable is table containing copy pairs
e.g. if there's an assignment x := a, then
copytable should contain
(x,a)

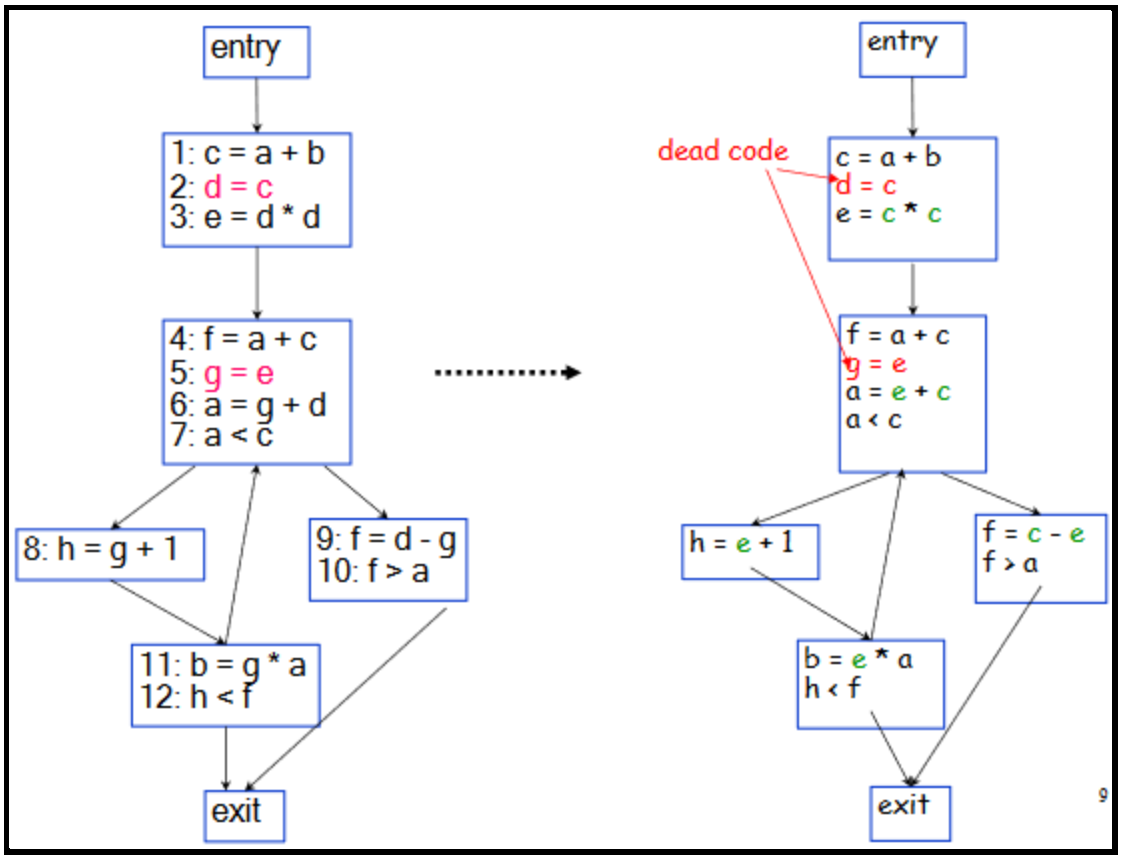
```
REPLACE(opd, copytable)
    if you find (opd, x) in copytable /* use hashing for faster access */
        return x
    else return opd
```

6

Global Copy Propagation

- In a global setting, we have to take the entire CFG into consideration
- Given a copy statement $x = y$ and use $w = x$, we can replace $w = x$ with $w = y$ if the following holds…
    - $x = y$ must be the **only** definition of x reaching $w = x$
        - Determined via output of reaching definitions
    - There may be no definitions of y on any path from $x = y$ to $w = x$
        - Use iterative data flow analysis to solve this
        - Can be combined w/ reaching definitions analysis
- Much like everything else, we must use data flow analysis to determine which instructions are candidates for global copy propagation…
    - **gen**[B_i] = {(**x**, **y**, i, p) | p is the position of $x = y$ in block B_i and neither x nor y is assigned a value after p}
    - **kill**[B_i] = {(**x**, **y**, j, p) | x = y, located at position p in block B_j =/= B_i, is killed due to a definition of **x** or **y** in B_i}
    - **in**[B] = ∩ **out**[P] over all predecessors P of B





Creating the Constant Propagation Algorithm

- GOAL: Produce an algorithm that will propagate all constants in a procedure, replacing constant expressions with the result of evaluating the expression at compile time
- We can approach this problem with a few different strategies based on what we've learned so far…
    - *Value Numbering* - Evaluate constant expressions in hashmap, as part of value numbering process
    - *Single Definition* - if a use of variable x is reached by a single definition which has the form, "x = <constant>", then the use can be directly replaced by the constant
    - *Copy Propagation* - Applied to copy statements of the form, "x = <constant>"
    - *Reaching Definitions* - model the rval of each def as a set of values, and take the union of all sets reaching a use to identify constants

```
// Initialization
Compute Def-Use chains
Worklist ← Ø
For i ← 1 to number of operations
   if in1 of operation i  is a constant ci
      then Value(in1,i ) ← ci
      else Value(in1,i ) ← Top
   if in2 of operation i  is a constant cj
      then Value(in2,i ) ← cj
      else Value(in2,i ) ← Top
   if (Value(in1,i ) is a constant &
       Value(in2,i ) is a constant)
      then Value(out,i ) ← evaluate op i
           Worklist ← Worklist ∪ {i }
```

```
// Iteration using meet operator, ∧, in a
// semi lattice
while ( Worklist ≠ Ø)
   remove a definition i from WorkList
   for each j ∈ USES(out,i )
      set x so that out of i is inx of j
      Value(inx,j ) ← Value(out,i )
      for each k ∈ DEFS(inx,j ), k ≠ i
         Value(inx,j ) ← Value(inx,j )
                         ∧ Value(out,,k )
      if (Value(in1,j ) is a constant &
          Value(in2,j ) is a constant)
         then Value(out,j ) ← evaluate op j
              Worklist ← Worklist ∪ {j }
         else if (Value(in1,j ) is ⊥ or
                  Value(in2,j ) is ⊥)
         then Value(out,j ) ← ⊥
```

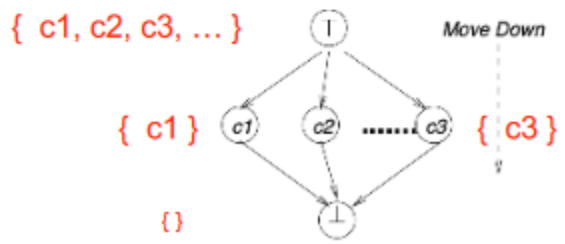
11

The Lattice Structure Revisited
(im just going to post the screenshots cause yeah)

The Lattice Structure

A lattice value denotes a set of possible constant values. We are interested in the case when the set is a singleton.

- We have a unique symbol ⊥ representing the fact that a contant value *cannot* be guaranteed
- Several (potentially unbounded number of) constant symbols C_i that denote the space of all possible constants and that dominate ⊥
- These constants are dominated by a unique ⊤ symbol that represents that fact that the corresponding variable/expression to which it is assigned *may* potentially be reducible to a constant



The Intuition

- We start out with all the nodes being assigned $\top$
- The idea is to move down the lattice towards $\bot$ and see whether the analysis stabilizes at a constant $\mathcal{C}_i$ in between or whether it reaches $\bot$
- The rules for combination are as follows where *Anysymbol* denotes $\top$, $\bot$ or one of the constants $\mathcal{C}_i$
  1. $Anysymbol \sqcap \top = Anysymbol$
  2. $Anysymbol \sqcap \bot = \bot$
  3. $\mathcal{C}_i \sqcap \mathcal{C}_i = \mathcal{C}_i$
  4. $\mathcal{C}_i \sqcap \mathcal{C}_j = \bot, i \neq j$

The meet operator, $\sqcap$ or $\wedge$ corresponds to the set intersection operation on sets denoted by the lattice values. It is performed at any point with two or more reaching definitions for the same variable

Background: What is a Semilattice?

- Some domain of values . . .
- Example: {a, b, c}
- . . . a meet operator: $\wedge$ . . .
- . . . and a top element: $\top$
- The top element is defined as $\top \wedge a = a$, for all elements a in the domain.
- Optionally, there may be a bottom element: $\bot$, where $\bot \wedge a = \bot$, for all elements a in the domain.

Partial Ordering

- The meet operator causes all semilattices to be partially ordered
- If an ordered set is one where the operation a < b applies to any two elements, a partially ordered one is where the operation $a \leq b$ applies.

- ≤ does not necessarily mean less than or equal to; it can be any ordering
- $a \leq b$ can mean…
  - a is smaller than or equal to b
  - a is a refinement of or the same as b
  - a has the same or fewer elements than b

    …as long as the relationship is always the same for all elements

Meet the Meet Operator

The meet operator can be any binary operator that has the following attributes for all elements in the domain:

- $a \wedge a = a$
- $a \wedge b = b \wedge a$
- $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

Good candidates for the meet operator are union ∪ and intersection ∩

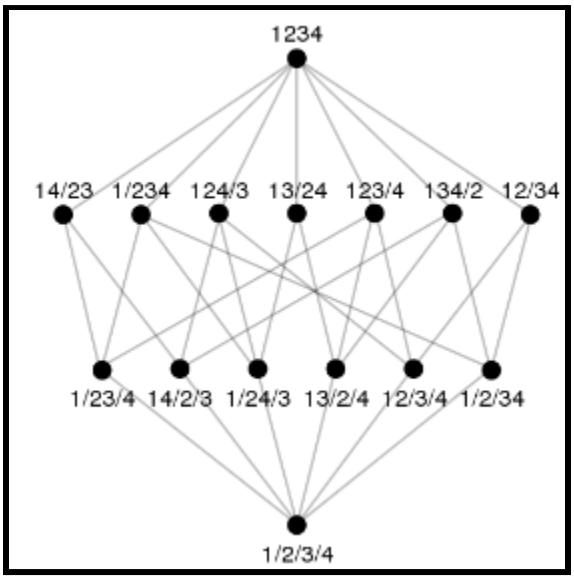
Remember $\top \wedge a = a$

If ⊤ is empty, ∧ is union

If ⊤ is all the elements in the domain, ∧ is intersection

Semilattice vs. Lattice

- Lattices have all the qualities of semilatticies
- They have two binary operators: a join operator (V) in addition to the meet operator (^)
- The join operator is complementary to meet: if meet means union, join means intersection, and vice versa



Detecting Unreachable Code

```
i = 1
…
if (i == 1)
    j = 1;
else
    j = 2;
```

- Note that control variable *i* is a constant
- Traditional data flow analysis would not catch the fact that…

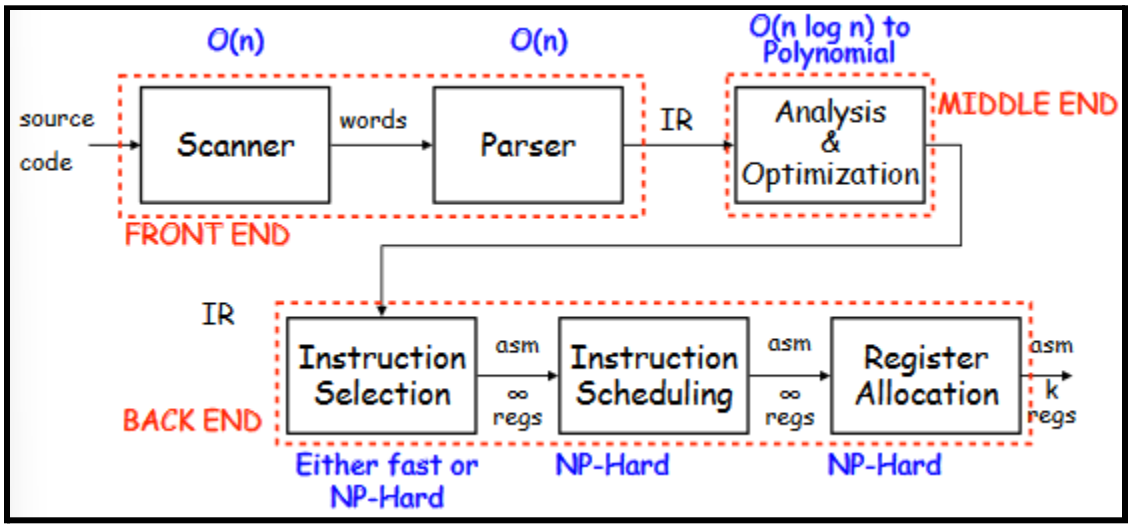
    - j is a constant and
    - $j = 2$ is *never executed*; this represents *unreachable code* which is a variety of dead-code

  because it does not factor in the outcome of control instructions such as conditional branches
    - With care, many data flow analyses can be combined with unreachable code elimination for increased precision

Summary of Optimizations

- Dead Code Elimination (with & without reaching definitions)
- Redundancy Elimination Driven by…
    - Local Value Numbering (LVN)
    - Superlocal Value Numbering (SVN)
    - Dominator Value Numbering Technique (DVNT)
    - Available Expression Analysis
    - Lazy Code Motion (Partial Redundancy Elimination)
- Constant Propagation
- Copy Propagation
- Unreachable Code Elimination

Lecture 9 - Instruction Selection



- So far, we've worked with the middle-end, which handles optimizing IR
- We consider this "end" to be "machine-independent," because we don't have to consider the specifications of the machine we are on
- The backend is considered "machine-dependent" because we have to consider the hardware; there are unique physical constraints (i.e. number of registers) that we have to account for when creating the backend
- Notice in the figure above that up until register allocation, we assume we have infinitely many registers, however in the final portion of the backend, we have to convert our infinitely many registers into *k* many registers. These are just some of the problems we have to handle when working with the backend
- Definitions…
    - **Instruction Selection**: Mapping IR into assembly code for a target processor (e.g. MIPS)
        - Assumes fixed storage mapping & coee shape
        - Combining operations, using address modes
    - **Register Allocation**: Decides which values will reside in registers
        - Reduces load, store, and copy statements in IR
        - Can create "false" dependencies when two variables are mapped to the same register
    - **Instruction Scheduling**: Reorders operations to hide latencies
        - Assumes a fixed program (set of operations)
        - Impacts the demand for registers

The Problem

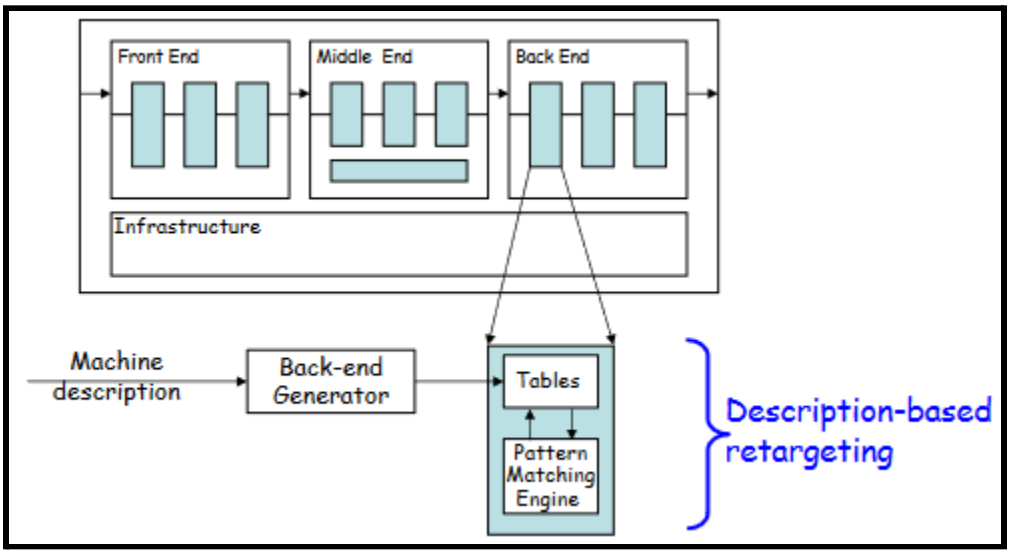
- Modern computers have many different ways to solve a problem
- Consider the operation of copying the value in one register to another…

addI r_i,0 ⇒ r_j	subI r_i,0 ⇒ r_j	lshiftI r_i,0 ⇒ r_j
multI r_i,1 ⇒ r_j	divI r_i,1 ⇒ r_j	rshiftI r_i,0 ⇒ r_j
orI r_i,0 ⇒ r_j	xorI r_i,0 ⇒ r_j	… and others …

- A human would ignore all these; but a computer must examine all of them and find the one with the lowest-cost

The Goal

- Our goal of the backend is to automate generation of instruction selectors

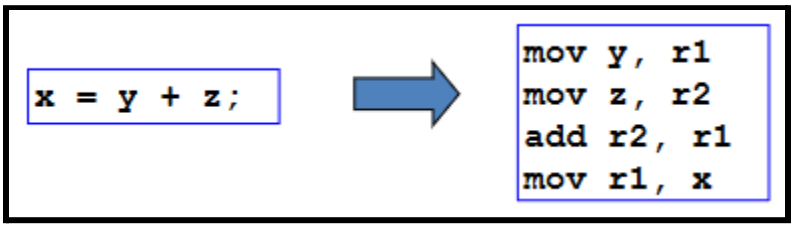


Instruction Selection

- The reason why we convert into IR first is so we can do machine-independent optimizations
- Once these optimizations are done, we need to convert them into ISA, which is machine *dependent*
- Having the IR be machine independent allows for different back ends and abstraction, which makes optimization easier
- Differences between IR and ISA…
  - *IR*: simple, uniform set of operations
  - *ISA*: many specialized instructions
- Example…
  Consider an example IR generated from the source code statement, a[i + 1] = b[j]

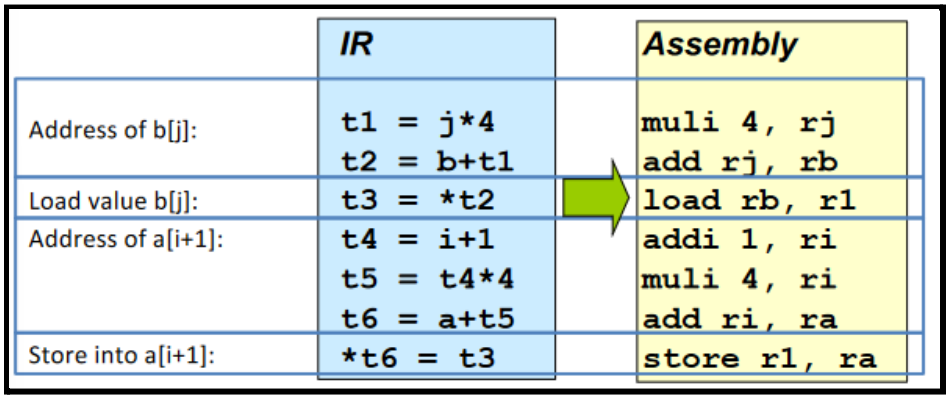
	IR
Address of b[j]:	t1 = j*4 t2 = b+t1
Load value b[j]:	t3 = *t2
Address of a[i+1]:	t4 = i+1 t5 = t4*4 t6 = a+t5
Store into a[i+1]:	*t6 = t3

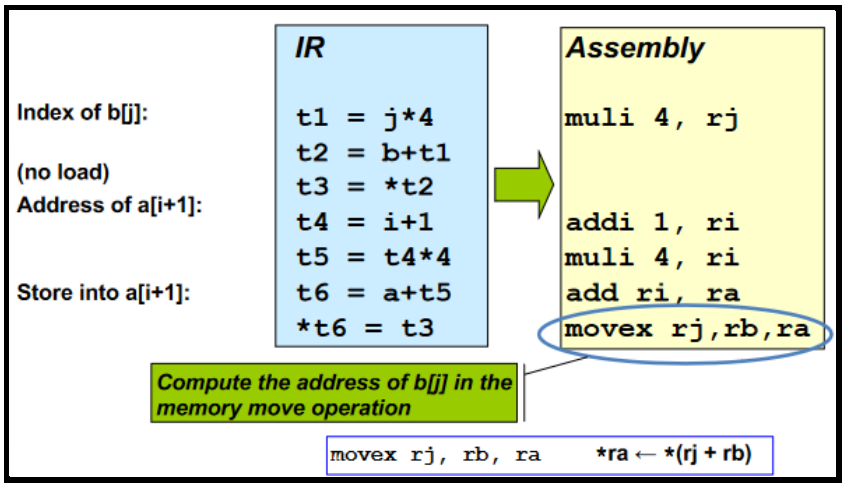
- Let's take a look at some of the different approaches we can take for *instruction selection*…
    - Map each IR operation to an instruction
        - Problem: Inefficient and very expensive



    - Convert IR to Assembly Code
        - Problem: Not every operation is necessary (i.e. multiplication is very expensive, and can be replaced with another, more efficient one like binary shifts) & some operations are more efficient than others

    Here is an unoptimized version, the next one uses two optimizations



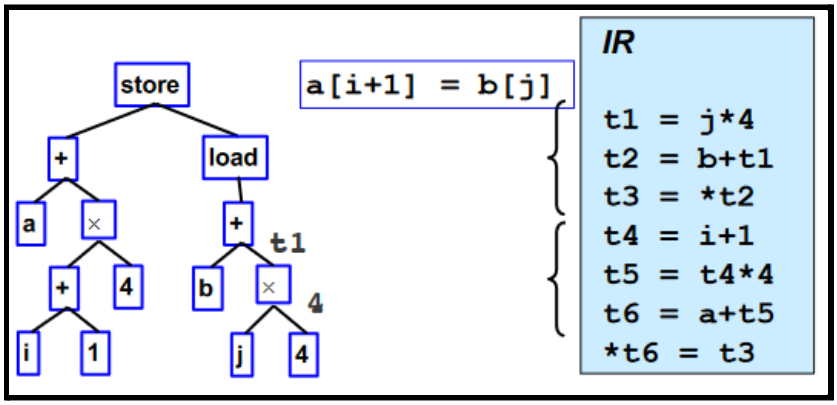


The Big Picture

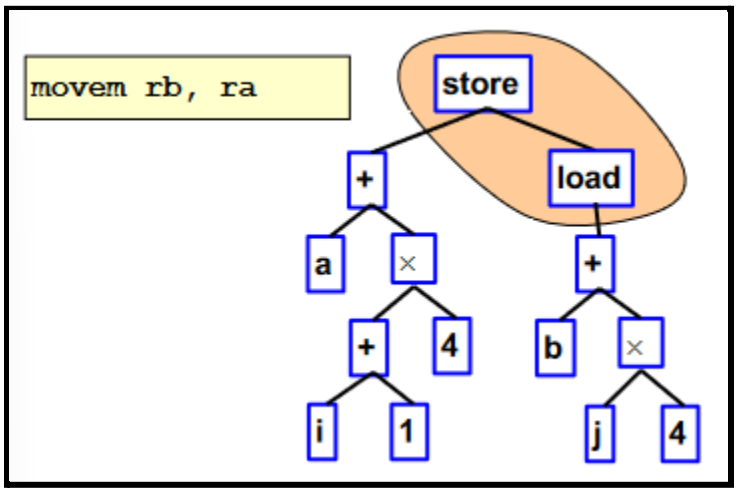
- *Instruction selection* is all about producing low-cost, efficient code
- Linear IR w/ three-address code suggests using some sort of string matching
    - Process takes strings as input, matcher as output
    - Each string maps to a target-machine instruction sequence
    - Use text matching or peephole matching
- Tree-oriented IR suggests pattern matching on **trees**
    - Process takes tree-patterns as input, matcher as output
    - Each pattern maps to a target-machine instruction sequence
    - Use dynamic programming or bottom-up rewrite systems
    - It is possible to extract trees from Linear IR as well

Minimizing Cost in Instruction Selection

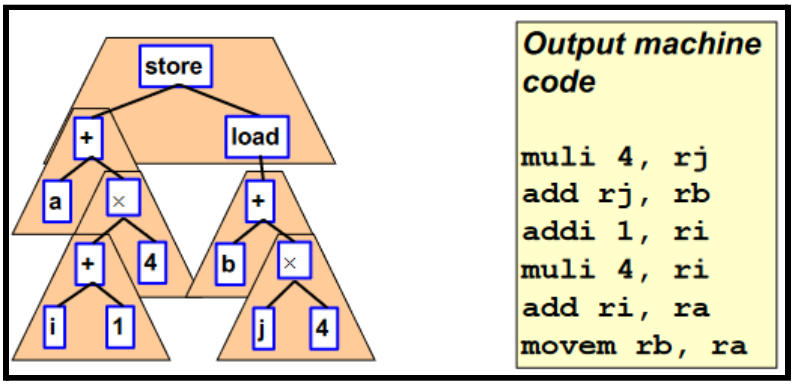
- If our goal is to find instructions with low overall cost, how do we find these patterns?
- The idea here is to use tree matching to go beyond "peephole matching"
    - This involves converting computation into a tree, and matching parts of the tree
- This is called a **dependence tree representation**
    - Edge from child to parent represents flow of values in constants / variables / temporary registers
    - A tree is build for each subsequence of IR instructions in a basic block with a *single root*
    - Typically corresponds to a single source code expression



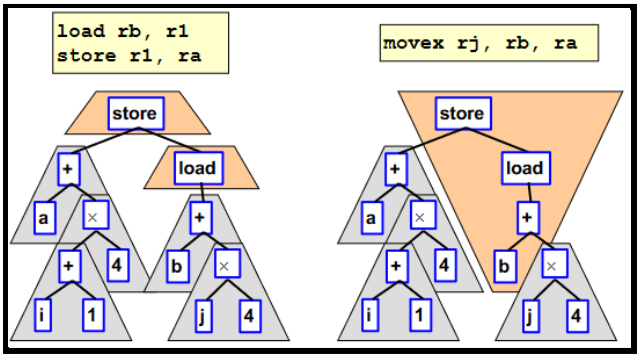
- Inside of a tree, a **tile** is a contiguous piece of the tree that corresponds to a machine instruction



- **Tiling** involves covering the tree with *tiles*



- Note that it is entirely possible to have multiple tilings for the same dependence tree…



Tiling

- Since there are multiple ways to tile a tree, we need to find the tree that minimizes the cost
- There are two ways to achieve this…
  - *Greedy Pattern Matching* - Making the most optimal choice every time

- *Optimal Dynamic Programming*

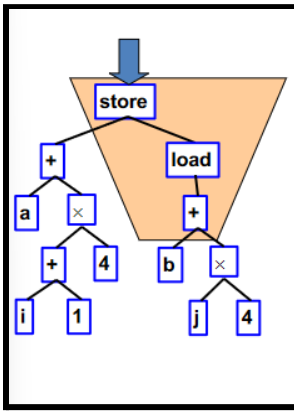
Greedy Algorithm

- Goal: Find a tiling with the lowest cost tiles
- Greedy top-down algorithm…
  - Start @ top of tree
  - Find largest tile that matches top node
  - Tile remaining subtrees recursively
  - Example…

```
Tile(n) {
 if ((op(n) == PLUS) &&
     (left(n).isConst()))
  {
    Code c = Tile(right(n));
    c.append(ADDI left(n) right(n))
  }
}
```

Optimal Dynamic Programming

- Divide a problem into smaller subproblems, then remember the solutions to these subproblems to help generate an optimal, overall solution
- **Memoization**: save previously computer solutions to sub-problems
- Can work top-down or bottom-up
- For the algorithm…
  - For each subtree, record best tiling in a table
  - At each node…
    - First check table for optimal tiling for this node
    - If none, try all possible tiles, remember lowest cost
    - Record lowest cost tile in table
    - Greedy, top-down algorithm



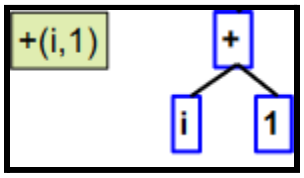

```
Tile(n) {
 if (best(n)) return best(n)
 // -- Check all tiles
 if ((op(n) == STORE) &&
     (op(right(n)) == LOAD) &&
     (op(child(right(n)) == PLUS)) {
    Code c = Tile(left(n))
    c.add(Tile(left(child(right(n))))
    c.add(Tile(right(child(right(n)))))
    c.append(MOVEX . . .)
    if (cost(c) < cost(best(n))
      best(n) = c
  }
 // . . . and all other tiles . . .
 return best(n)
}
```

A General Algorithm

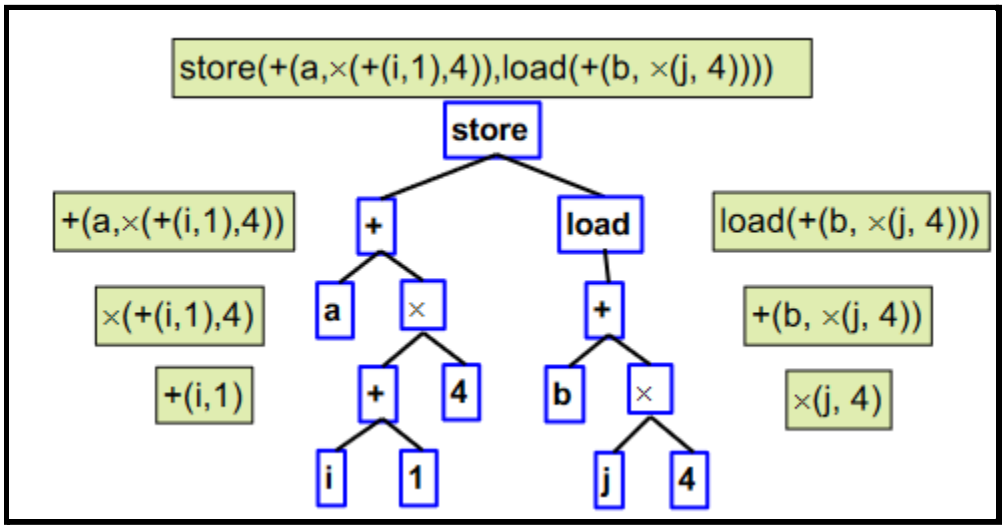
- Enumerating through every single possibility can be incredibly tedious

Tree Notation

- In order to accurately describe a tree, we use something called **prefix notation**



- By stringing these together, we can linearize a tree…

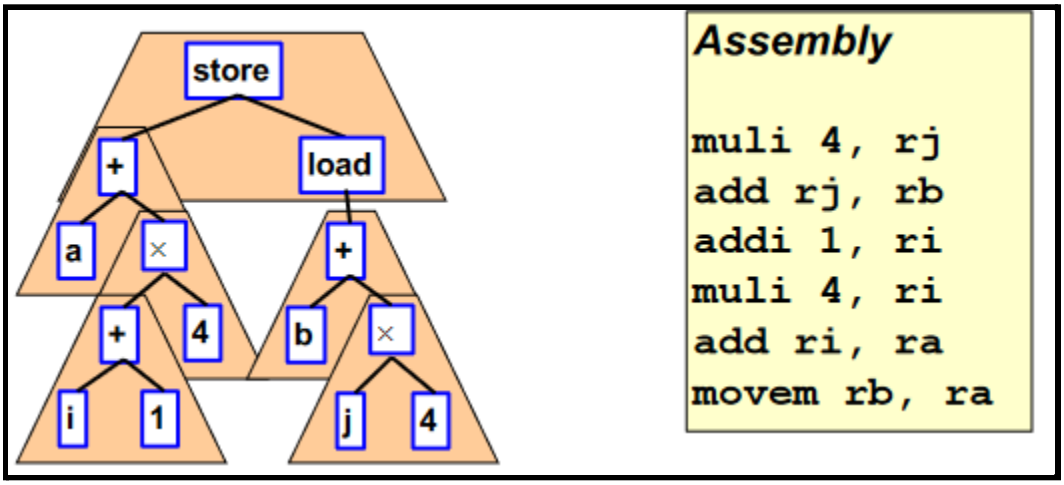


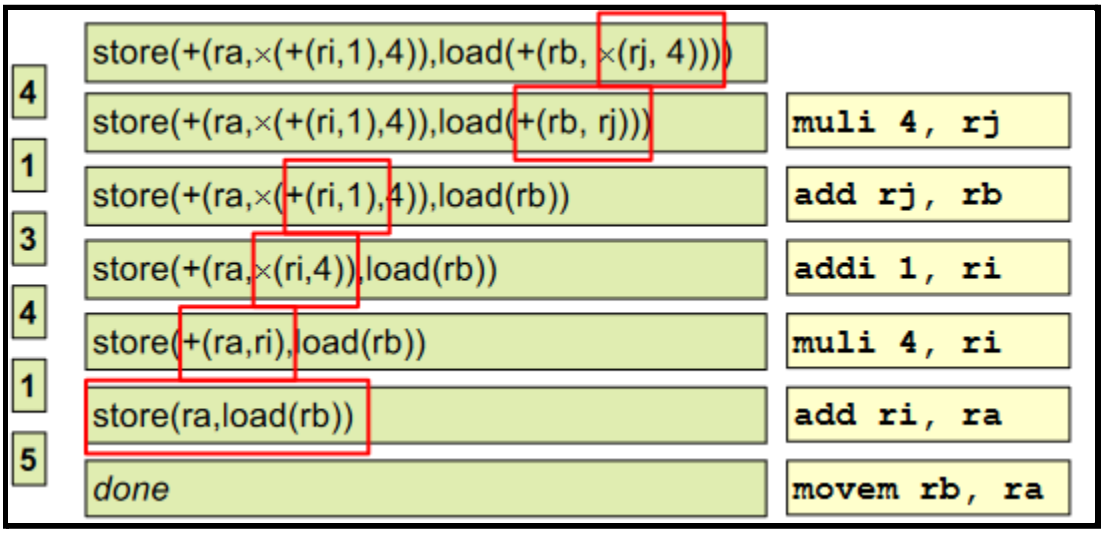
Rewriting Rules

Pattern, replacement	*Cost*	*Template*
$+(reg_1,reg_2) \rightarrow reg_2$	1	add r1, r2
$store(reg_1, load(reg_2)) \rightarrow done$	5	movem r2, r1

#	*Pattern, replacement*	*Cost*	*Template*
1	$+(reg_1,reg_2) \rightarrow reg_2$	1	`add r1, r2`
2	$\times(reg_1,reg_2) \rightarrow reg_2$	10	`mul r1, r2`
3	$+(num,reg_1) \rightarrow reg_2$	1	`addi num, r1`
4	$\times(num,reg_1) \rightarrow reg_2$	10	`muli num, r1`
5	$store(reg_1, load(reg_2)) \rightarrow done$	5	`movem r2, r1`

Rewriting Process





Lecture 10 - Peephole Matching & Register Allocation

- Exam Info: Midterm on March 16, 2022
    - Open note: only one-page, 4A-size note is allowed (both sides of the paper, but just one paper)
    - **Not** open computer

Peephole Matching

- Compiler can discover local improvements
    - Views a small set of adjacent operations
    - Move a "peephole" window over code, and search for improvement
- Examples…

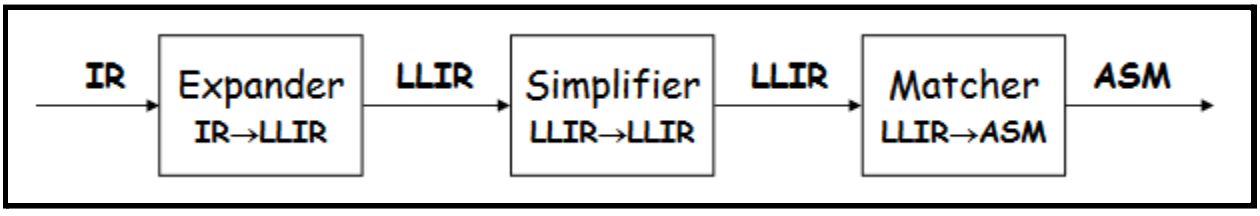
Simple example from textbook

Original code	Improved code
storeAI $r_1 \Rightarrow r_0,8$ loadAI $r_0,8 \Rightarrow r_{15}$	storeAI $r_1 \Rightarrow r_0,8$ i2i $r_1 \Rightarrow r_{15}$

Simple algebraic identities (adding 0 is always redundant; multiplying by 2 is equivalent to adding the same item to itself [note that multiplying is a much more expensive operation, which is why this is improved])

Original code	Improved code
addI $r_2,0 \Rightarrow r_7$ mult $r_4,r_7 \Rightarrow r_{10}$	mult $r_4,r_2 \Rightarrow r_{10}$
multI $r_5,2 \Rightarrow r_7$	add $r_5,r_5 \Rightarrow r_7$

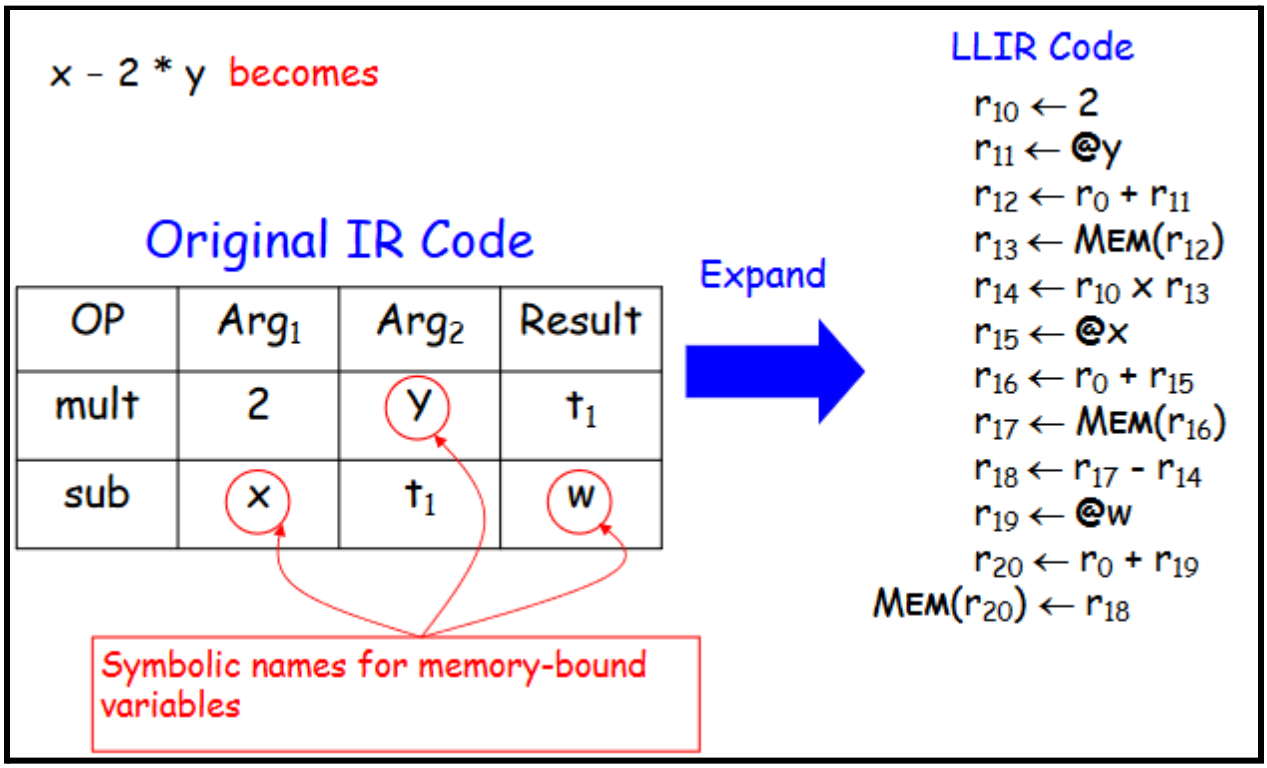
- In order to implement peephole matching, we need to break down the problem into three tasks…



    - ASM = assembly code
    - LLIR = low-level IR
- Expander
    - Turns IR code into a low-level IR (LLIR) such as RTL (Register Transfer Language)

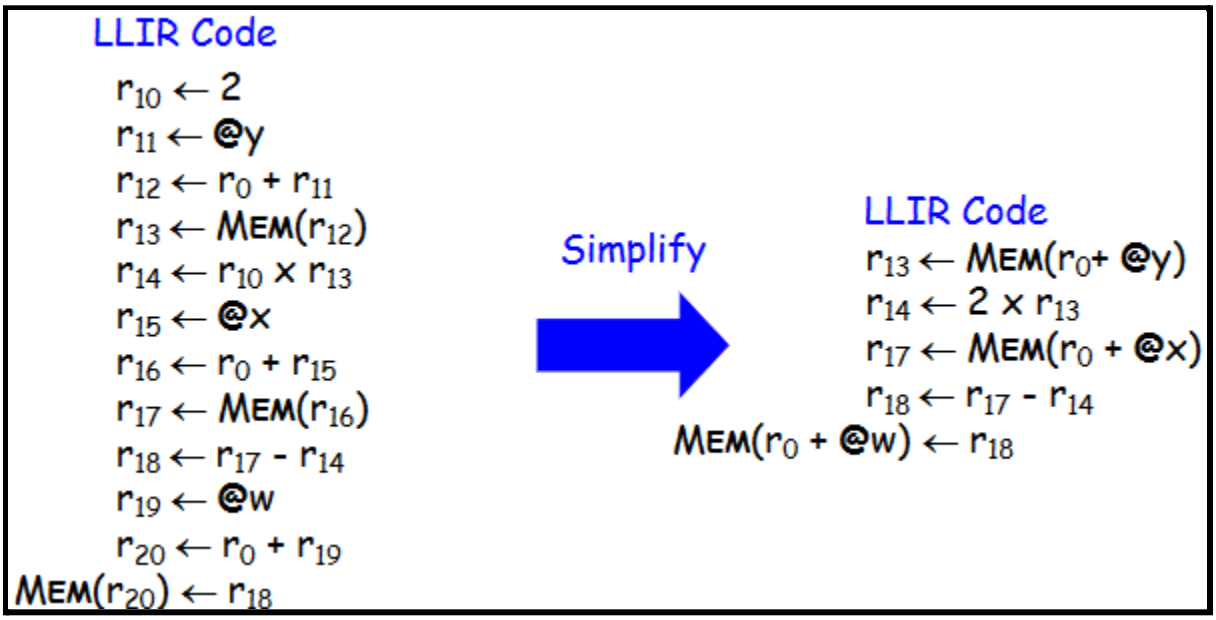
        - NOT the same as LLVM
    - Operation-by-operation or template-driven rewriting techniques
- Simplifier
    - The place where we actually use peephole optimization
- Matcher
    - Compares LLIR against a library of patterns
    - Picks low-cost pattern that captures effects
    - Must preserve LLIR effects, but may add new ones
- Let's look at an example starting from the expander…

Expander Phase



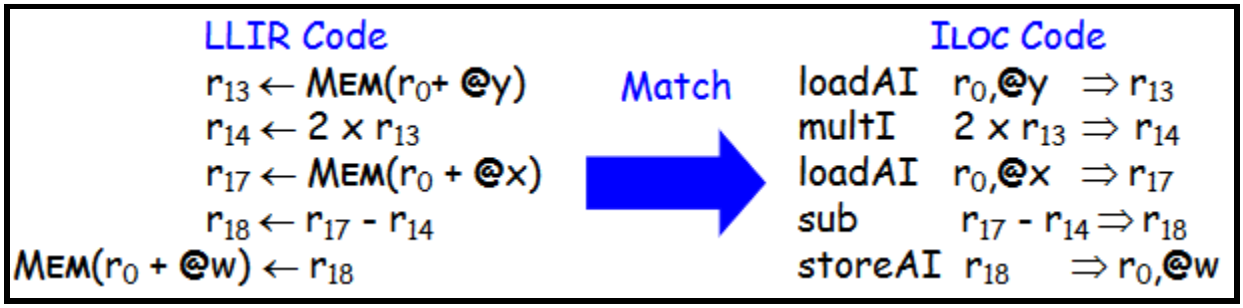
    - Notice that we did not simplify; this is delegated to the 2nd phase
    - Also, when we have a variable, we need to utilize the Base + Offset approach in order to grab the data from that variable

Simplifier Phase



- This is where we do our **peephole optimization**.
  - Take a blank slate. From the LLIR code, we add one instruction at a time. Once the instruction has been added, ask "*Can we optimize this from where we are*?" If so, optimize. Otherwise, add the next instruction. Do this until we reach the bottom.

Matcher Phase

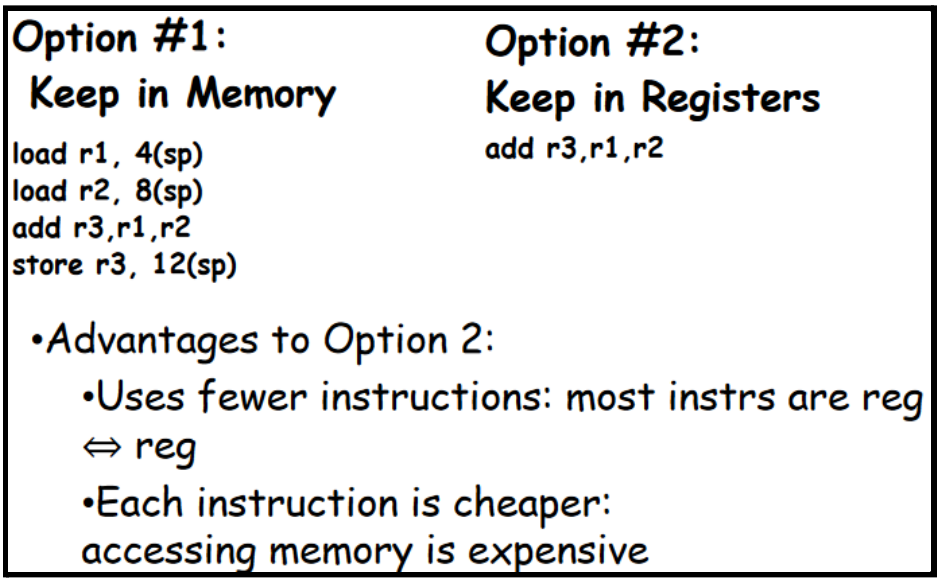


Review

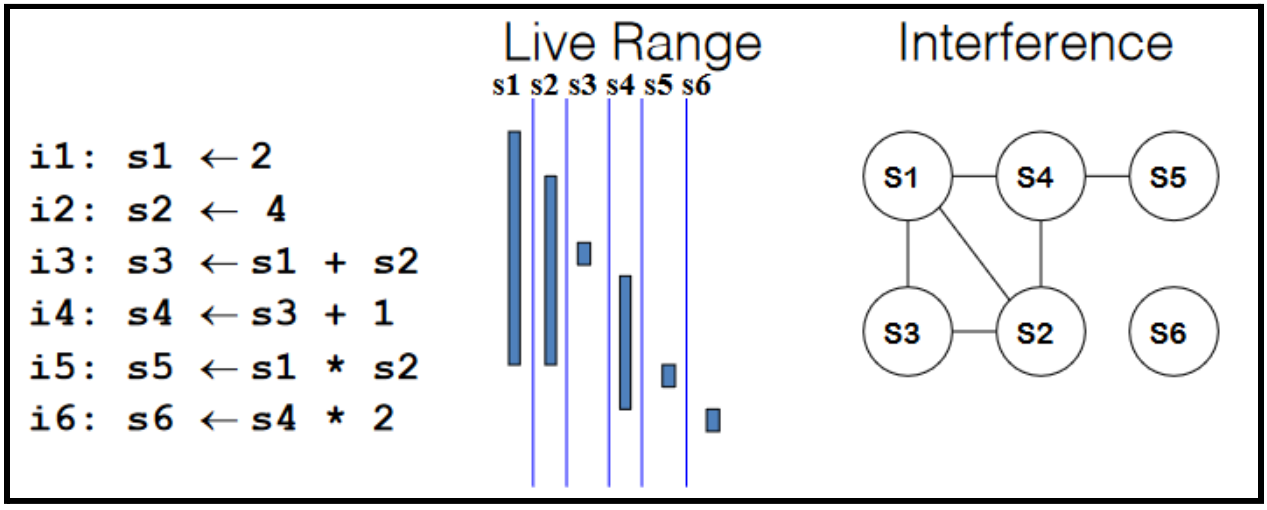
- For the **instruction selection** component of the back-end, we have talked about 2 ways to do this…
  - *Tree*
  - *Peephole Matching*
- After instruction selection, we should now have assembly code that we will use for the remainder of our backend operations
- Next up, we have **instruction scheduling** and **register allocation**

Register Allocation

- Input: IR w/ 3-address machine-level instructions (after instruction selection), but w/ an unbounded number of virtual/symbolic registers
- Output: For each virtual register *r*, we want to choose a physical machine register to store *r*'s values
- Intermediate step: Determine the live ranges of each virtual register *r*



- Goal: *Minimize* the traffic between the CPU registers and the memory hierarchy
- Remember that here in register allocation, even if we see optimizations possible, *that is not for register allocation to take care of*. We were simply trying to determine the # of registers required and then try to lower that if at all possible
- In order to do this, we need to determine the *live range*. Let's take a look at an example…



  - **Live**: variable at current program point will be used in the future before it is overwritten
  - **Dead**: variable at current program point will be overwritten before it is used

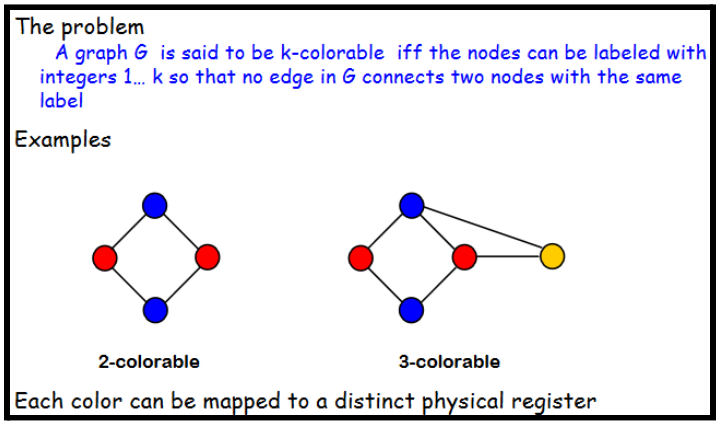
- **Live range**: set of program points where a variable is live (need not be a contiguous interval in general)
- Looking at the live range diagram, we can see which "registers" (registers in quotes because currently, we are working with symbolic registers)
- Looking at this, we can see that we only ever need (in this example) three registers at a time, so therefore we can do the following…

Intermediate Code w/ Symbolic Registers	Machine Code w/ Physical Registers
s1 ← 2	r1 ← 2
s2 ← 4	r2 ← 4
s3 ← s1 + s2	r3 ← r1 + r2
s4 ← s3 + 1	r3 ← r3 + 1
s5 ← s1 * s2	r1 ← r1 * r2
s6 ← s4 * 2	r2 ← r3 * 2

- So, in order to do register allocation, we need to be able to develop an **interference graph**

Building the Interference Graph

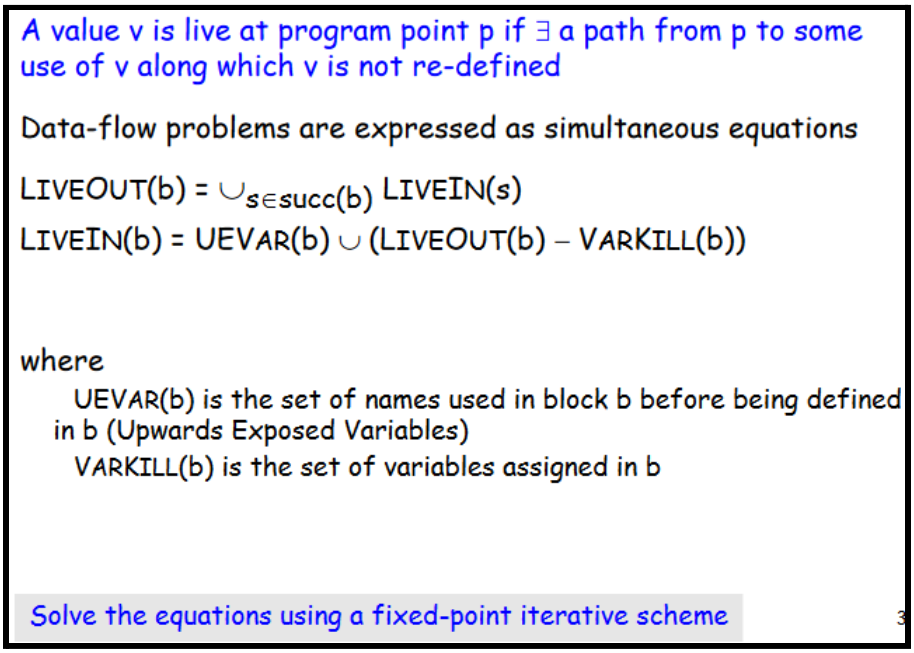
- Two values **interfere** if there exists a program point where both are simultaneously live
- If *x* and *y* interfere, they cannot occupy the same register
- The interference graph, $G_I = (N_I, E_I)$
- Nodes represent values, or live ranges
- Undirected edges represent interferences, and therefore we need to have separate registers while these interferences exist
- Graph Coloring



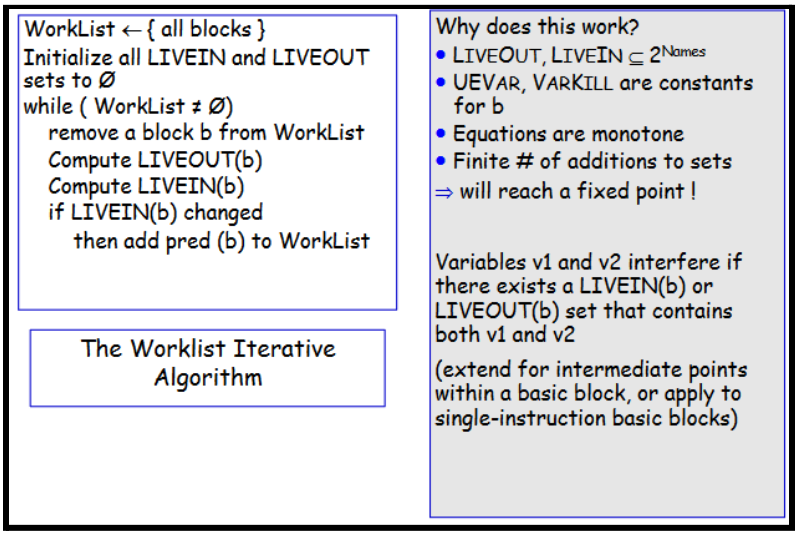
- This coloring technique is incredibly useful to determine the number of registers we need. How can we effectively color each node?

Computing LIVE Sets (live ranges)

- Equations for the sets used for the algorithm…

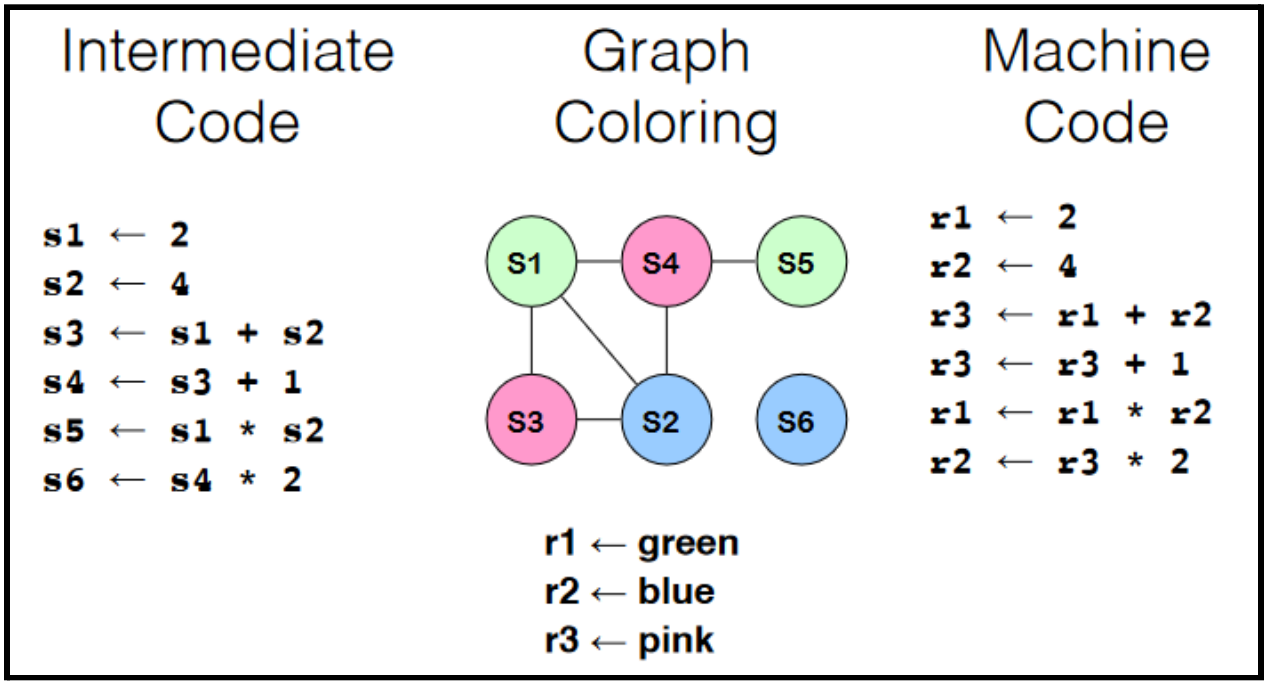


- The algorithm used to find the LIVE sets…



- This algorithm will give us the live ranges, but our goal is to color the graph, and we can do that using the live ranges
- So, with the information on live ranges, how do we color the graph?

Coloring the Graph



Register Allocation

1. Determine live ranges for each symbolic register
2. Determine overlapping ranges (**interference**)
3. Compute the benefit of keeping each live range in a register (**spill cost**)
4. Try to assign each live range to a machine register (**allocation**). If needed, **spill** or **split** live range
5. Generate code, including spills

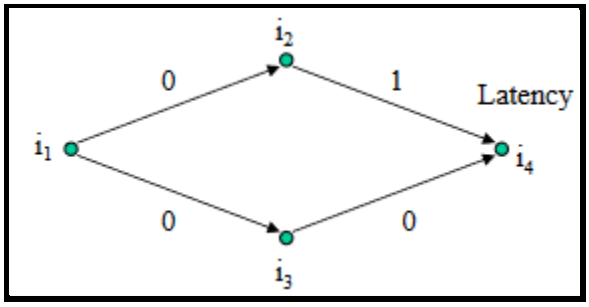
- We color it using something called **Chaitin's Algorithm**

Lecture 12 - Instruction Scheduling

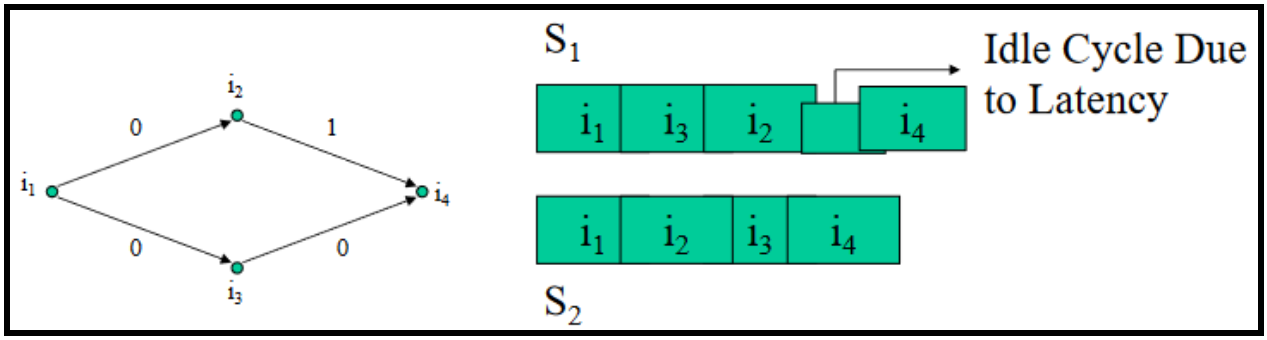
- Current computers use superscalar processors, which allow for multiple executions to occur at once
- If we can schedule our executions efficiently, we can optimize the number of clock cycles it takes for our code to execute
    - "*Given a source program P, schedule the operations so as to minimize the overall execution time on the functional units in the target machine*"
- Example: Suppose we wanted to LOAD and ADD. Usually, we'd say it takes 4 clock cycles (3 for LOAD, 1 for ADD, 3 + 1 = 4). However, with instruction scheduling, there may be a way to overlap the executions of these instructions such that it takes less than 4 clock cycles.

Operation	Cycles
load	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Local Instruction Scheduling Example
    - Think back to our value numbering, we had *local* and other bounds. *Local instruction scheduling* takes place in one basic block and is represented by an acyclic graph



    - Each node is an instruction and each edge corresponds to a *dependence constraint* labeled with *latencies*
- **Data dependence** $S_1 \rightarrow S_2$ exists between CFG nodes S_1 and S_2 with respect to variable X if and only if…
    - There exists a path from S_1 to S_2 in the CFG with no intervening write to X, and
    - At least one of the following is true:
        - (flow) X is written by S_1 and later read by S_2, or
        - (anti) X is read by S_1 and later is written by S_2 or
        - (output) X is written by S_1 and later written by S_2
- Based on the example above, there are two possible schedules…



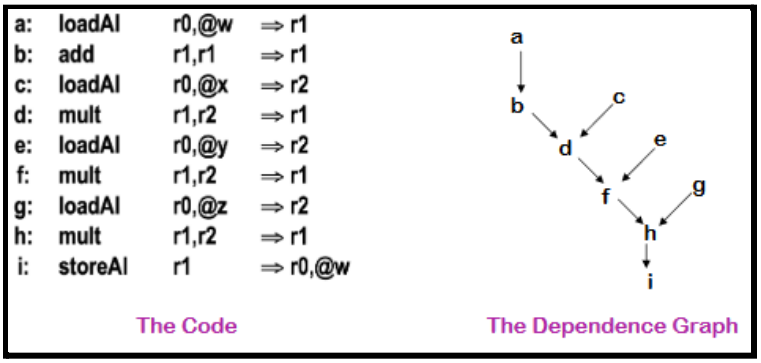
- The length of the schedule is the number of cycles required to execute the operations
- As we can see, we have an empty clock cycle in S_1 where nothing is being executed, but in S_2, during that downtime, we are executing i_3. This is what instruction scheduling is all about.

Formalizing the Instruction Scheduling Problem

- Input: DAG representing each basic block where…
  - Nodes encode *unit execution time* (single cycle operations)
  - Each node requires a definite class of FU
  - Additional time delays encoded as latencies on the edges
  - Number of FUs of each type in the target machine
- **Feasible Schedule**: A specification of a *start time* for each instruction such that the following are obeyed…
  - Resource: # of instructions of a given type of any time <= corresponding # of FUs
  - Precedence and LAtency: For each predecessor *j* of an instruction *i* in the DAG, *i* is started only *x* cycles after *j* finishes where *x* is the latency labeling the edge (*j*, *i*)
- Output: A schedule w/ the minimal overall completion time

Dependence Graph

- To capture properties of the code, build a dependence graph G
  - Nodes are operations w/ type(n) and delay(n)
  - An edge (x, y) belongs in G if and only if y uses the result of x



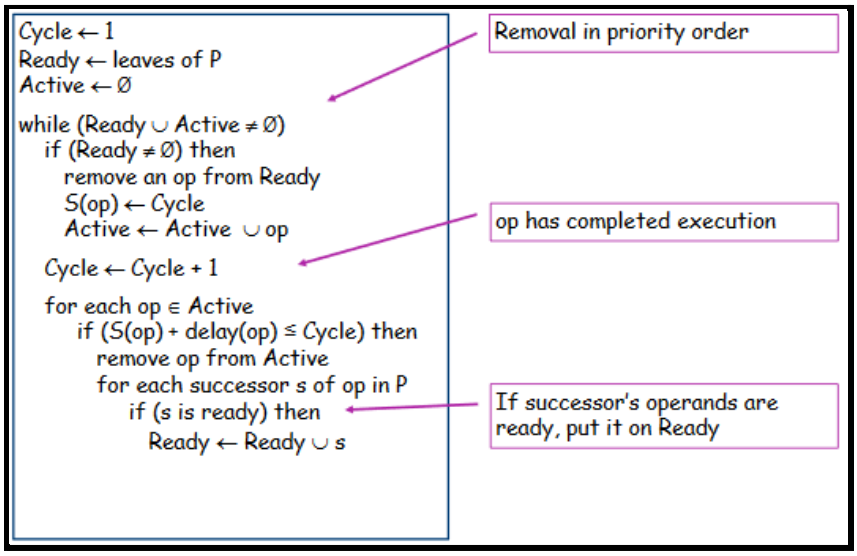
- A **correct schedule** S maps each node into a non-negative integer representing its cycle number, and…
  1. $S(n) \geq 0$, for all $n \in N$, obviously
  2. If $(n_1,n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$
  3. For each type t, there are no more operations of type t in any cycle than the target machine can issue
    - S is **time-optimal** if L(S) <= $L(S_1)$, for all other schedules S_1
- The **length** of a schedule S, denoted L(S), is…

$$L(S) = \max_{n \in N} (S(n) + delay(n))$$

- The idea is to find the shortest possible correct schedule

Algorithm

1. Build a dependence graph, P
2. Computer a priority function over the nodes in P
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle…
      i. Choose the highest priority ready operation & schedule it
      ii. Update the ready queue

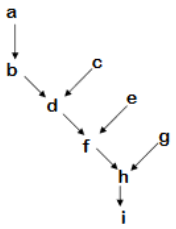


Example of the Algorithm

1. Build the dependence graph

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r2
f:  mult     r1,r2  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code

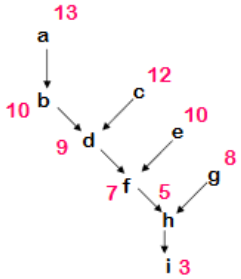


The Dependence Graph

2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r2
f:  mult     r1,r2  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

3. Perform list scheduling

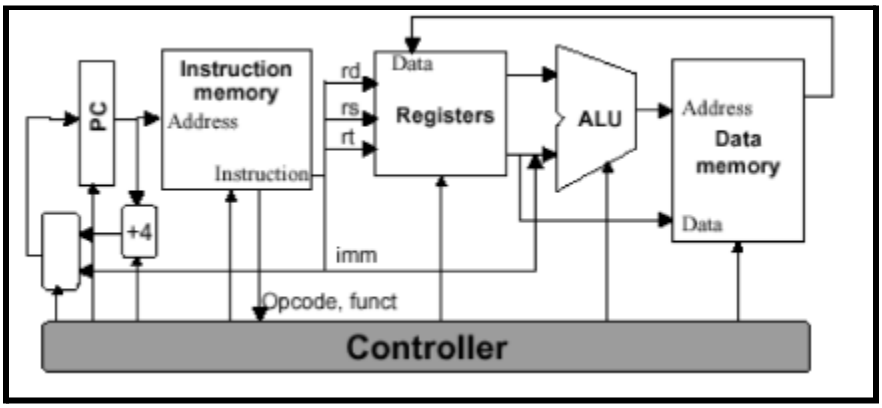
Lecture 13: MIPS Processor Architecture, SPIM Simulator

- So far, we've looked at the back-end of the compiler, and have looked at…
  - Instruction selection
  - Register allocation
  - Instruction scheduling
- Now, we are going to focus on the output of the compiler: the machine code itself.
- In computers, there are two types of instruction sets…
  - RISC (Reduced Instruction Set Computer)
  - CISC (Complex Instruction Set Computer)
- For us, we will be utilizing the MIPS architecture
  - 32-bit RISC processor
  - 32 32-bit Registers ($0 - $31)

Fetch-decode-execute cycle (FDX)

1. **fetch** the next instruction from memory
2. **decode** the instruction
3. **execute** the instruction

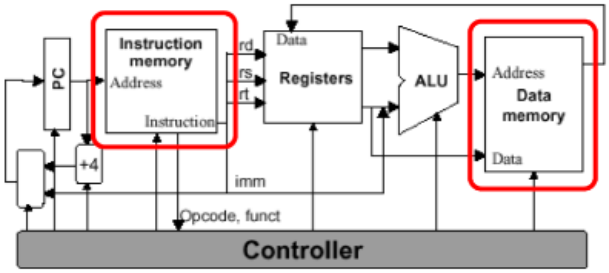
- Let's take a look at the datapath…



- Major components include…
  - Program counter (PC)
  - Instruction register (IR)
  - Register file
  - Arithmetic and logic unit (ALU)
  - Memory

Memory in MIPS

- In MIPS, memory is partitioned into two segments…
  - *Text segment* - "instruction memory", part of memory that stores the program (machine code), **read only**



    - *Data segment* - “data memory”, part of memory that stores data manipulated by program, **read/write**
- For reading/writing the data segment, MIPS uses BASE + OFFSET
    - *Load word* (read word from memory into register)

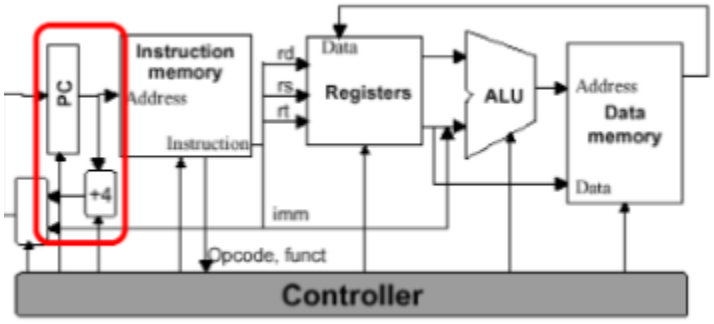
```
lw $t1,8($t2)    ⇒    $t1 := Memory[$t2+8]
```

    - *Store word* (write word from register into memory)

```
sw $t1,4($t2)    ⇒    Memory[$t2+4] := $t1
```

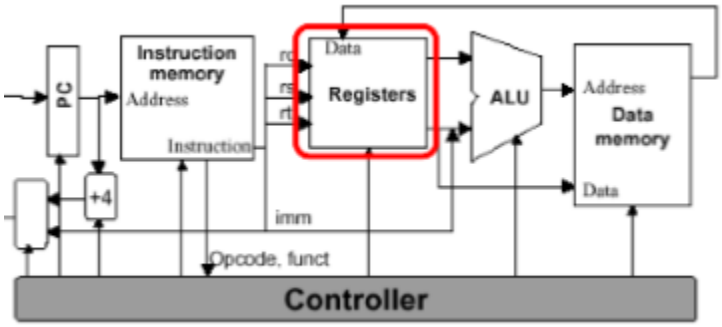
Program Counter (PC)

- Program counter is a register that stores the address of the next address to fetch



Register File

- The register file holds 32 32-bit registers

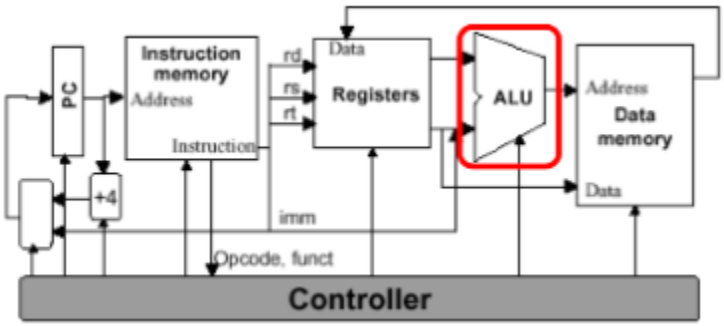


- By convention, certain registers are used for certain things…

Number	Name	Usage	Preserved?
$0	$zero	constant 0x00000000	N/A
$1	$at	assembler temporary	✗
$2–$3	$v0–$v1	function return values	✗
$4–$7	$a0–$a3	function arguments	✗
$8–$15	$t0–$t7	temporaries	✗
$16–$23	$s0–$s7	saved temporaries	✓
$24–$25	$t8–$t9	more temporaries	✗
$26–$27	$k0–$k1	reserved for OS kernel	N/A
$28	$gp	global pointer	✓
$29	$sp	stack pointer	✓
$30	$fp	frame pointer	✓
$31	$ra	return address	✓

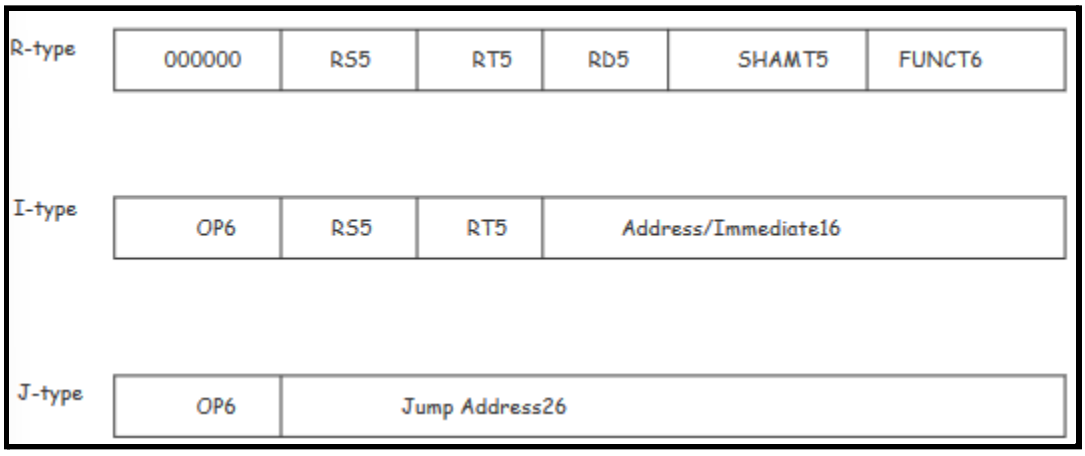
Arithmetic and Logic Unit (ALU)

- Implements binary arithmetic and logic operations
    - *Inputs*: 2 x 32-bit operands and a control signal for the operation
    - *Outputs*: 1 x 64-bit and the condition codes



MIPS Instruction Formats

- There are three types of instruction formats in MIPS…
    - *R format* - Uses three register operands, used by all arithmetic and logical instructions
    - *I format* - Uses two register operands & an address/immediate value. Used by load and store instructions, along with arithmetic and logical instructions with a constant
    - *J format* - Contains a jump address. Used by jump instructions



MIPS Addressing Modes

- MIPS utilizes many different addressing modes…
    - *Register* - Uses value in register as operand
        - Ex. $2 - uses register 2 as operand
    - *Direct Address* - Uses value stored in memory at given address
        - Ex. 100 - Uses value stored at location 100 in memory
    - *Register Indirect Address* - Uses value in register as address of operand in memory

        - Ex. ($3) - Uses value in reg. 3 as address of memory operand
    - *Indexed Address* - Adds address field and register value and uses this as address of operand in memory
        - Ex. 100($2) - Adds 100 to the value in register 2 and reads the operand from the resulting memory address
    - *Immediate Addressing* - Uses constant value contained in instruction
        - Ex. addi $1, $2, 4 - adds constant 4 to register 2 and stores result in register 1
    - *PC relative* - Address from instruction is added to the current value in the Program Counter
        - Ex. J 4 - Jumps to address PC + 4
- Some examples…

```
ADD $1,$2,$3
 — Register 1 = Register 2 + Register 3
SUB $1,$2,$3
 — Register 1 = Register 2 - Register 3
AND $1,$2,$3
 — Register 1 = Register 2 AND Register 3
ADDI $1,$2,10
 — Register 1 = Register 2 + 10
SLL $1, $2, 10
 — Register 1 = Register 2 shifted left 10 bits
```

MIPS Assembler Directives

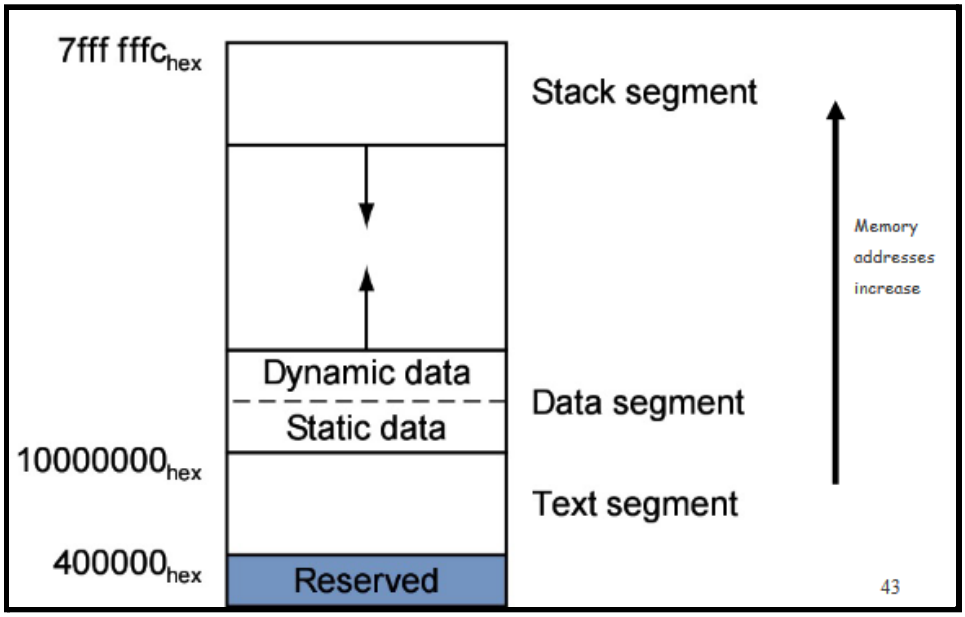
- Assembler directives (also called *pseudo ops*) are commands for the assembler that do not generate machine instructions
- Used to set up data and instruction areas

```
— LI $t1, 0xABCD1234    // Load immediate
— LA $t1, 0xABCD1234    // Load address
```

SPIM

- **SPIM** is a simulator that reads in an assembly program and models its behavior on a MIPS processor
- Note that a "MIPS add instruction" will eventually be converted to an add instruction for the host computer's architecture, it's just that this translation happens under the hood
- The simulator allows us to inspect register/memory values to confirm that our program is behaving correctly

Memory Layout



System Calls

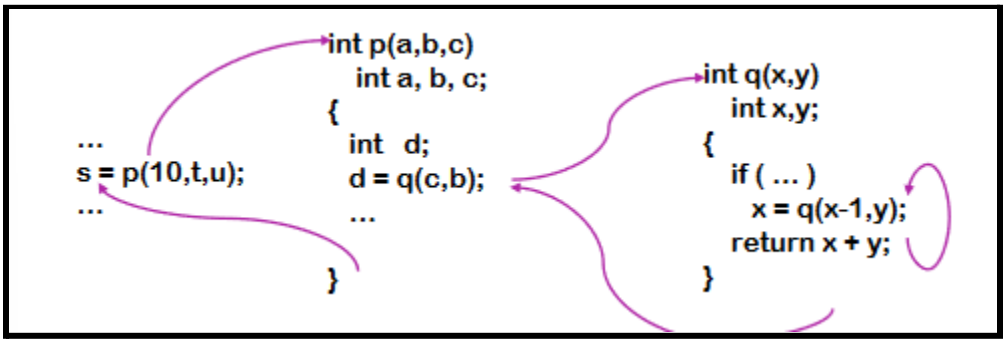
- SPIM provides some OS services, which include I/O operations such as read, write, file open, file close
- The arguments for the syscall are placed in $a0-$a3
- $v0 is also used for the syscall's return value

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		

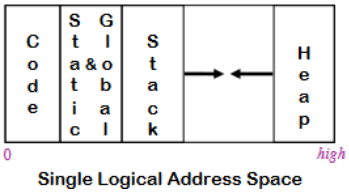
Lecture 14: Procedure Abstraction, MIPS Calling Convention

Procedural Abstraction

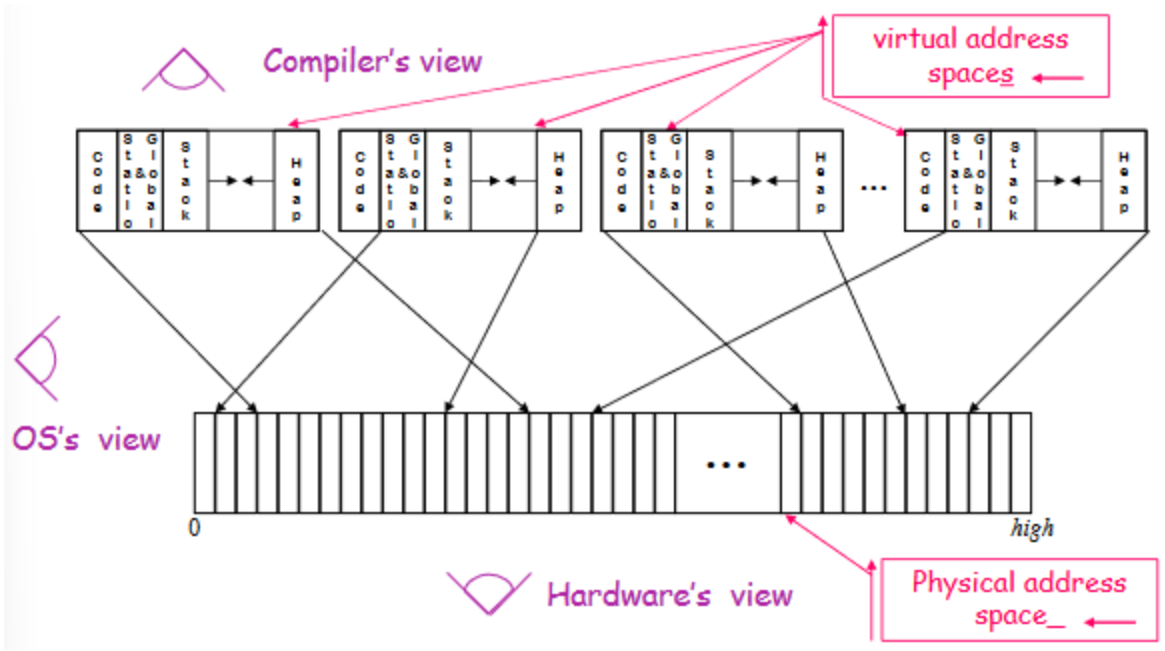
- **Procedures** (methods, functions, etc.) are abstractions that make programming more useful and practical
    - Information hiding
    - Distinct, separable name spaces
    - Uniform interfaces
- Hardware does little to support these abstractions, so we need our compiler to be designed in such a way to allow for procedures to be designed
- Because of this, the compiler decides basically everything…
    - Location for each value
    - Method for computing each result
    - Compile-time vs. runtime behavior
    - How to locate objects & values created & manipulated by code that the compiler cannot see
- There are three types of abstractions that the procedure handles…
    - *Control Abstraction*: Procedures have well-defined control-flow
        - Well-defined entries & exits, mechanism to return control to caller



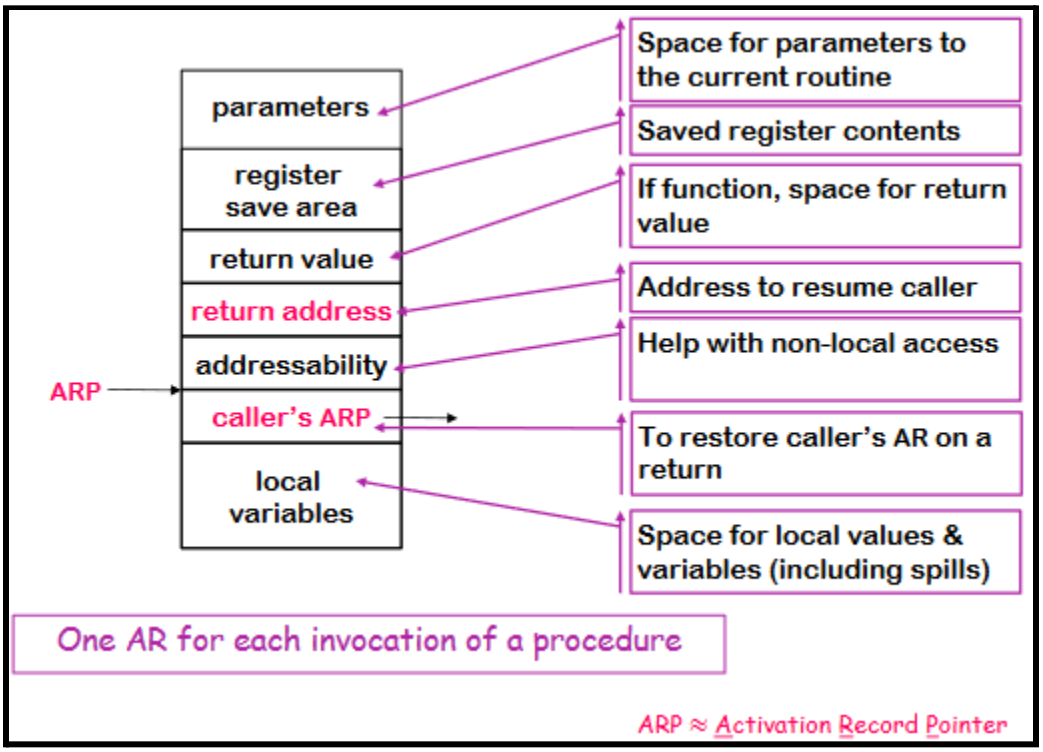
        - The above is an example of the Algol-60 procedure call…
            - Invoked @ call site w/ set of actual parameters
            - Control returns to call site, immediately after invocation
        - Maintaining this flow of control involves storing lots of data…
            - Requires code to save and restore a "return address"
            - Must map *actual parameters* to <u>formal parameters</u>
            - Must create storage for local variables
            - Need to preserve p's *state* while q executes
        - How can we keep these together?



Single Logical Address Space



- If we keep a *record* of everything we need about a procedure in memory, we can utilize procedures.
- An **activation record** is a block of information that is pushed onto the stack in memory that indicates all the required information pertaining to the method

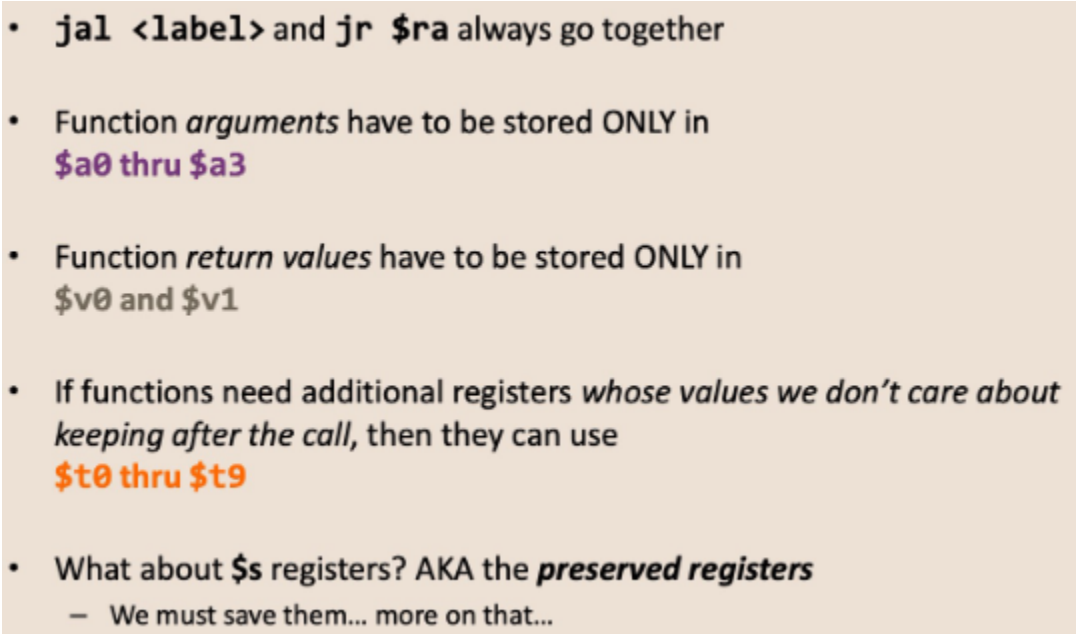


- *Clean Namespace*
  - Clean slate for writing locally visible names

        - Local names may obscure identical, non-local names
    - *External Interface*
        - Access is by procedure name & parameters
        - Clear protection for both caller & callee
- Procedures allow us to use *separate compilation*, which allows us to build non-trivial programs
    - Keeps compile times reasonable, requires independent procedures, etc.
- The procedure *linkage convention* (calling convention) ensures that each procedure inherits a valid run-time environment and he callers environment is restored on return

Calling Convention

- A **calling convention** is a protocol about how you call functions and how you are supposed to return them
- Each CPU architecture has one, and they can differ from one architecture to another
- Why have a calling convention?
    - Makes programming easier for everyone
    - Makes testing easier
- In MIPS, we do not have an inherent way of doing recursion
- Things to watch out for…



- Certain registers are *preserved*, in that they are required to hold the same value before **and** after a function returns
- Values held in *unpreserved* registers must always be assumed to change after a function call

Preserved vs Unpreserved Regs

- **Preserved:** **$s0 - $s7**, and **$sp , $ra**
- **Unpreserved:** **$t0 - $t9**, **$a0 - $a3**, and **$v0 - $v1**

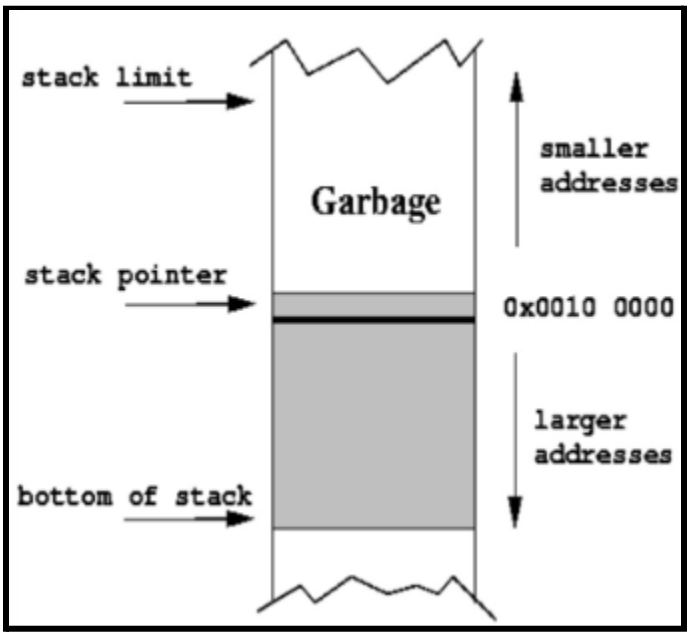
- By MIPS convention, certain registers are designated to be preserved across a call
  - *Preserved* registers are saved by…

    function called (e.g., $s0 - $s7)

  - *Non-preserved* registers are saved by…

    caller of the function (e.g., $t0 - $t9)

- Register values are saved on the *stack*, the top of the stack is held in **$sp** (stackpointer)
  - The stack grows from higher addresses to *lower* addresses

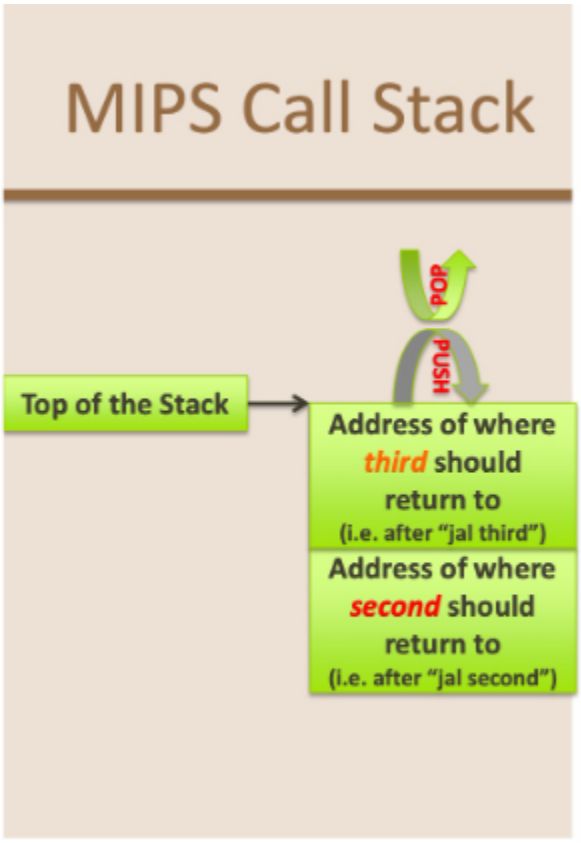


- The **stack pointer** is a register ($sp) that stores the address to the top of the stack
  - $sp contains the smallest address *x* such that any address lower than *x* is considered garbage
- In the above figure, $sp holds 0x0000 1000, and the shaded region is the valid part of the stack
- Some other vocabulary…
  - **Stack bottom**: the *largest* valid address of the stack (remember that the higher the value, the farther down the stack we go)
    - When the stack is initialized, $sp points to the stack bottom
  - **Stack limit**: the *smallest* valid address of a stack
    - If $sp gets smaller than this, we get a **stack overflow error**

MIPS Call Stack

- A **call stack** is used for storing the *return addresses* of various functions which have been called
- When you call a function, the address that we need to return is pushed into the call stack

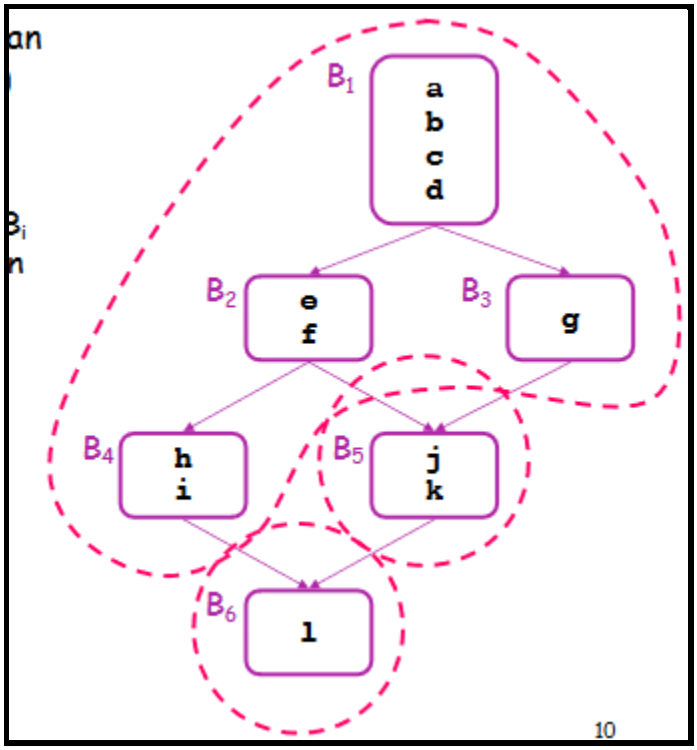
```
void first()
{
   second()
   return; }

void second()
{
   third ();
   return; }

void third()
{
   fourth ();
   return; }

void forth()
{
   return; }
```



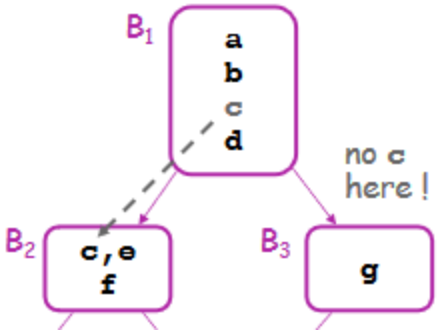

```
fourth:
  jr $ra

third:
  push $ra
  jal fourth
  pop $ra
  jr $ra

second:
  push $ra
  jal third
  pop $ra
  jr $ra

first:
  jal second

li $v0, 10
syscal
```

Lecture 15: Global Instruction Scheduling, Software Pipelining

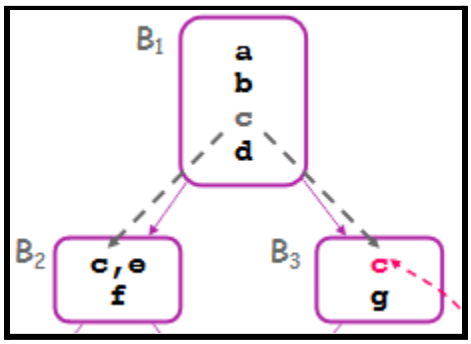
- To talk about global instruction scheduling, let's quickly review the different scopes leading up to global
- *Basic Block* - We are all too familiar with these
- *Extended Basic Block* - one step above the basic block; it's a maximal set of blocks such that each block other than the first one has exactly one predecessor. The example below can be split into three EBBs…



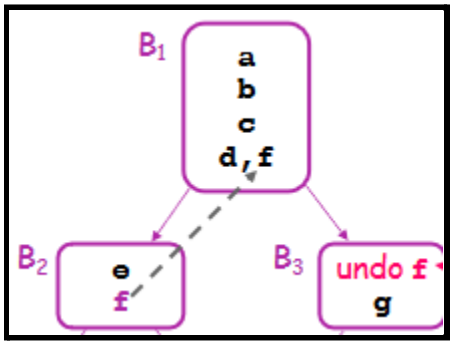
- Similar to the concept of *superlocal valuing*, from the middle-end segment of the course which used EBBs, we can also have **superlocal scheduling**, which schedules entire paths through EBBs
- But with our new superlocal scope, we run into new problems…
    - Moving out of B_1
    - Suppose we have an operation *c* inside B_1 and we find that it's better to move it into B_2, but B_1 has successors B_2, B_3



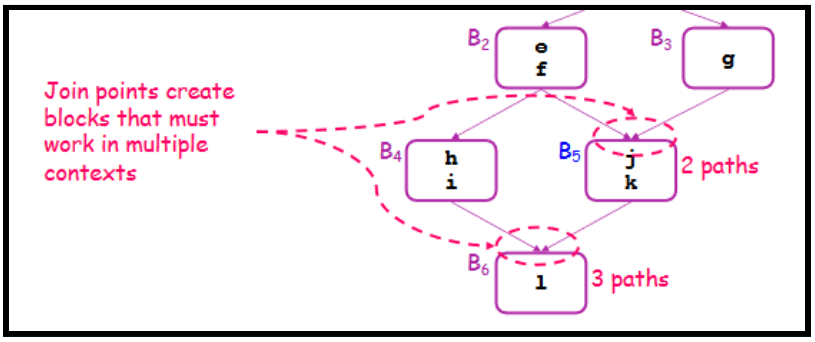
- We can't simply move *c* out of B_1 because then there exists situations where *c* never gets executed (i.e. if *c* is left out of B_3, then it's possible to reach B_6 and *c* is never executed, whereas *c* must always be executed)
- To alleviate this, we need to compensate by adding instruction *c* into B_3 as well



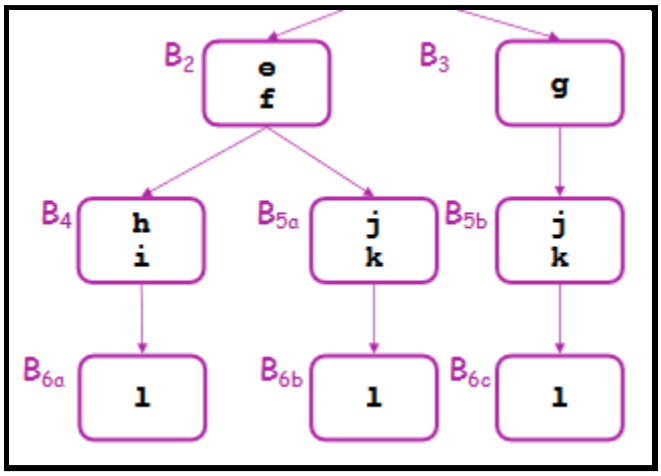
- Note however that this is only done for correctness, *not for speed*. Moving *c* into B_2 was for speed, but the compensation for B_3 was done for *correctness*.
- <u>Moving into B_1</u>
- This also introduces a problem, but it's the opposite.
- Suppose it's more efficient to run instruction *f* in B_1, so we move it from B_2 to B_1
- However, B_3 is never supposed to run *f*, but by having *f* in B_1, its possible for instruction *f* to be executed and also traverse B_3
- To alleviate this, we need to *undo f*



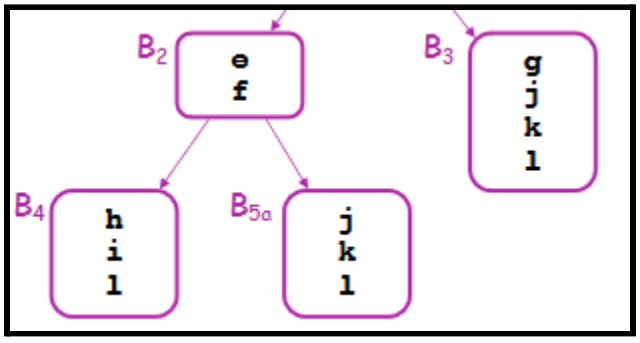
- Node Splitting



- There are many different contexts we'd have to account for when working with this CFG. In this case, it'd be easier to split up each context into its own path…

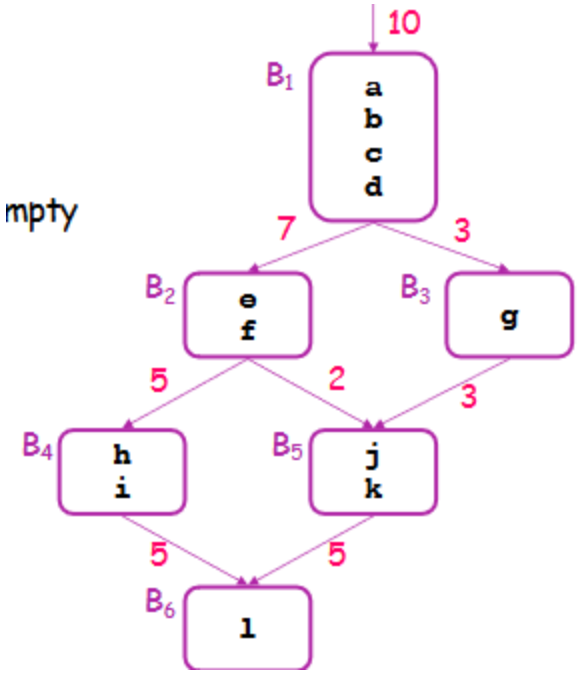


- After doing this, it's entirely possible that the resulting blocks can be combined…



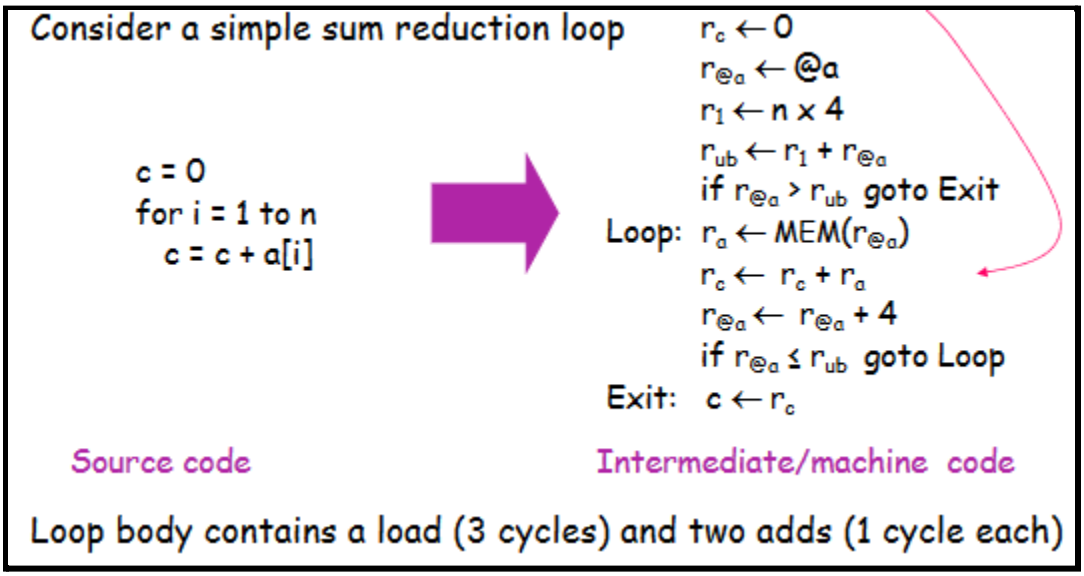
Trace Scheduling

- Start with execution counts for edges, and pick the hot path
- Schedule it accordingly (insert compensation code, etc.)
- Remove the hot path from the CFG
- Repeat until CFG is empty



Software Pipelining

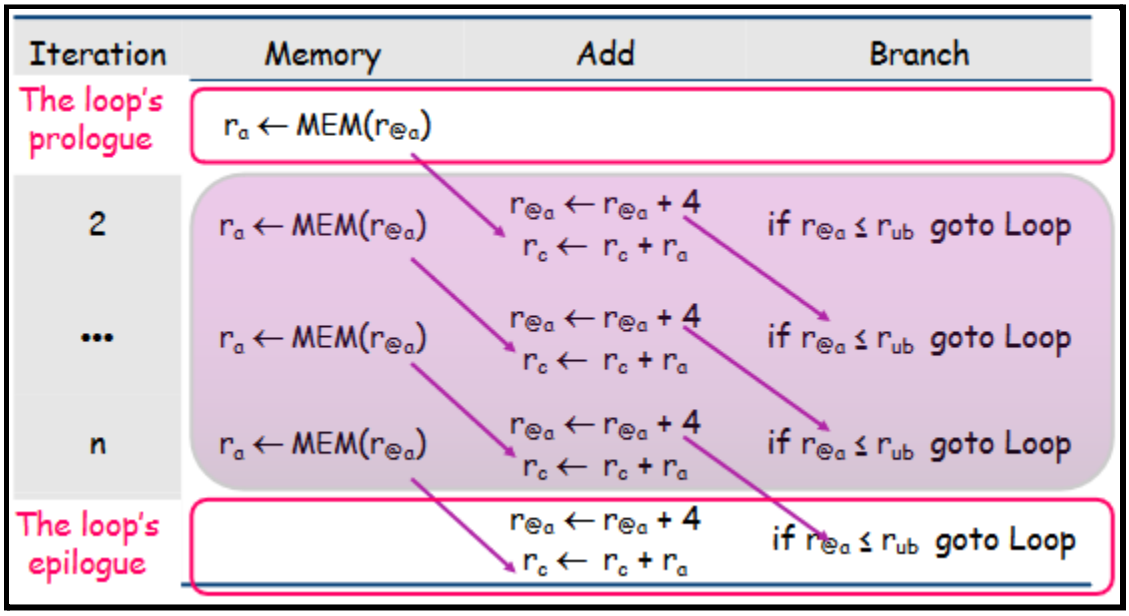
- Optimizes code in the presence of loops
- Software pipelining takes the original loop and transforms it
- Reduces the length of each iteration in a loop, called the "initiation interval" (II)
    - Initiation interval = # of clock cycles each iteration takes
- Does *not* reduce the total number of instructions, but it reduces the total number of cycles
- Let's look at an example to justify the purpose for software pipelining…



    - Remember, our goal here is to reduce the II, the number of clock cycles each iteration takes
- Let's examine the loop code itself: the load is 3 cycles, and we have two adds of 1 cycle each

$r_a \leftarrow MEM(r_{@a})$
$r_c \leftarrow r_c + r_a$
$r_{@a} \leftarrow r_{@a} + 4$
if $r_{@a} \leq r_{ub}$ goto Loop

- Think back to our instruction scheduling and remember: we aren't removing any instructions.
- Notice the dependency on line 1 and 2; you can't run like 2 without line 1 completing, but it takes 3 clock cycles for line 1 to finish.
- Furthermore, notice that line 3 does not depend on line 2 or 1.
- As such, we can run the most expensive operation first (line 1), and while that's cooking, run line 3 (since it doesn't depend on line 1), and then run like 2. Now, we're at 4 clock cycles.
- But we can actually do better. In fact, we can actually have this run in 2 cycles with software pipelining



How do we Get These 2 Clock Cycles

- To figure out how to do this, let's look at the algorithm for implementing software pipelining

1. Choose an initiation interval, ii
   - > Compute lower bounds on ii
   - > Shorter ii means faster overall execution
2. Generate a loop body that takes ii cycles
   - > Try to schedule into ii cycles, using modulo scheduler
   - > If it fails, bump ii by one and try again
3. Generate the needed prologue & epilogue code
   - > For prologue, work backward from upward exposed uses in the schedulued loop body
   - > For epilogue, work forward from downward exposed definitions in the scheduled loop body

- Step 1: Choose an initiation interval, ii
    - There are two lower bounds on ii…
        - *Resource Constraint* - ii must be large enough to issue every operation; a physical constraint
            - $max_u(ceiling\ of\ I_u/N_u)$ where I_u is the number of operations of type u and N_u is the number of functional units of type u
        - *Recurrence Constraint* - ii must be large enough to cover the latency around the longest recurrence in the loop
            - A **recurrence** is a loop-based computation whose value is used in a later itation of the loop
            - $max_u(ceiling\ of\ d_r/k_r)$ where k_r are the number of iterations where the loop computes a recurrence r, and d_r is the delay on r
    - To get an ii, we need to make an estimate based on our lower bounds
    - In our example, we'd be doing max(2, 1), so our ii is 2.
- Step 2: Generate a loop body that takes ii cycles
    - Now that we have an ii of 2,

Lecture 16: Static Single Assignment (SSA) Form

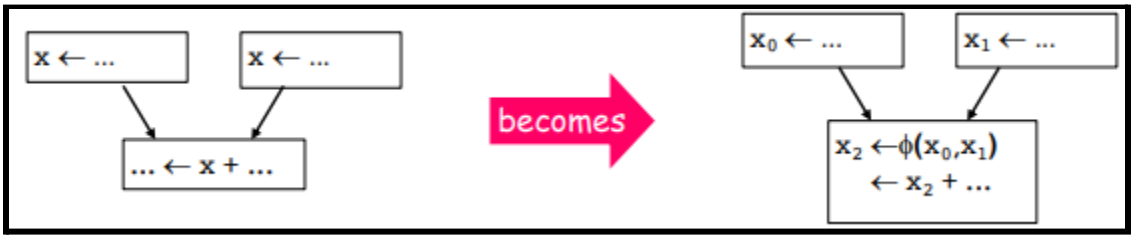
Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

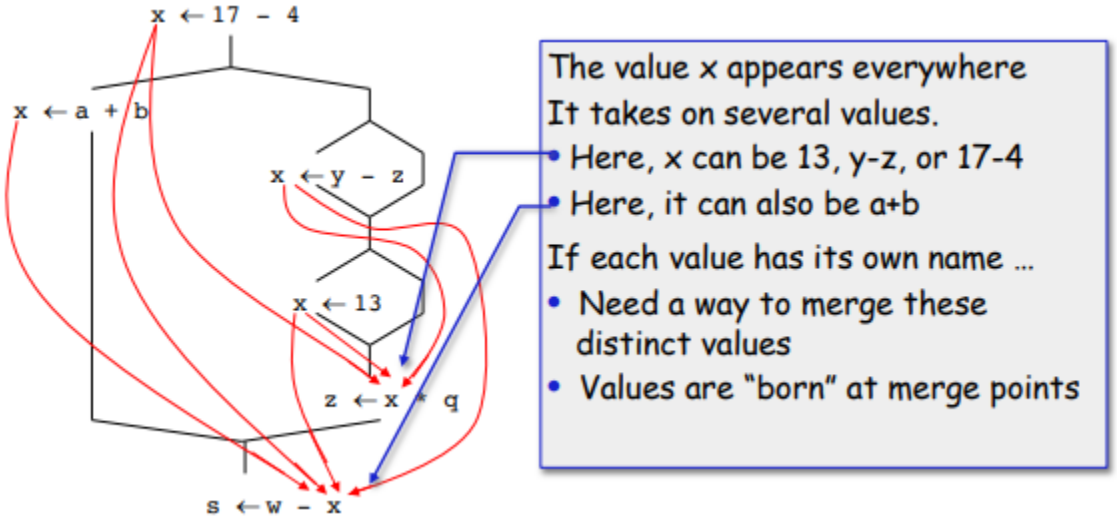
SSA-form

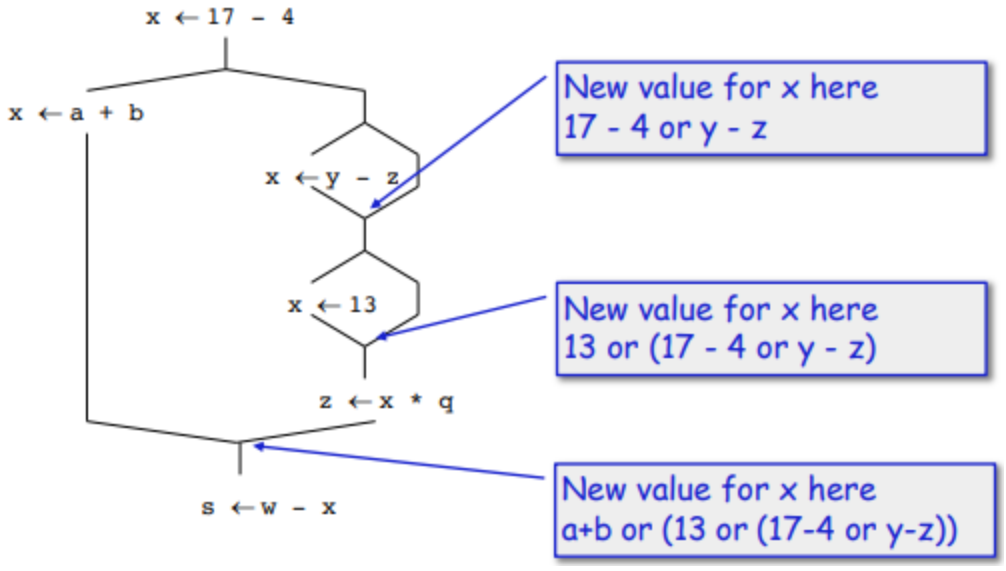
```
        x0 ← …
        y0 ← …
        if (x0 >= k) goto next
loop:   x1 ← ϕ(x0,x2)
        y1 ← ϕ(y0,y2)
        x2 ← x1 + 1
        y2 ← y1 + x2
        if (x2 < k) goto loop
next:    …
```

- Two principles…
  - Each name is defined by exactly one operation
  - Each operand refers to exactly one definition
- The problem with SSA is that sometimes we can have multiple definitions reach a certain site, and so we need to choose which definition reaches the use
- To alleviate this, we use a *phi node* and denote the fact that we choose whichever variable reaches that spot



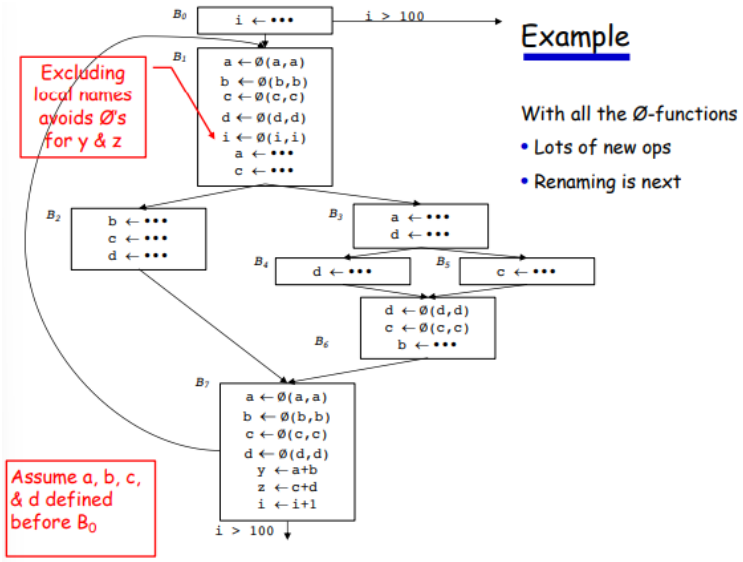
- How do we know where to place these *phi nodes*? We place them at **birth points** on the CFG

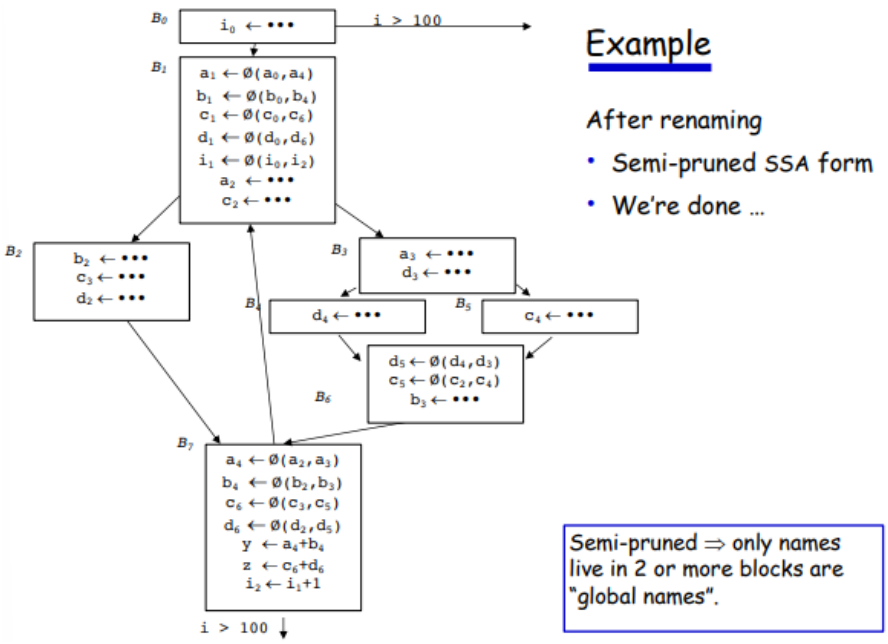




SSA Construction Algorithm

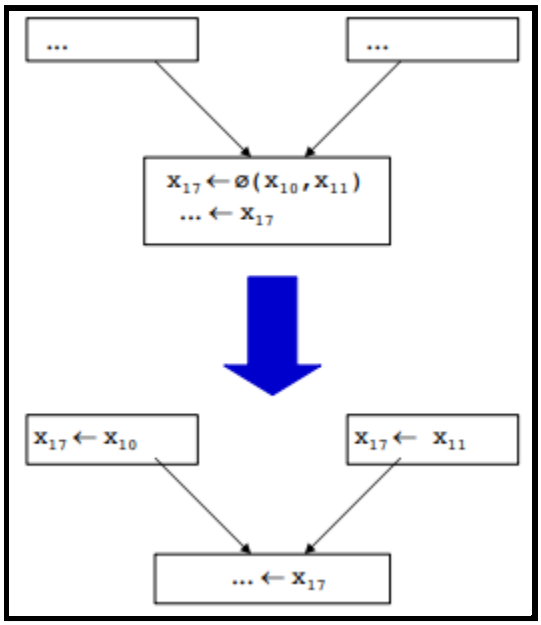
- 1. Insert *phi nodes* at every join for every name
    - This is very expensive though, so how about we only put phi nodes at places where they are viable? There are two ways to approach this…
        - *Semi-pruned SSA:* discard names used in only one block
            - Consider the definition as local, so we don't care about it
            - Cheap to compute
        - *Pruned SSA*: only insert *phi nodes* where their value is live
            - Requires global live variable analysis
            - More expensive
- 2. Solve reaching definitions
- 3. Rename each use to the def that reaches





SSA Deconstruction

- Once we're done, we need some executable code; we can't keep the phi operations
- To do so, insert the phi-function operands into the respective basic blocks

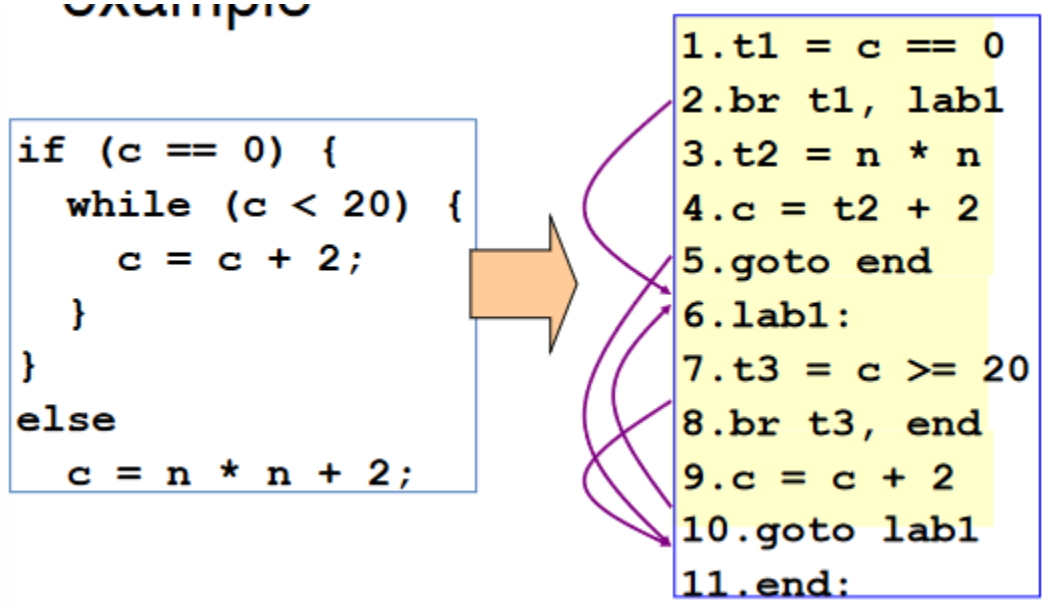


Lecture 17 - Midterm Review

- Midterm Coverage: Lecture 1 -16
- Time: 3/16 (3:30pm - 4:45pm), in class, 75 minutes
- Practice midterm (do this)
- 3 problems, 25 points each

Lecture 1

- Three major types of IR…
  - *Structural*: Graphically oriented, heavily used in source-to-source translators, large (Ex. Abstract Syntax Tree (AST))
  - *Linear*: Pseudo-code for an abstract machine. Simple, compact data structures (Ex. 3 Address Code, Stack machine code)
  - *Hybrid*: Bit of both (Ex. Control-flow graph (CFG))



  - A **control-flow graph** models the transfer of control in the procedure using nodes called *basic blocks*, which are maximal length sequences of straight-line code
  - **Dead code elimination** is conceptually similar to *mark-sweep garbage collection*
    - Marks useful operations, and everything else not marked is useless
    - Starts by marking **critical** operations (I/O statements,

```
Mark
1.  for each op i
2.     clear i's mark
3.     if i is critical then
4.        mark i
5.        add i to WorkList
6.  while (Worklist ≠ Ø)
7.     remove i from  WorkList
8.        (i has form "x←y op z" )
9.     for each instruction j that
10.    writes to y or z
11.     if j is not marked then
12.       mark j
13.       add j to WorkList
```

linkage code (entry & exit blocks), return values, calls to other procedures, global variables on program exit), then looks through data flow to mark instructions that the critical operations are dependant on

Lecture 2: Improved Dead-Code Elimination

- Only the last write to a variable matters when determining dead code, so we can edit our dead-code elimination algorithm

```
Mark
1.  for each op i
2.     clear i's mark
3.     if i is critical then
4.        mark i
5.        add i to WorkList
6.  while (Worklist ≠ Ø)
7.     remove i from  WorkList
8.        (i has form "x←y op z" )
9.     for each instruction j that
10.    writes to y (or z), and is not
11.    followed by a subsequent
12.    write of y (or z) before i
13.     if j is not marked then
14.       mark j
15.       add j to WorkList
```

- Getting these most recent writes we can use **reaching definitions**

» Def = Write to a variable in an IR instruction
  » An IR instruction typically has a single def, but there may be exceptions, e.g., a procedure call that updates multiple global variables
» Use = Read of a variable in an IR instruction
  » It is common for an IR instruction to have more than one use
» A definition d reaches program point u if there is a control-flow path from **d** to **u** that **does not** contain an intervening definition of the same variable as **d**
  » Implies that there may be some program execution in which the value of d may reach u; this is not a requirement for all program executions
  » Definition applies to any program point u, but we will be especially interested in the case when u corresponds to a use of the variable written by d

- Given a statement/instruction *S*, there are 4 sets we need to define…
  - Local Sets
    - GEN[S] = set of definitions in S ("generated" by S)

        - KILL[S] = set of definitions that are overridden by S
    - Global Sets
        - IN[S] = set of definitions that reach the entry point of S
        - OUT[S] = set of definitions in S as well as definitions from IN[S] that go beyond S; defs not killed by S

$$OUT[S] = GEN[S] \ U \ (IN[S] - KILL[S])$$

$$IN[S] = \cup_{p \in predecessors} OUT[p]$$

Lecture 3: General Roadmap for IR Optimizations

1. Look for an opportunity to modify IR to improve performance (e.g. dead code elimination)
2. Create an algorithm for performing the code transformation (e.g. worklist algorithm)
3. Identify analyses needed by the algorithm in step 2 (e.g. reaching definitions)
4. Define data flow equations to formalize the desired solution to the analyses in step 3 (e.g. IN[S], OUT[S], GEN[S], KILL[S])
5. Create an algorithm to solve the data flow equations in step 4
6. Use the analysis results from step 5 in the algorithm in step 2

- Sounds simple at first glance, but gets complicated once we have to handle cycles. In order to account for cycles in our algorithm, we need to use the **fixed point iteration method**
    - In step 5 of our roadmap, instead of making one pass to solve the data flow equations, we need to make multiple iterations
    - In fact, we need to do as many iterations as we need to until the IN[S] & OUT[S] from one iteration is the same as the IN[S] & OUT[S] from another iteration

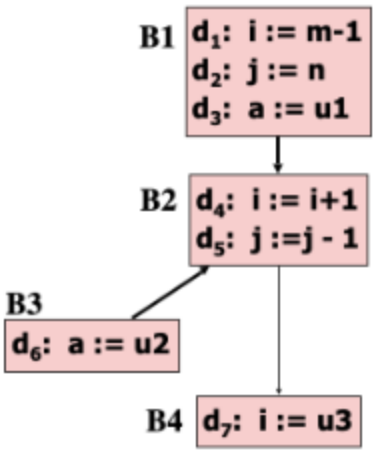
Reaching Definitions:

- // Initialize *out* under the assumpt1jion that $in = \emptyset$ by setting $out[B] := gen[B]$ for all the blocks //
- $change :=$ **true**

  // This initiates the iteration and if there is a change after the iteration in *any* of the *out* sets, then it remains true//
- While *change* remains **true** compute
  - $in[B] = \cup_{p \in P} out[p]$ where P is the set of all predecessors of block B
- $tempout := out[B]$
- $out[B] := gen[B] \cup (in[B] - kill[B])$
- if $out[B] \neq temoput$ $change :=$ **true**

- An example of the above algorithm…

Step 1: Compute gen and kill for each basic block



gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}

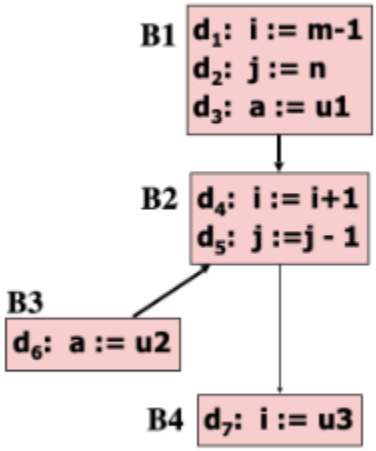
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}

gen[B3] = {d_6}
kill [B3] = {d_3}

gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

Step 2: For every basic block, make:

out[B] = gen[B]



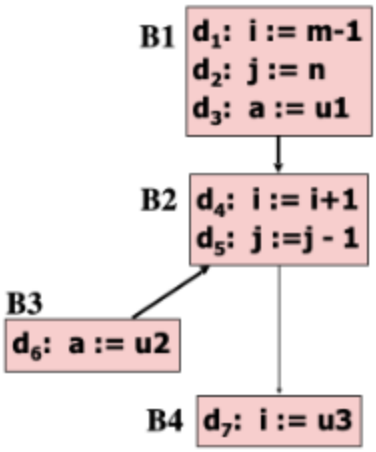
Initialization:

in[B1] = ∅
out[B1] = {d_1, d_2, d_3}

in[B2] = ∅
out[B2] = {d_4, d_5}

in[B3] = ∅
out[B3] = {d_6}

in[B4] = ∅
out[B4] = {d_7}



To simplify the representation, the in[B] and out[B] sets are represented by bit strings. Assuming the representation $d_1d_2d_3\ d_4d_5d_6d_7$ we obtain:

Initialization:

in[B1] = ∅
out[B1] = {d_1, d_2, d_3}

in[B2] = ∅
out[B2] = {d_4, d_5}

in[B3] = ∅
out[B3] = {d_6}

in[B4] = ∅
out[B4] = {d_7}

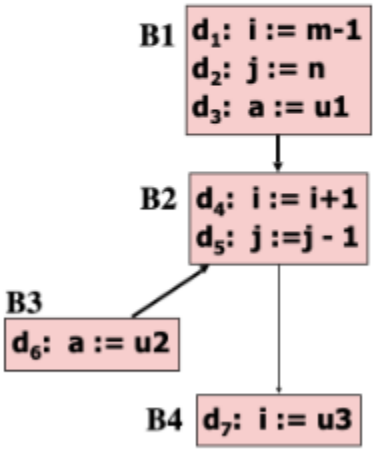
Block	Initial	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

Algorithm for Reaching Definitions

while a fixed point is not found:
in[B] = ∪ out[P] where P is a predecessor of B
out[B] = gen[B] ∪ (in[B]-kill[B])



Block	Initial	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	000 0000	000 1100
B_3	000 0000	000 0010
B_4	000 0000	000 0001

Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	000 0101

Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

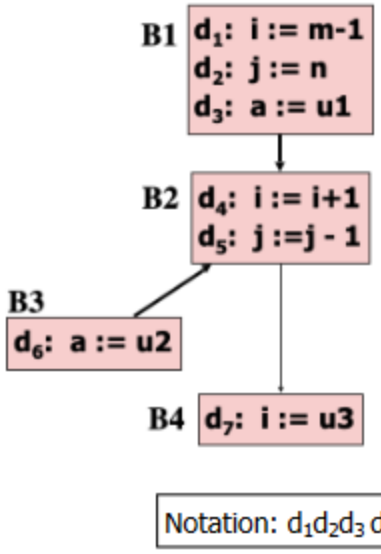
gen[B1] = {d_1, d_2, d_3}
kill[B1] = {d_4, d_5, d_6, d_7}
gen[B2] = {d_4, d_5}
kill [B2] = {d_1, d_2, d_7}
gen[B3] = {d_6}
kill [B3] = {d_3}
gen[B4] = {d_7}
kill [B4] = {d_1, d_4}

Algorithm for Reaching Definitions

while a fixed point is not found:

in[B] = ∪ out[P] where P is a predecessor of B

out[B] = gen[B] ∪ (in[B]-kill[B])



Notation: $d_1d_2d_3\ d_4d_5d_6d_7$

Block	First Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 0010	001 1110
B_3	000 1100	000 1110
B_4	000 1100	000 0101

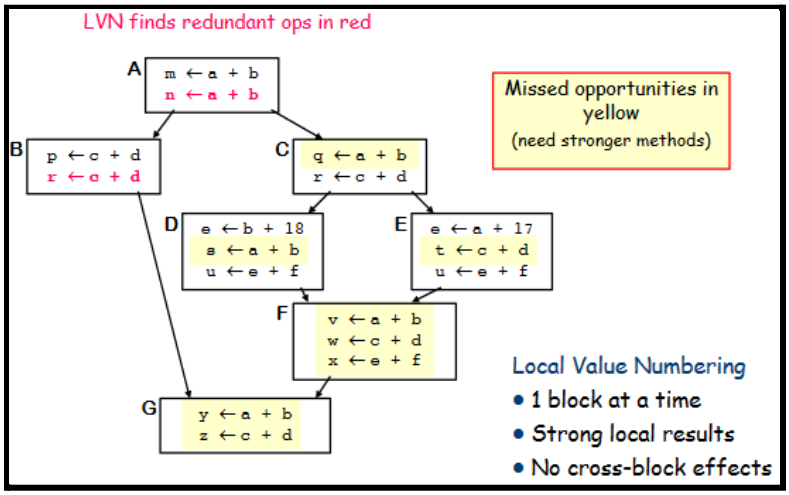
Block	Second Iteration	
	in[B]	out[B]
B_1	000 0000	111 0000
B_2	111 1110	001 1110
B_3	001 1110	000 1110
B_4	001 1110	001 0111

Improved dead-code elimination algorithm w/ reaching definitions

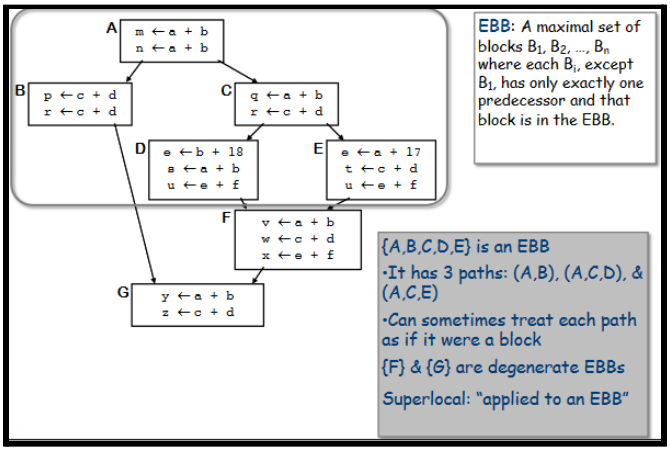
```
Mark
1.   for each op i
2.      clear i's mark
3.      if i is critical then
4.         // for simplicity, assume all
5.         // branch instructions are critical
6.         mark i
7.         add i to WorkList
8.   while (Worklist ≠ Ø)
9.      remove i from  WorkList
10.        (i has form "x←op y" or
           "x←y op z" )
11.     for each instruction j that
12.     contains a def of y or z that
13.     reaches i
14.        if j is not marked then
15.           mark j
16.           add j to WorkList
```

Lecture 4: Constant Propagation (Value Numbering)

- Idea: Assign an identifying "value number", V(i), to each expression (value) in IR instruction i. If two expressions have the same value number, they will always have the same value
- There are multiple types of value numbering…
    - *Local Value Numbering*: Works only on 1 basic block at a time, cannot reach other basic block

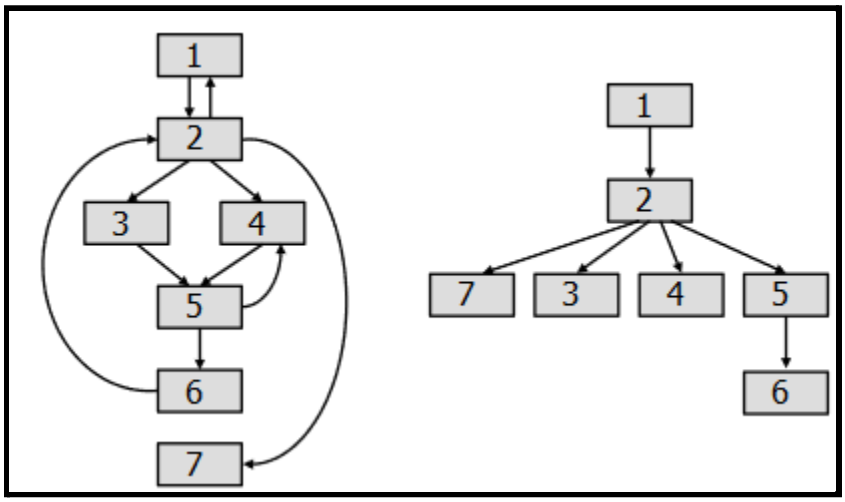


    - *Superlocal Value Numbering*: Works on an extended basic block



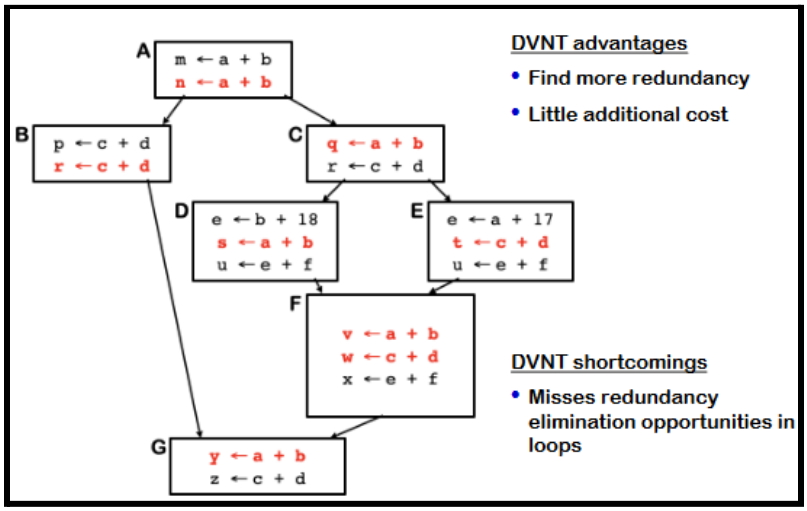
- Since there's still basic blocks that we can't reach, we need to somehow account for larger scopes. To do this, we need to use the concept of **dominators**
    - We say x **dominates** y if, and only if, every acyclic path from the entry of the control-flow graph to the node for y includes x
        - In other words, you cannot reach basic block G without having to go through basic block A. Every other basic block is unknown; there exist multiple paths beyond A to reach G, but regardless of the path you have to go through A
    - We call the closest dominator an **immediate dominator**

- NOTE: A basic block always dominates itself, but it cannot be its own immediate dominator
- Using the immediate dominator, we can create a *dominator tree*

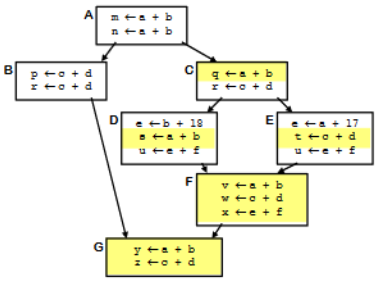


Lecture 5: Dominator Value Numbering & Available Expressions

- Now that we have dominators, we can use dominator value numbering in order to account for the entire CFG



- Even if we use DVNT, we can still miss some elimination opportunities
- An expression x+y is **available** if, and only if, along every path from the procedure's entry, it has been evaluated, and it's constituent subexpressions (x & y) have <u>not</u> been re-defined



AVAIL(A) = Ø

AVAIL(B) = {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b,c+d}

AVAIL(E) = {a+b,c+d}

AVAIL(F) = {a+b,c+d,e+f}

AVAIL(G) = {a+b,c+d}

- For a formal definition of available expressions, we need 3 different sets…
  - *Global Sets*
    - Avail(b) = the set of expressions available on entry to basic block b
  - *Local Sets*
    - DEExpr(b) = set of expressions computed in b and available on exit
    - ExprKill(b) = set of expressions killed in b

$$Avail(b) = \bigcap_{x \in pred(b)} (DEExpr(x) \cup (Avail(x) - ExprKill(x)))$$

preds(b) is the set of b's predecessors in the control-flow graph

  - NOTE: We don't look in basic block *b* to find it's available set, we use the set of predecessors
- Computing Available Expressions
  1. Build a control-flow graph
  2. Gather the initial (local) data — DEExpr(b) & ExprKill(b)
  3. Propagate information around the graph, evaluating the equation
  4. Use output of Available Expression analysis as needed, e.g., for lazy code motion or redundancy elimination

Lecture 18 - Midterm Review (cont.)

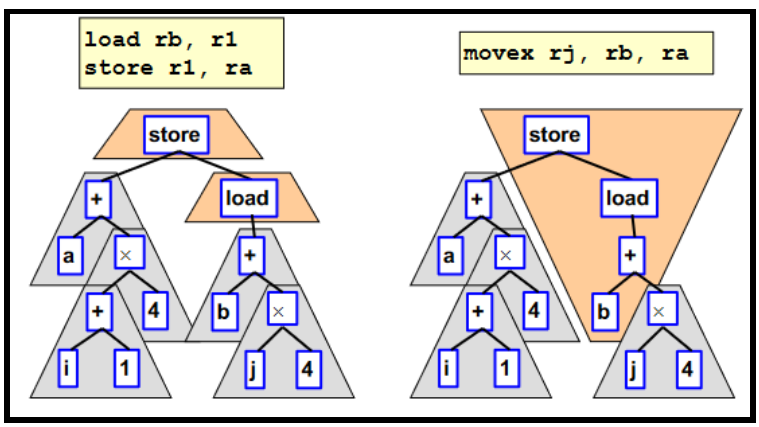
- Updated Midterm Exam
  - Lectures 1 - 6, 8 - 12, 16
  - One letter-sized paper, two sides allowed

Lecture 8: Copy Propagation

- Given an assignment $x = y$, replace later uses of x with uses of y, provided there are no intervening assignments to x or y
- Two types…
  - *Local Copy Propagation*
  - *Global Copy Propagation*: Have to take the entire CFG into consideration
    - Given a copy statement x = y, and use w = x, we can replace w = x with w = y if…
      - (1) x = y must be the only definition of *x* reaching w = x
      - (2) There may be no definitions of *y* on any path from x = y to w = x

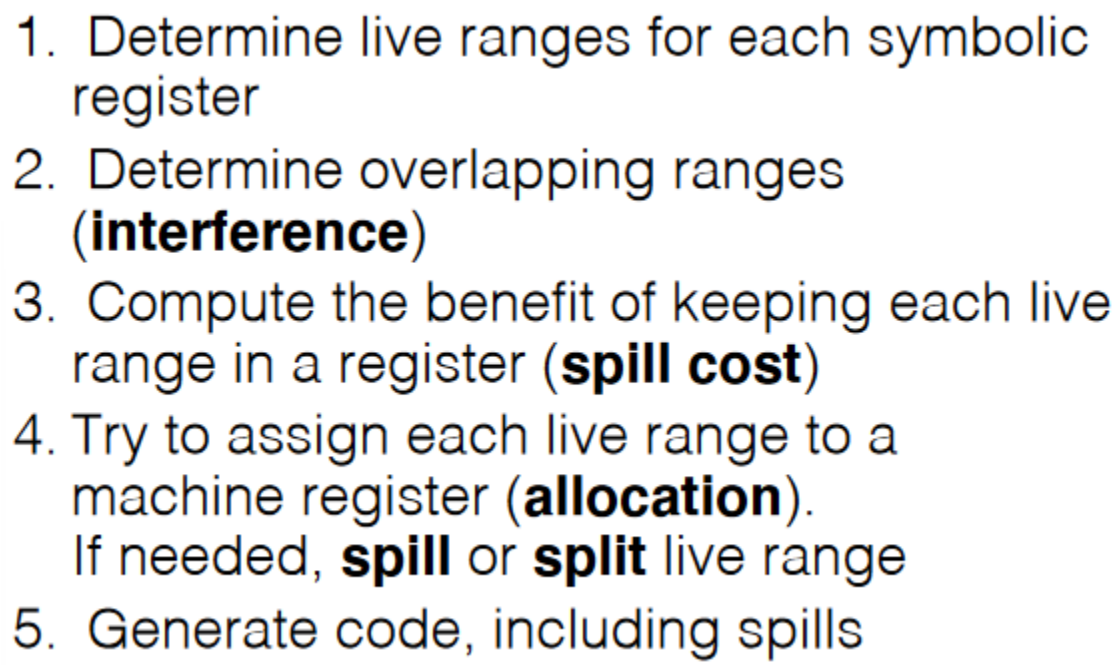
Lecture 9: Instruction Selection

- The purpose of the back-end is to convert IR into low-level assembly instructions (ISA)
  - IR: simple, uniform set of operations
  - ISA (e.g. MIPS): many specialized instructions
- Because ISA has more specialized instructions, we want to efficiently assign IR instructions to optimal ISA instructions
- One approach for this is a 1:1 mapping, where we map each IR operation to an instruction (or set of) instructions
  - Oftentimes inefficient and doesn't allow us to utilize the optimal instructions contained in an ISA
- A better approach is to use **dependence tree**, which involves converting our computation into a tree and matching parts of the tree to ISA instructions
  - The tree that we make is called a *dependence tree*
  - *Tiling* the tree is the process of converting parts of the tree to ISA instructions
  - However, there are multiple ways to tile a tree, so we need to find a way to *efficiently* tile such that our final instruction set has *minimal* cost



- Two ways to do this…
  - *Greedy Algorithm* - Start at the top, make the biggest possible tiling, continue downward
  - *Dynamic Programming* - Start at the bottom and sort of do the same thing, but instead of simply taking the largest, take the one with the lowest cost

Lecture 10 - Register Allocation



- Symbolic register = variable
- Use math3012 method to color interference graph

Lecture 12: Instruction Scheduling

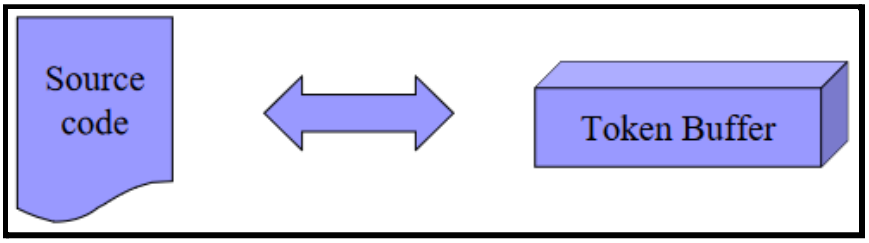
- A data dependence, S_1 depends on S_2, exists between CFG nodes S_1 and S_2 with respect to variable X if and only if…
  - There exists a path P from S_1 to S_2 in the CFG with no intervening write to *X*, and
  - At least one of the following is true…
    - (*flow*) X is written by S_1, and later read by S_2 or
    - (*anti*) X is read by S_1 and later written by S_2 or
    - (*output*) X is written by S_1 and later written by S_2
- For the purposes of this class, *assume we have infinitely many FUs* (CPUs)

Lecture 19 - Lexical Analysis (Scanning), Introduction to Parsing

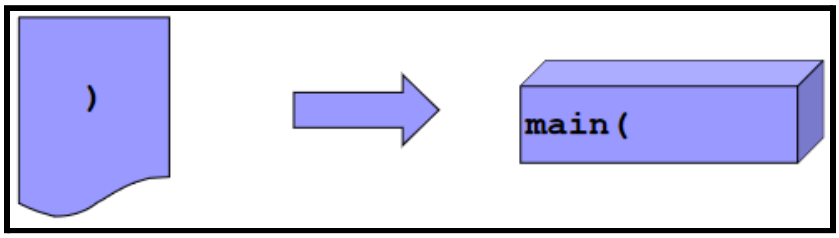
- Thus far, we have worked with the "middle end," which handles code optimizations through data flow analysis, and the "back end," which takes in this optimized IR and turns it into assembly code
- Now, we are going to work with the "front end," which uses…
    - **Scanners** - converts source code into stream of known chunks called **tokens**
        - *Lexical rules* of language dictate how legal *tokens* are formed as a sequence of alphabet symbols
    - **Parsers** - validates program based on grammar by building a *tree-based* representation of code
        - *Grammar* dictates how legal *tree* is formed as a sequence of tokens
    - **Semantic Analysis**
- Scanner reads things character by character and feeds tokens to the parser. The parser attempts to match the token based on specification of syntax as a context-free grammar

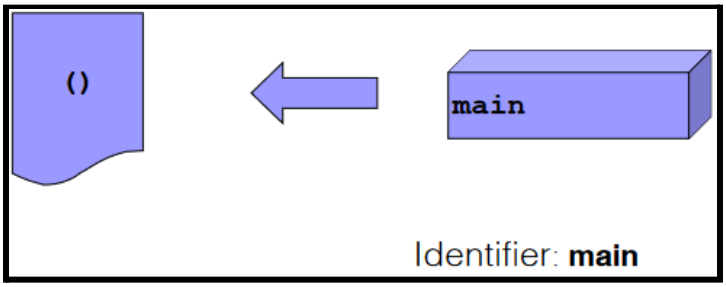
Scanning/Tokenization

- The scanner builds the token sequences; the parser tries to understand the meanings of these token sequences
- Scanners build blocks of tokens called *token buffers* which contain a token being identified



- It's a two way street because characters can both be read from a token and unread…





- Scanning can be difficult for a multitude of reasons…
    - Reserved words
    - Insignificant blanks: ex. do 10 i = 1.25
    - String constants w/ special characters: newline, tab, quote, etc.
    - Finite closures
- Scanning is also referred to as *lexical analysis*…
    - Read input one character at a time & group characters into tokens, remove white space && comments, and encode token types
    - A lexical language tells us which are legal strings in that language
    - Lexical rules specify how alphabet can be combined to form legal strings

Regular Expressions

- Symbols and Alphabet
    - A **symbol** is a valid character in a language
    - An **alphabet** is a set of legal symbols (denoted as summation)
- Strings and Languages
    - A **string** is a (possibly empty) sequence of symbols drawn from an alphabet
    - A **language** is a (possibly empty) set of strings
- **Metacharacters/metasymbols** that have special meanings…

  Construct regex's (e.g. **ϕ**, **ε**, **|**, **(**, **)**, *****, **+**, etc.)
  Treat as symbol by using escape character (\), e.g., \+

- For example…
    - Language, L(r), denoted by regex *r*

- Atomic regular expressions
  For each character **a** ∈**Σ**, **L(a) = { a }**
  Empty string: **L(ε) = { ε }**
  Empty set: **L(ϕ) = { }**
- Recursively-defined regular expressions:
  Alternation: for all **r1, r2** that are regexs,
  r1 | r2 is a regex that denotes
  L(r1 | r2) = L(r1) U L(r2)
  Concatenation: for all **r1, r2** that are regexs,
  r1 r2 is a regex that denotes
  L(r1 r2) = { s1 s2 | s1 ∈ L(r1) and s2 ∈ L(r2)}
  Repetition: for each **r** that is a regex, **r*** is a regex that denotes
  L(r*) = L(ε) U L(r) U L(r r) U L(r r r) U …

- Remember: A language is the set of all valid (possibly empty) strings that are made up of the valid *alphabet*
- Precedence of operations: Repetition is **tighter than** concatenation which is **tighter than** alternation, parenthesis change precedence
- Some examples of regex…

  - **a b | b a**
    - Matches **a b,** i.e., a b ∈ L(a b | b a)
    - Doesn't match **a a**
  - **(a | b) (b | a)**
    - Matches **a a**, **a b**, **b a**, **b b**
    - Doesn't match **a** or **a b a**
  - **(a a | b b)***
    - Matches **a a a a**, **a a b b a a**
    - Doesn't match **a b a b**

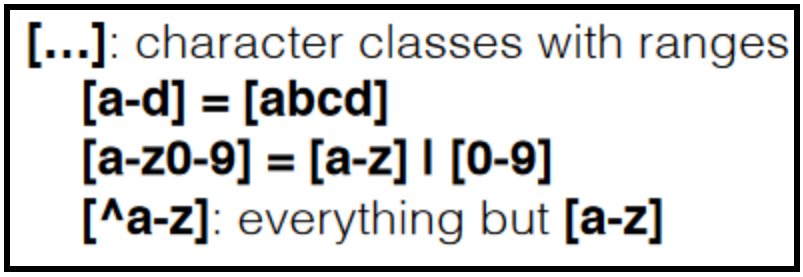
- Some regex shorthand notations for convenience…

  - **a? = (ε | a)**
    - Empty string or **a**
  - **a+ = a a***
    - Concatenation of one or more strings drawn from **a**
  - **[...]**: character classes
    - **[abcd] = (a | b | c | d)**

      [^abcd]: every symbol except **a, b, c, d**

      [a-z]: all lower case letters

      Note that square brackets [ ] have a different purpose from parentheses ( ). Parentheses in regular expressions are used to establish precedence as in arithmetic expressions, and can be nested.



- A period . means any character

Parser

- Parse the code and build a parsing tree in respect to the grammar
- If it cannot understand your input sequence, it will give a syntax error. If it can, it can either do nothing and get the next token *or* it can do some semantic action
- Here's an example of the grammar rules for C

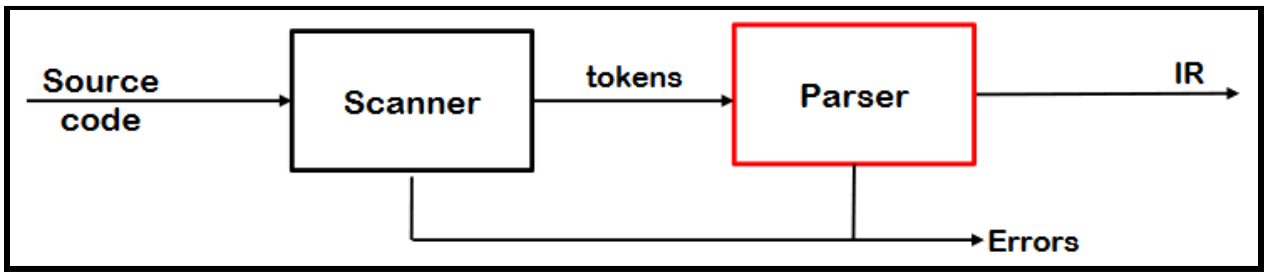
```
<C-PROG> → MAIN OPENPAR <PARAMS> CLOSEPAR <MAIN-BODY>
<PARAMS> → NULL
<PARAMS> → VAR <VARLIST>
<VARLIST> → , VAR  <VARLIST>
<VARLIST> → NULL
<MAIN-BODY> → CURLYOPEN <DECL-STMT> <ASSIGN-STMT>
  CURLYCLOSE
<DECL-STMT> → <TYPE> VAR <VARLIST>;
<ASSIGN-STMT> → VAR = <EXPR>;
<EXPR> → VAR
<EXPR> → VAR<OP><EXPR>
<OP> → +
<OP> → -
<TYPE> → INT
<TYPE> → FLOAT
```

- The job of the parser is to look at the grammar and look at the possible ways it can expand. To start, the parser prompts the scanner to give it the next token, and the token will scan to get the next token. Once it has the token, it passes it to the parser and the parser looks to see if it can recognize it

- Matching Slide
  - Start matching using a rule
  - When match takes place at certain position, move further (get next token & repeat)
  - If expansion needs to be done, choose appropriate rule (How to decide which rule to choose?)
  - If no rule found, declare error
  - If several rules found, the grammar (set of rules) is ambiguous

Lecture 20 : Context Free Grammars, Top Down Parsing

Parser: The Front End



- The parser does many things…
  - Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
  - Determines if the input is syntactically well formed
  - Guides checked at deeper levels than syntax
  - Builds an IR representation of the code
- The entire process for parsing is testing whether or not an input string belongs to a language using *context-free grammar*
- Therefore, for our parser to work we need to be able to develop a model for our syntax - the grammar
- **Context-free syntax** is specified with a *context-free grammar* and is usually denoted as a four tuple, G = (S, N, T, P)
  - *S* is the start symbol (set of strings in L(G))
  - *N* is a set of non-terminal symbols (syntactic variables)
  - *T* is a set of terminal symbols (words)
  - *P* is a set of productions or rewrite rules (P: N -> $(N \cup T)^+$)
- Suppose we have this complex grammar…

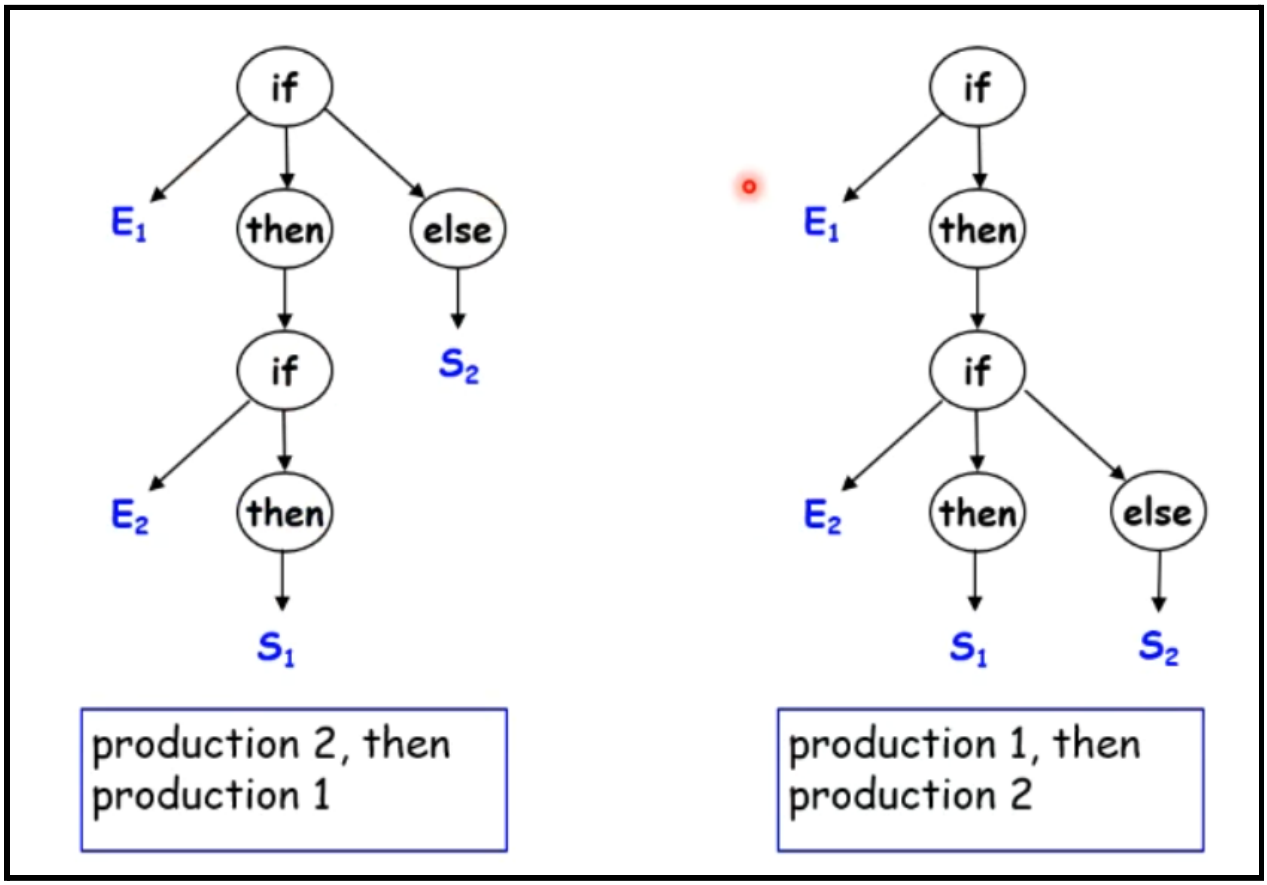
1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

- At each step, we need to choose a non-terminal to replace. There are two ways we can do this…
    - *Leftmost Derivation* - Replace leftmost NT at each step; used in top-down LL parsing
    - *Rightmost Derivation* - Replace rightmost NT at each step; used in bottom-up LR parsing
- If a grammar has more than one leftmost/rightmost derivation for a single sentential form, the grammar is **ambiguous**. For example (the if-then-else problem)...
    - Suppose you are given the task to check for *if-then-else*…

```
Stmt → if Expr then Stmt
     | if Expr then Stmt else Stmt
     | … other stmts …
```

    - At first this seems trivial, you just check for the different variations of the statement. However, notice that we are replacing "Stmt" and we also use "Stmt" in our variations. Consider the 1st Stmt on the 2nd line. If that statement is also an *if-then*, then our chain would be *if-then-if-then-else*, and then it's ambiguous where the else goes to



- To solve this, we need to be able to remove ambiguities by rewriting current grammar…

1	Stmt	→	if Expr then Stmt
2		|	if Expr then WithElse else Stmt
3		|	Other Statements
4	WithElse	→	if Expr then WithElse else WithElse
5		|	Other Statements

- In this grammar, we explicitly write it such that the inner statement for then *requires* to have an *else* otherwise it's not a valid instruction

Parsing Techniques

- There are two different parsing techniques…
  - *Top-down Parsers (LL [(L)eft-right, (L)eft-most derivation] Parsers)*
  - *Bottom-up Parsers (LR[(L)eft-right, (R)ight-most derivation] Parsers*
- We will primarily focus on the top-down parsers
- A **top-down parsers** starts with the root of the parse tree; the root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree
Repeat until the lower fringe of the parse tree matches the input string
1 At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
2 When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack
3 Find the next node to be expanded (label ∈ NT)

- The biggest problem with top-down parsers is that they cannot handle left-recursive grammars

Formally,
A grammar is left recursive if $\exists$ $A \in NT$ such that
$\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

Our expression grammar is left recursive
» This can lead to non-termination in a top-down parser
» For a top-down parser, any recursion must be right recursion
» We would like to convert the left recursion to right recursion

Non-termination is always a bad property in a compiler

- Since we don't want left recursion, we need to find a way to remove it from our grammar…

Consider a grammar fragment of the form

Fee → Fee α
| β

where neither α nor β start with Fee

We can rewrite this fragment as

Fee → β Fie
Fie → α Fie
| ε

where Fie is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

- We can simplify this to a general transformation…

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
			Expr - *Term*				*Term* / *Factor*
			Term				*Factor*

Applying the transformation yields

Expr	→	*Term Expr′*	*Term*	→	*Factor Term′*
Expr′	→	+ *Term Expr′*	*Term′*	→	* *Factor Term′*
		- *Term Expr′*			/ *Factor Term′*
		ε			ε

- However, this algorithm doesn't handle *every* instance of left recursion; it only handles *immediate* left recursion. If we want to handle more general, indirect left recursion, we need to use a more general algorithm…

arrange the NTs into some order $A_1, A_2, \ldots, A_n$
for i $\leftarrow$ 1 to n
    for s $\leftarrow$ 1 to i - 1
    replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
    where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for A_s
    eliminate any immediate left recursion on A_i
        using the direct transformation

Lecture 22 - LL(1) Parsing

Predictive Parsing

- Generally, given $A \rightarrow \alpha \mid \beta$, the parser should be able to decide between α & β
- To do this, we will be utilizing something called **first sets**
    - For some rhs $\alpha \in G$, **first**(α) is the set of tokens that appear as the first symbol in some string that derives from α. Ex. $x \in First(\alpha)$ if $\alpha \Rightarrow^* x\gamma$, for some γ
- After we have the sets, we must state the **LL(1) Property**…

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, then

$$First(\alpha) \cap First(\beta) = \emptyset$$

- Utilizing this LL(1) property, our parser can make a correct choice with a lookahead of exactly one symbol
    - In other words, by simply looking ahead one symbol, we know immediately which choice α or β we need to choose. This is because the next symbol following α *or* β can only be in one First set, according to the LL(1) property
- However, things aren't so simple: ϵ-productions force us to make additional checks in our LL(1) definition…

If $A \rightarrow \alpha$ & $A \rightarrow \beta$ & $\epsilon \in First(\alpha)$, then we need to ensure that $First(\beta)$ is disjoint from Follow(A), too, where **Follow(A)** is the set of terminal symbols that can immediately follow A in a sentinel form

- For convenience, let's define **First$^+$(**$A \rightarrow \alpha$)…
    - $First(\alpha) \cup Follow(A),\ if\ \epsilon \in First(\alpha)$
    - $First(\alpha),\ otherwise$
- Now, we have our new, better, more general definition for LL(1)...

A grammar is LL(1) if, and only if, $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$First^+(A \rightarrow \alpha) \cap First^+(A \rightarrow \beta) = \emptyset$$

- By having a grammar that has this LL(1) property, we can write a simple routine to find the next symbol…

```
/* find an A */
if (current_word ∈ FIRST+(A→β1))
   find a β1 and return true
else if (current_word ∈ FIRST+(A→β2))
   find a β2 and return true
else if (current_word ∈ FIRST+A→β3))
   find a β3 and return true
else
   report an error and return false
```

Grammars with the LL(1) property are called predictive grammars because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the LL(1) property are called predictive parsers.

14

First & Follow Sets

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentential form

FOLLOW(S) = {EOF}, where S is the start symbol

To build FOLLOW sets, we need FIRST sets ...

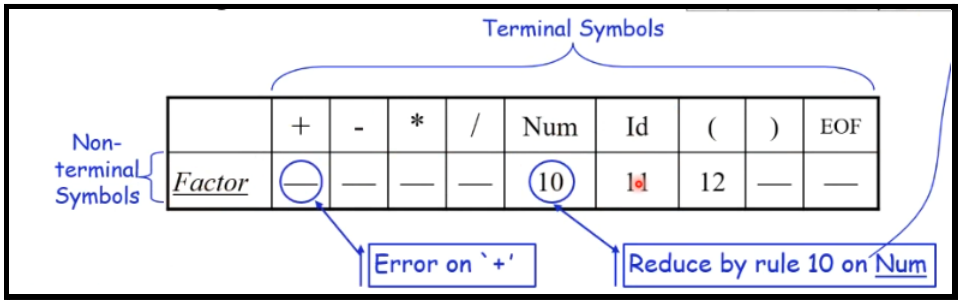
- Computing First Sets

```
for each x ∈ T, FIRST(x) ← {x}
for each A ∈ NT, FIRST(A) ← Ø
while (FIRST sets are still changing) do
   for each p ∈ P, of the form A→β do
      if β is B1B2...Bk then begin;
         rhs ← FIRST(B1) - {ε}
         // i = 1; i<= k-1 and ε ∈ FIRST(Bi) ; i++
         for i ← 1 to k-1 by 1 while ε ∈ FIRST(Bi) do
             rhs ← rhs ∪ (FIRST(Bi+1) - {ε})
             end  // for loop
       end        // if-then
       if  i = k and ε ∈ FIRST(Bk)
          then rhs ← rhs ∪ {ε}
        FIRST(A) ← FIRST(A) ∪ rhs
       end  // for loop
   end      // while loop
```

Computing FOLLOW Sets

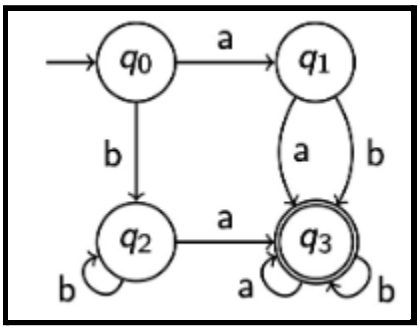
```
for each A ∈ NT, FOLLOW(A) ← Ø
FOLLOW(S) ← {EOF}
while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A→B1B2 ... Bk
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then                  // domain check
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ε ∈ FIRST(Bi)             // add right context
                  then TRAILER ← TRAILER ∪ (FIRST(Bi) - {ε})
                  else TRAILER ← FIRST(Bi)   // no ε => no right context
            else TRAILER ← {Bi}              // Bi ∈ T => only 1 symbol
```

- Once you have the FIRST and FOLLOW sets, only then can you come up with the FIRST$^+$ sets
- With the FIRST$^+$ sets, we can create a **parsing table**, which is simply a table that lays out the information displayed from the FIRST$^+$ sets…

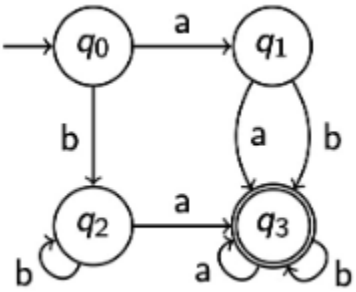


Lecture 23: Finite State Automata

- The front-end is about two components: the scanner and the parser. Today, we will learn how to automatically build a scanner using finite state automata
- A **finite state automaton** (**finite state machine**) is a directed graph where each node represents a *state*, and each edge denotes a *state transition*

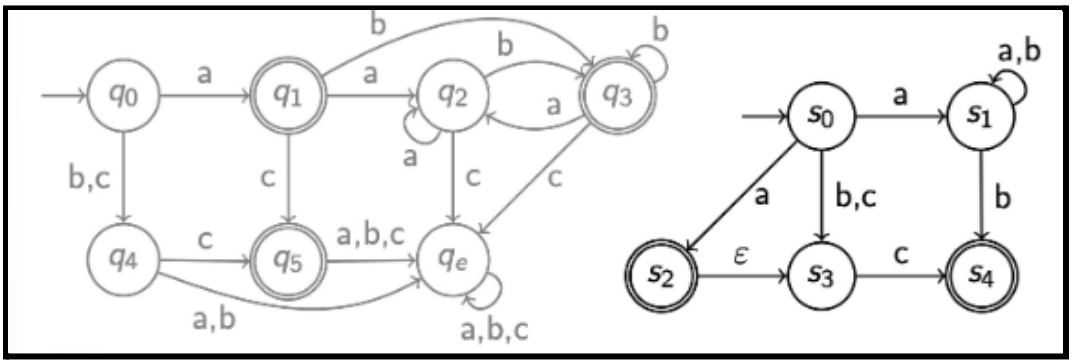


- Some states are special, such as q_0, the starting state, and Q_f, the final state
- We can test certain inputs to see if it belongs to the language of the finite state machine



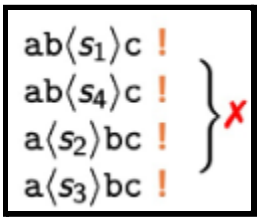
bbaba⟨q_3⟩ ✓
b⟨q_2⟩bb

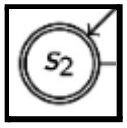
- There are two types…
    - *DFA* - *at most* one outgoing transition per each conditional $a \in \sum$(left)
    - *NFA* - *zero or more* transitions per conditional (right)



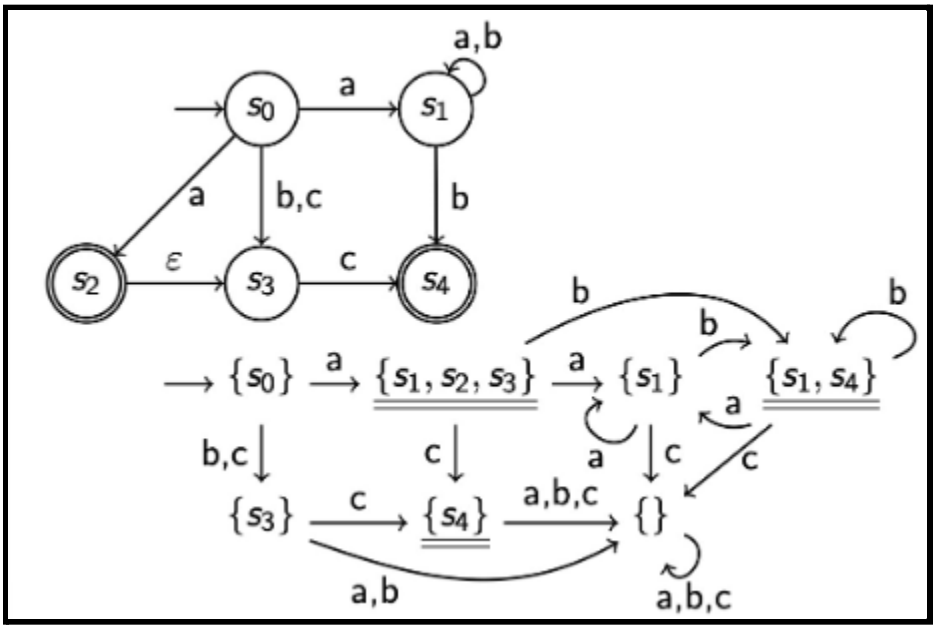
    - Acceptance: exits an accepting run
    - Rejection: all possible runs reject

- Remember that we have to check *all possible* runs because an NFA allows for the same conditional to be used more than once



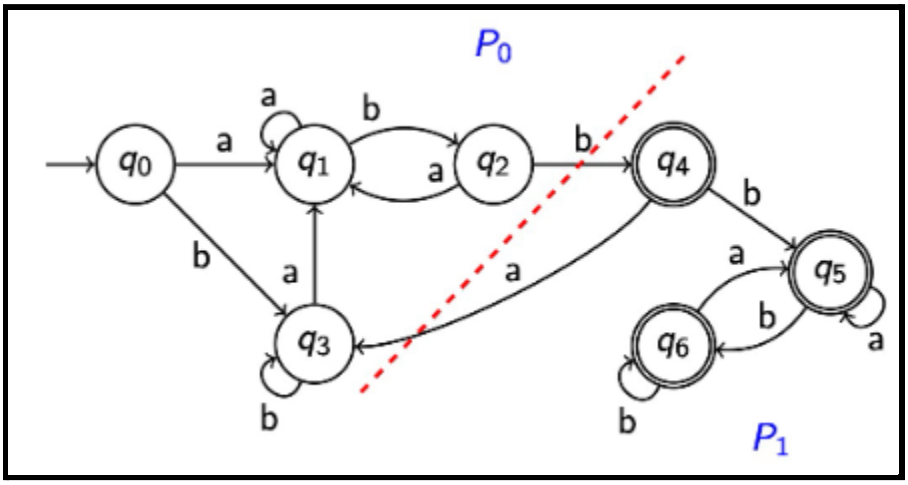
- A double ring in an NFA represents an accepting state → 

- The explanation is boring, but essentially the main takeaway is this: ***regular expressions, DFAs, and NFAs are all interchangeable***
    - In other words, there cannot exist a language that you can express with a DFA but not with an NFA and vice versa. Similarly, there cannot exist a language that you can express with an RE but not with an NFA/DFA and vice versa
- Since they are all interchangeable, if we can find a way to simply build one of them (NFA or DFA), we can then build the other one
- To do this, we will use something called **subset construction**, which is used to go from an NFA to a DFA
- For *subset construction*, we will build a DFA to simulate all NFA runs simultaneously…

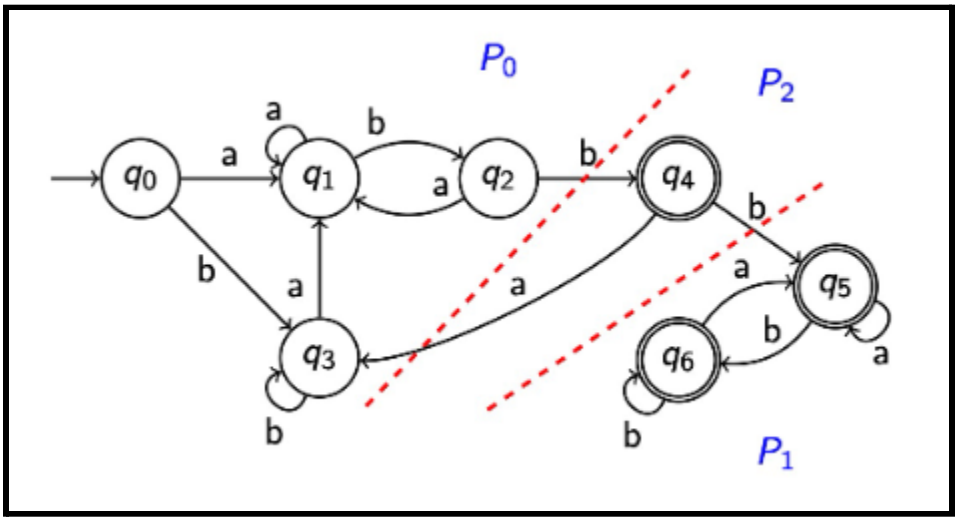


    - For each conditional edge outgoing from a node, have it point to a set of states where each state in the set has a state transition to it. As you can see, from s_0, we can travel along the state transition *a* to get to s_1, s_2, & s_3 (s_3 is reachable because of the empty string)

- As we can see above, the DFA we found is pretty large and complicated. Luckily, there exists a way to reduce the above DFA to a *minimal DFA*, where the number of possible states is minimal. Note that this minimal DFA will be **unique**
- ASIDE: Why have a DFA in the first place? Because in a DFA, we only need to do one pass in the graph in order to determine if the string is acceptable. In an NFA, while it visually is more concise, we have to check *all possible* paths, which is incredibly expensive.
- The algorithm to do this called **Hopcroft's algorithm**, and it works by gradually partitioning our graph so that states in different partitions do not behave the same, but states in the same partition do
    - Step 1: Split into a partition of the non-accepting sets and the accepting sets



    - Step 2: Ensure the partitions are consistent with the conditionals on the state transitions/edges (i.e. if you start at some node, and you have *a*, do they all behave the same?)



        - Look at P_1. If I start at q_6, and I have *a*, then we go to q_5. If I start at $q_5$and I have *a*, we also end up at q_5. Because of this, we say they *behave the*

same and we put them into the same partition. Since q_4 doesn't have the same, we split the partition along the state transition from q_4 to q_5

-

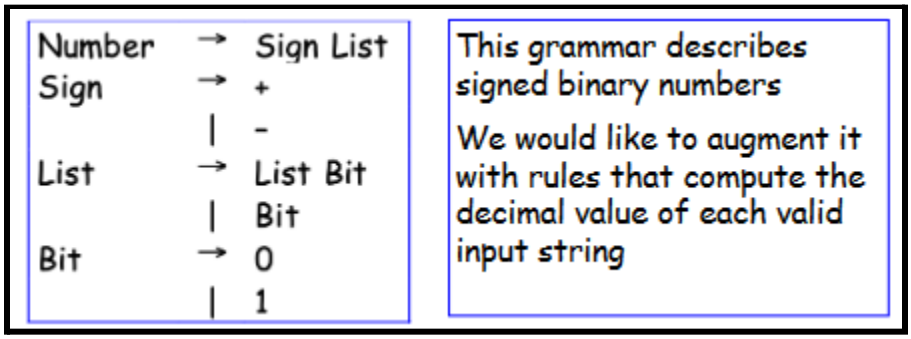
Lecture 24: Attribute Grammars, Type Checking

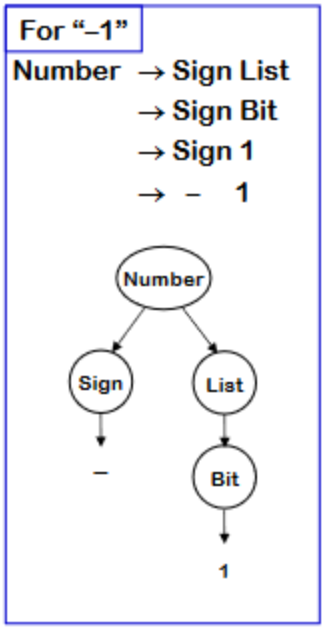
Semantic Analysis

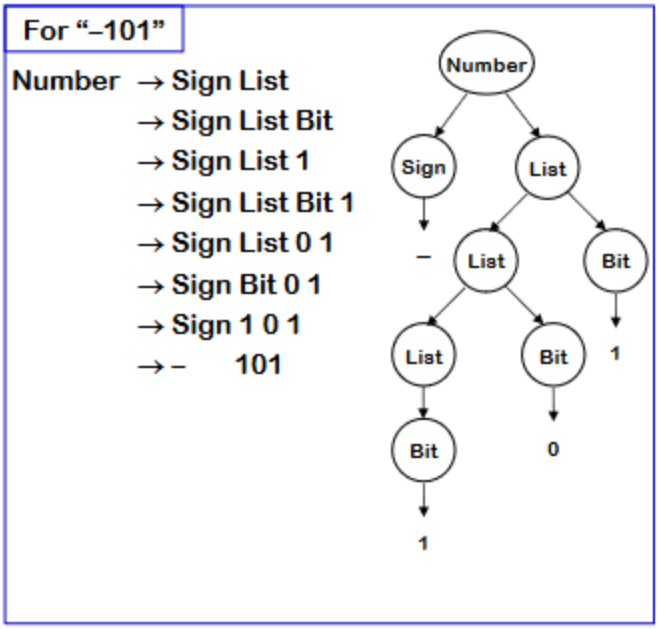
- **Semantic analysis** is the third step of the front end and ensures that the code makes sense
- For instance, it might be acceptable to say "x = ," in your code, however what happens if *x* is not yet declared?
- There are two different ways we can tackle these types of problems..
    - *Formal Methods*
        - Context-sensitive grammars
        - Attribute grammars
    - *Ad-hoc Techniques*
        - Symbol tables
        - Ad-hoc code

Attribute Grammars

- **Attribute grammar** extends context-free grammar by augmenting it with a set of rules
- Each symbol in the derivation/parse tree has a set of named values/attributes
- The rules specify how to compute a value for each attribute





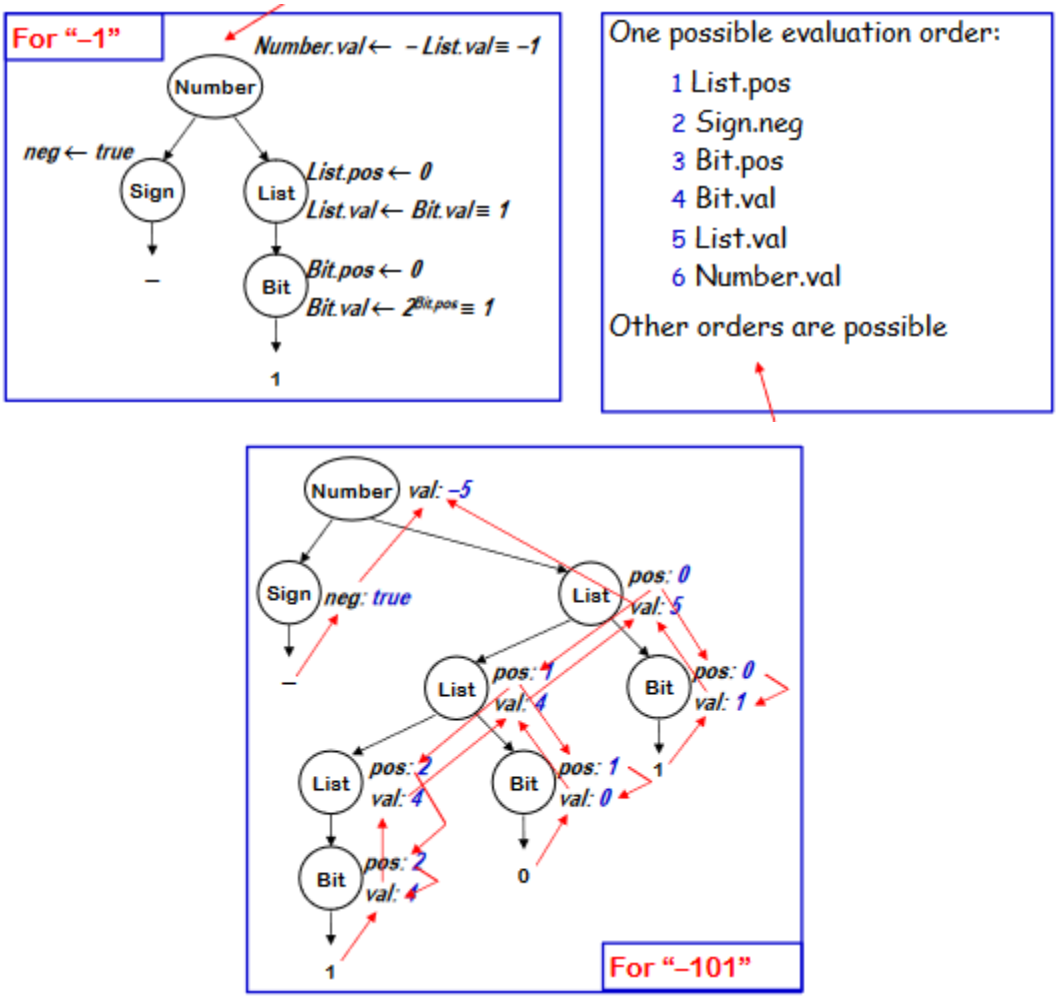


- Using the context-free grammar described for this, we can convert it to an *attribute grammar* by adding attributes and rules…

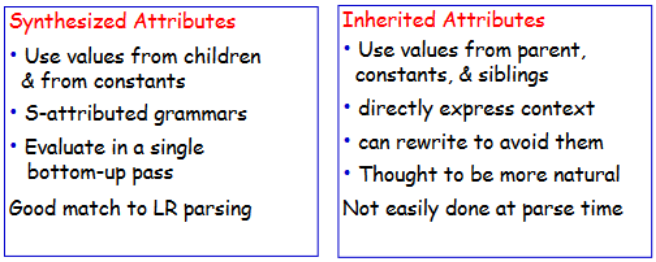
Productions			Attribution Rules
Number	→	*Sign List*	List.pos ← 0 If Sign.neg then Number.val ← – List.val else Number.val ← List.val
Sign	→	+	*Sign.neg ← false*
	|	–	*Sign.neg ← true*
$List_0$	→	$List_1$ *Bit*	$List_1$*.pos ←* $List_0$*.pos + 1* *Bit.pos ←* $List_0$*.pos* $List_0$*.val ←* $List_1$*.val + Bit.val*
	|	*Bit*	*Bit.pos ← List.pos* *List.val ← Bit.val*
Bit	→	0	*Bit.val ← 0*
	|	1	*Bit.val* ← $2^{Bit.pos}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

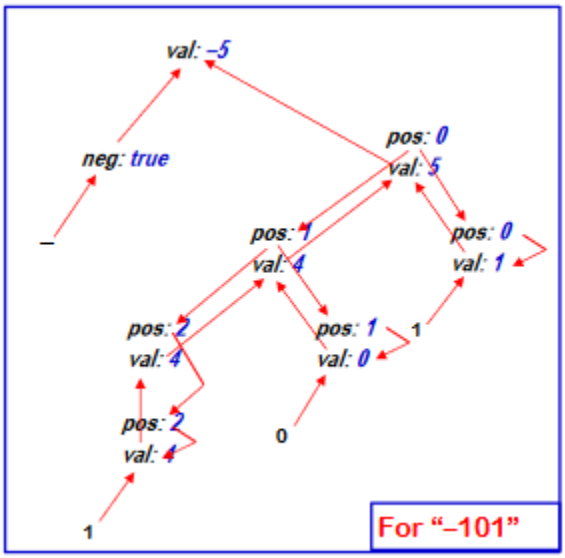
  - This attribute grammar computes the decimal value of a signed binary number
- The rules from the attribute grammar and the parse tree from the context-free grammar can be combined to create an *attribute dependence graph*; note that he evaluation order must be consistent with the attribute dependence graph



- When the graph flows from top-to-bottom, this is where we can see the *inherited attributes*, once it reaches the bottom and flows back upward this is where we get *synthesized attributes*
    - In practice, this order doesn't necessarily matter; it depends on the application & implementation
    - This isn't necessarily a strict process; it's simply a concept that needs to be understood
- To sum up…
    - Attribute grammars are an extension of context-free grammars
    - Because of this, it's not often used in compilers



- Note that the attribute dependence graph cannot contain a cycle…



All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The dynamic methods sort this graph to find independent values, then work along graph edges.

The rule-based methods try to discover "good" orders by analyzing the rules.

The oblivious methods ignore the structure of this graph.

- Tradeoffs
  - Pros: compiler writer thinks declaratively, lets a single engine do all the computation; similar to parsers and parsers-generators
  - Cons: restricting enough that they can be a bit awkward for expressing some practical analysis
- Because in practice we generally don't use attribute grammars, we generally use *ad-hoc analysis*

Ad-Hoc Grammar

- Declare program state per AST node (analogous to node features)
-