

Program Analysis

{ Mario Barrenechea
{ mario.barrenechea@colorado.edu

What's the Point of Analyzing Programs?

- ⌘ **Benefits:** Program *correctness, optimization, verification, performance, profiling*, ...
- ⌘ **Costs:** Development time or testing time, depending on when analysis is done.
 - ⌘ Some analyzers are very expensive (GramaTech [1] has a static analyzer for C/C++ that costs almost \$6000 for a single license).
- ⌘ **Alternatives:** Brute force testing, testing, testing.
 - ⌘ But you never really know when you're done...
- ⌘ **Consequences (for not doing it):** Sometimes inexplicable and critical failures that lead to software crises [WP].
 - ⌘ NASA Mariner 1
 - ⌘ Mars Polar Lander
 - ⌘ F22 Raptor
 - ⌘ Radiation Therapy machine from the 1980's.
 - ⌘ Patriot Missile System.
 - ⌘ Software bugs costs the U.S \$59.5 billion annually, according to a 2002 NIST report [WP].

Testing vs. Program Analysis

- ⌘ These forms of *software verification* are hard to pull apart. Testing can be thought of as a program analysis technique (verification, validation), yet program analysis also has applications for performance, profiling, and even more formal methods for verifying program correctness (instead of robustness or fault-tolerance, for example).
- ⌘ **Testing:** Focused on the verification and validation of software programs, often by utilizing executable, non-formal methods such as:
 - ⌘ Black, gray, and white box testing
 - ⌘ Unit/integration/subsystem/regression/acceptance testing
 - ⌘ Mutation testing
 - ⌘ Other methods.
 - ⌘ Testing is the de facto standard for performing *quality assurance* for a software project.
- ⌘ **Program Analysis:** Focused on utilizing tools and techniques (not so much methodology) on the rigorous and sometimes formal examination of program source code:
 - ⌘ Data flow analysis
 - ⌘ Dependency Analysis
 - ⌘ Symbolic Execution
- ⌘ Can you pull them apart in a different way?
 - ⌘ Definitely. Testing is considered a form of *dynamic* verification, while program analysis is more often a form of *static* verification. Think about what it means to perform static examinations of a program.

Three Kinds of Analyses

- ⌘ Generally speaking, there are three ways in which program analysis can be performed to analyze program source code:
 - ⌘ **Static:** Set of techniques to analyze source code without actually executing the program:
 - ⌘ Data-flow Analysis (DFA)
 - ⌘ Symbolic Execution
 - ⌘ Dependence Analysis
 - ⌘ **Dynamic:** Set of techniques to rigorously examine a program based on some criteria during run-time:
 - ⌘ Code Coverage Analysis
 - ⌘ Error-seeding and mutation testing, regression testing, other testing
 - ⌘ Program slicing
 - ⌘ Assertions
 - ⌘ **Human:** Often goes without saying, but human analyses include:
 - ⌘ Program comprehension
 - ⌘ Code reviews and walkthroughs
 - ⌘ Code inspections

A brisk walk through these analyses

- ⌘ We will visit some static, dynamic, and human analysis techniques.
 - ⌘ But it won't get too complicated; the idea is only to get an idea of how these analysis techniques can help aid the developer in producing quality software.
 - ⌘ And there will be pointers to some tools out there that exemplify how these techniques can be useful!

Static Analysis

- ⌘ *Static analysis* is a rigorous examination of program source code during compile-time (before run-time). The programmer must specify from the array of static analysis tools to fulfill the job of helping to satisfy some *criteria*, or the set of concerns shared by the programmer:
 - ⌘ Memory leaks
 - ⌘ Dangling pointers
 - ⌘ Uninitialized variables
 - ⌘ Buffer overflow
 - ⌘ Concurrency Issues {deadlock, race conditions}
 - ⌘ Performance bottlenecks
- ⌘ You can think of a set of criterion (or criteria) [3] as some predicate $C(T, S)$, where T is the set of test inputs on an executable component S , for which T satisfies some selection criterion over executing S . The expression $S(T)$ shows the results of executing S on T .
 - ⌘ An example of a criteria is something like, for these inputs (T) and this system (S), $C(T, S) = \text{“Does this input instance create a } \textit{memory leak?} \text{”}$

More on Static Analysis

- ⌘ We also need a way to *compare* compile-time criteria:
 - ⌘ Not all criteria can be satisfied with a single static technique.
 - ⌘ Ideally, we would like $C(T, S)$ such that for any S and every $T \subseteq D(S)$, where D is the domain of execution of S :
 - ⌘ if $S(T)$ is correct, then S is correct.
 - ⌘ Again, ideal, not realistic.
 - ⌘ But we can use *subsumption* to analyze and evaluate these criteria w.r.t the techniques used:
 - ⌘ Ex: Branch Coverage (S,T) \Rightarrow Statement Coverage (S,T)
 - ⌘ That is to say, branch coverage “subsumes” statement coverage; every program S run successfully on branch coverage will also run successfully on statement coverage.
- ⌘ Note that static analysis can not possibly examine *everything*.
 - ⌘ Since the analyzer is not given the program executable, it cannot infer any optimizations that the compiler will make on the program.
 - ⌘ The implication of this is that a static analyzer can trace through lines of code and make evaluations based on the logic represented by those statements. However, it cannot make evaluations based on the execution of those statements.
 - ⌘ The best thing to do? Do both static and dynamic analyses on your program.

Static Analysis: *Data-flow Analysis (DFA)*

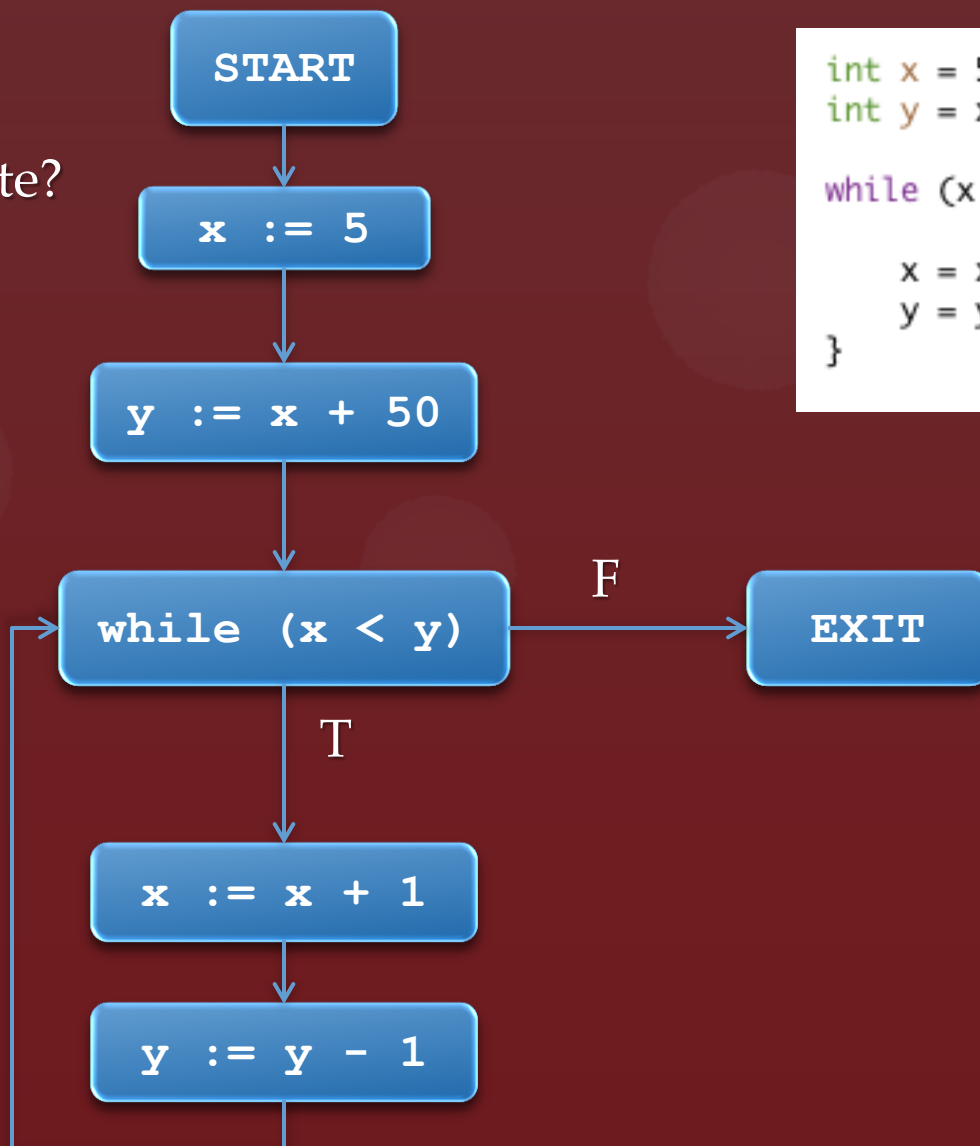
- ⌘ Data flow analysis is a technique to monitor how variables and their values change through the program flow. This is awfully generic, so there are *sub-techniques* that belong within the DFA framework that specialize in this form of analysis.
- ⌘ DFA can be broken down into two approaches: **forward analysis** and **backward analysis**. Essentially, to compute several properties of program statements, some sub-techniques require a backward approach to DFA while others require a forward one.
 - ⌘ **Reaching Definitions:** *Given a variable x and its assignment, where does it “reach” to without intervening assignments? At what point is the current value of x irrelevant?*
 - ⌘ **Live Variable Analysis:** *Given a variable x and its assignment, how long does it retain a specific value before being re-assigned?*
 - ⌘ **Available Expressions:** *Given an expression $(x+y)$, where can the program re-use this expression such that it doesn't have to be re-computed?*

DFA: CFGs!

- ⌘ Before diving in to these sub-techniques, we need a way in which we can model the program flow. Robert Floyd devised a flowchart language [4] that allows for propositional interpretation of programs. Today, we call his construct a *flowchart*, or more formally, a *control-flow-graph* (CFG).
- ⌘ A CFG is a graph $G(V, E, S, T)$ with V vertices, E edges, where $u, v \in V$ and an edge connecting u and v is represented as $(u, v) \in E$, S as the starting vertex, and T as the terminating (exit) vertex. Since programs naturally have looping structures, we consider CFGs as directed, cyclic graphs.
- ⌘ Note that there is more to the eye than just graphically representing a program using vertices and edges. Floyd was arguing about a novel construct that could help reason about program correctness using propositions that are generated after each vertex.
 - ⌘ So if a particular program statement assigned the value 5 to a variable x , then the proposition, “ $x = 5$ ” is generated in conjunction with all other propositions that came before that statement. We don’t worry about this so-called, “propositional propagation” here.

An example of a CFG

Does this code terminate?



```
int x = 5;  
int y = x + 50;  
while (x < y){  
    x = x + 1;  
    y = y - 1;  
}
```

DFA: Available Expressions

⌘ The sub-technique called *available expressions* allocates re-usable expressions that recur within the code and propagates them throughout the program.

⌘ Consider the following code:

```
int x = 5;
int y = x + 50;
double z = x + y + 5.0;

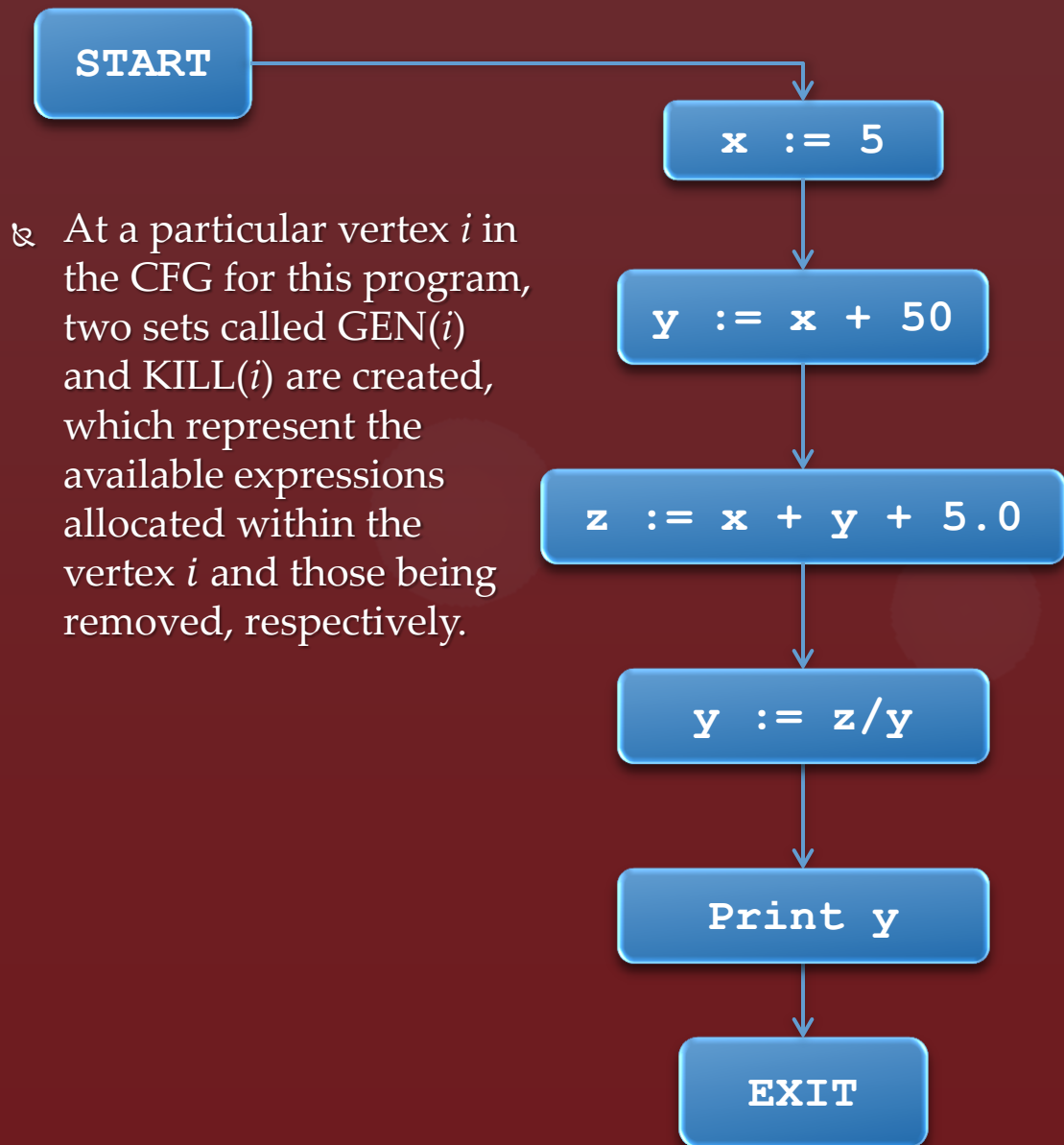
//Change to expression in y
y = (int) z/y;
System.out.println(y);
```

⌘ The value for x wouldn't be saved, since it's a simple primitive value. However, the expression for $y = x + 50$ and $z = x + y + 5.0$ would be saved and kept as available expressions.

⌘ However, the trick here is that when y or z are changed later in the program, its assigned expression cannot be re-used again since the values for those variables have changed.

⌘ $\{ (y = x + 50), (z = x + y + 5.0) \}$ are allocated by the analyzer, but once it evaluates $y = (int) z/y;$, we cannot rely on the expression $(z = x + y + 5.0)$ as an available expression, since the value for y has changed.

DFA: Available Expressions



- At a particular vertex i in the CFG for this program, two sets called $GEN(i)$ and $KILL(i)$ are created, which represent the available expressions allocated within the vertex i and those being removed, respectively.

```
int x = 5;
int y = x + 50;
double z = x + y + 5.0;

//Change to expression in y
y = (int) z/y;
System.out.println(y);
```

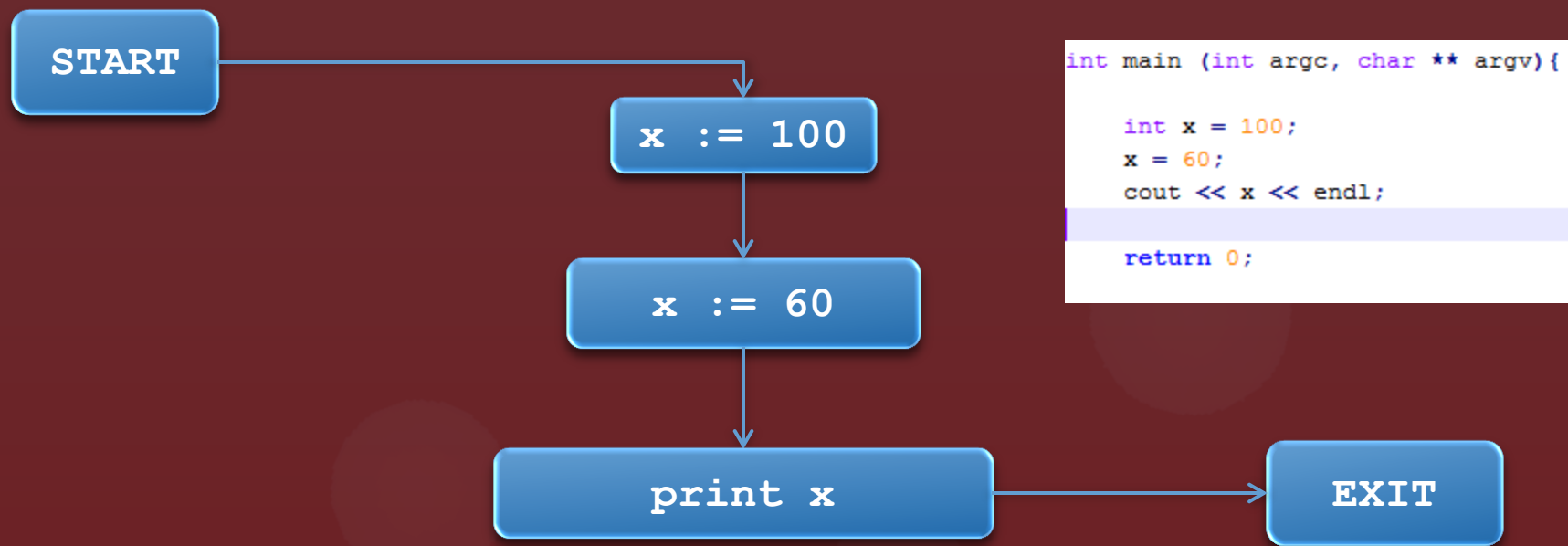
- Before each vertex is evaluated, the analyzer takes the set intersection of the GEN sets coming into the vertex and propagates that through, inserting new expressions into the GEN set and removing others from the $KILL$ set. The result of this sub-technique is several available expressions that can be *saved* and used throughout the program.

DFA: Reaching Definitions

- ⌘ Recall: Given an assignment x , where does x “reach out” to? The sub-technique *Reaching Definitions* is yet another forward analysis.
- ⌘ We need to define *def* (x) and *use* (x) pairs for a particular variable x .
 - ⌘ *def* (x) = The point in the program where x is defined or re-defined.
 - ⌘ *use* (x) = The point in the program where x is being used or referred.
 - ⌘ We call this linkage a *def-use chain* or *pair*.
- ⌘ Why do we care about this?
 - ⌘ Reaching Definitions can help us spot “dead code”, or code that contains a def or ref of some variable x that will not be used in the program.
 - ⌘ Ex: Where is the dead code here?

```
int main (int argc, char ** argv){  
  
    int x = 100;  
    x = 60;  
    cout << x << endl;  
  
    return 0;  
}
```

DFA: Reaching Definitions



⌘ We perform the same forward propagation here: At the vertex where $[x := 100]$, we add that assignment to the GEN set for this vertex. Then, for the next vertex at $[x := 60]$, we make the set assignment:

$$\varnothing \quad \text{GEN}_{[x:=60]} = \text{GEN}_{[x:=100]} - \text{KILL}_{[x:=100]}$$

⌘ When we find that the current assignment overlaps with an assignment in the GEN set, we add that GEN set member to the KILL set so that we get rid of it – dead code.

DFA: Live Variable Analysis (LVA)

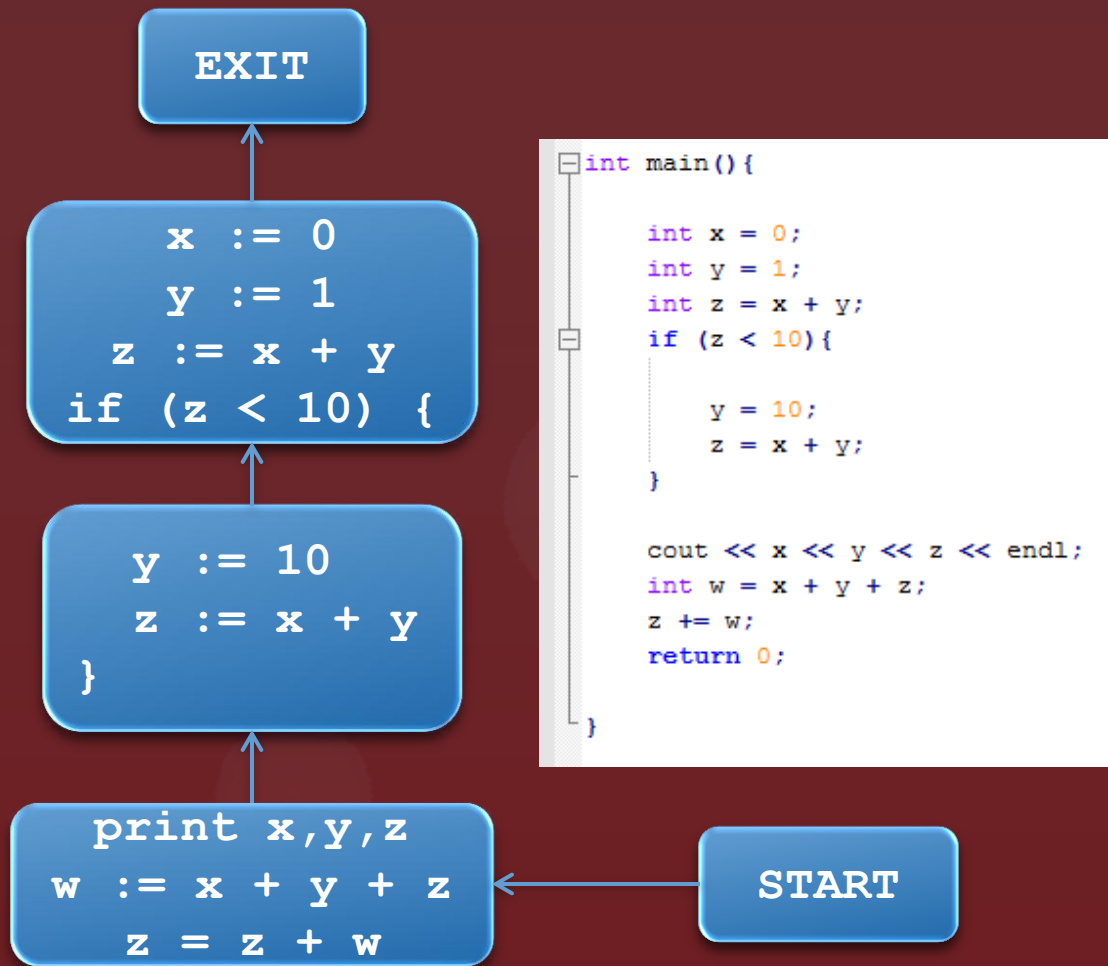
- ⌘ LVA: “What variables might get used later?”
- ⌘ This time, we’ll use a backward analysis sub-technique known as LVA.
- ⌘ LVA allows us to compute the GEN and KILL sets of variables from the bottom going up. For each vertex in the CFG, the analyzer keeps track of used variables and eliminate those that have been defined. This backward propagation allows us to keep track of variables that are “live” at various points in the program.
- ⌘ With this code split into two blocks (separated by line break), the GEN set for the first vertex going up is {z} and KILL is {y} since it is being assigned. Consequently, the GEN set for the second code block must be {} since it is at the top of the program, and all of the variables have been shown to be defined at some point.

```
int x = 5;  
int y = x + 50;  
double z = x + y + 5.0;  
  
//Change to expression in y  
y = (int) z/y;  
System.out.println(y);
```

DFA: LVA

⌘ The last code block performs a set union from the GEN set from the first and the second vertices to represent true and false branches from the conditional. Finally, we see that all of the variables have been defined here, as our GEN set after the vertex has been evaluated is $\{x, y, z\}$. The KILL set, on the other hand, is $\{x, y, z\}$.

⌘ The next block is the body of the if conditional. The GEN set for this vertex after it has been evaluated contains $\{x\}$. The KILL set contains $\{z, y\}$.



⌘ We start down here, remember? The GEN set for this vertex after it has been evaluated contains $\{x, y, z\}$. The KILL set was $\{z, w\}$ because both were defined here.

Dynamic Analysis

- ⌘ While static analysis is done to make rigorous evaluations of the program source code for optimization, correctness, or performance purposes, *dynamic analysis* is well suited for making evaluations based on program runtime, or execution.
- ⌘ Typically, a dynamic analyzer needs an output specification to compare the actual output to. Generally speaking, an *oracle* is the specification against which actual outputs compare their results.
 - ⌘ Can be another system, model, person, customer, etc.
 - ⌘ In the case of black box testing, the output specification *is* the oracle.
- ⌘ Here, we see criteria play a role in which some techniques (or even sub-techniques) are better than others.
 - ⌘ Coverage analysis is a great example of this: The most costly (and yet most powerful) form of code coverage is all-paths, where $All\ Paths(T, S) \Rightarrow Branch(T, S) \Rightarrow Statement(T, S)$

Dynamic Analysis: Assertions

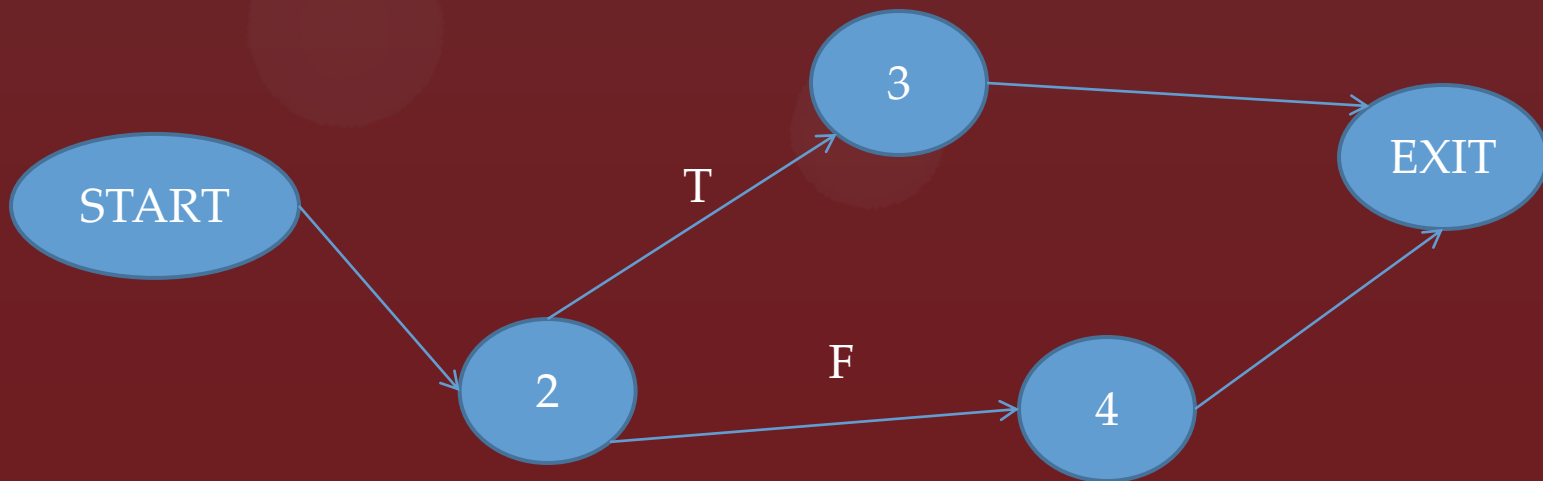
- ‡ *Assertions* are fantastic ways to perform dynamic analysis in a program. It's easy, almost all languages have assertions built into them, and they help prevent dubious or malicious inputs from creating unintended behavior.
- ‡ Assertions are simple checks to make as actual program statements within the program source code. When the program is run, all assertions must be checked and satisfied before program execution can proceed. If at a point in the program the assertion is not satisfied, the program halts, and an error message is produced.

‡ Ex:

```
int main(int argc, char * [] argv){  
  
    assert (argc > 2);  
    return 0;  
}
```


Dynamic Analysis: Coverage Analysis

As learned in class, code coverage, or *coverage analysis*, is a way to dynamically evaluate the pathways through program execution for given inputs. The idea here is that programmer strives for 100% coverage, through true/false branches of while/for/if/case structures, as well as all program statements.



Dynamic Analysis: Testing

- ⌘ Don't forget: Testing is yet another form of dynamic analysis! Testing is a method of quality assurance that allows developers to estimate with a certain percentage of confidence that the program will produce correct output:
 - ⌘ Black/Gray/White box testing: Testing with certain transparency of the system, which directs how we test the system itself.
 - ⌘ Unit testing: Testing several methods, functions, classes
 - ⌘ Integration Testing: Testing the connection between modules and their intra-functionality.
 - ⌘ Subsystem Testing: Testing subsystems of modules together with their intended functionality.
 - ⌘ Acceptance Testing: Testing to see if the program meets the requirements of the customer. Here, the oracle is the customer him/herself!

Human Analysis

- ⌘ Self-explanatory, but it's the most natural (and presumably most labor-intensive) way of verifying that the code produces correct results. In human analysis, programmers, designers, managers, and testers perform qualitatively and quantitatively controlled review processes to examine code without use of a computer.
- ⌘ Code reviews, inspections, and human-factors methodologies (ex. SCRUM) focus in on how to designate teams for improved developer productivity, communication, and feedback.
- ⌘ Defect Detection methods, developer group size, and single-interval vs. multiple-interval sessions are factors discussed in [6] that show significant tradeoffs made when humans seek to inspect code bugs without aid of systematic methods.



Human Analysis: Inspections

- ⌘ In human analysis, the *inspection* allows a team of program role members (programmers, testers, managers) to find errors in the program code in a formal, efficient, and economical fashion [10].
 - ⌘ Usually in teams of 4-5, so there is no strict number.
- ⌘ The inspection team for a particular module comprises the tester, designer, developer, and the *moderator*:
 - ⌘ Moderator: The key to any successful inspection, the moderator uses his/her leadership to conduct meetings, delegate tasks, and follow-up on rework.
- ⌘ After the individual preparation of the module being inspected, the inspection team gathers and enumerates through the logic of the module.
 - ⌘ The designer first explains the design, and with the general understanding of that design, the team members walk carefully through the module line by line (and branch) in order to find errors.

Human Analysis: Inspections

⌘ An example [10] of a code inspection report that the inspection team uses to monitor error-tracking:

**CODE INSPECTION REPORT
SUMMARY**

Date 11/20/-

To: Design Manager KRAUSS Development Manager GIOTTI

Subject: Inspection Report for CHECKER Inspection date 11/19/-

System/Application _____ Release _____ Build _____

Component _____ Subcomponents(s) _____

Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				Sub-component
					Pre-insp			Est Post			Rework			Prep	Insp Meetg	Re-work	Follow-up	
					A	M	D	A	M	D	A	M	D					
	N		McGINLEY	HALE	348			400			50			9.0	8.8	8.0	1.5	
Totals																		

Reinspection required? YES Length of inspection (clock hours and tenths) 2.2

Reinspection by (date) 11/25/- Additional modules/macros? NO

DCR #'s written C-2

Problem summary: Major 13 Minor 5 Total 18

Errors in changed code: Major _____ Minor _____ Errors in base code: Major _____ Minor _____

LARSON McGINLEY HALE

Initial Desr Detailed Dr Programmer Team Leader Other Moderator's Signature

⌘ Note that this is how bug-tracking was done in 1976!

Program Analysis Tools?

- ⌘ I'm quite bitter about them.
- ⌘ There are two worlds out there:
 - ⌘ One world with program analysis tools that come integrated into development environments that enable the developer to make more rigorous examinations of her code.
 - ⌘ The other world chalk full of exciting and (mostly) free program analysis tools that are exceedingly difficult to configure or install properly, comprehend, and use within your development environment.
 - ⌘ I'm a fan of the first world and not the second.

Case Study: Eclipse

- ⌘ The Eclipse Integrated Development Environment (IDE) [9] is an excellent way to take advantage of a plethora of static analysis techniques at your fingertips.
- ⌘ When you create a new Java project and start writing code, you will notice red and yellow squiggly lines underneath some of your code.
 - ⌘ The yellow squiggly line indicates a future compiler warning; that is, the eclipse environment has evaluated your code and foretold you that some program statement is causing an irregular behavior, perhaps an unused variable, un-genericized type, or some other reason.
 - ⌘ The red squiggly line, on the other hand, indicates a future compiler error that will cause unpredictable behavior if the program is run. This is most typically due to syntax errors, but it can also be because of referring to a type not recognized, a variable not defined yet, etc.
 - ⌘ What kinds of static analyses can detect these problems?

Case Study: Eclipse

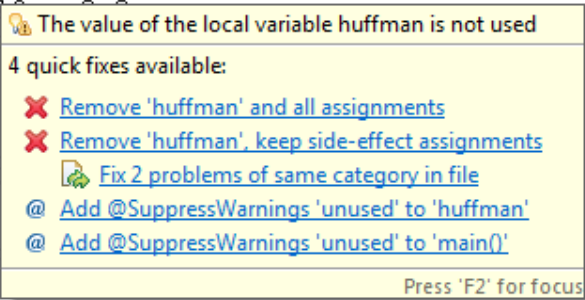
☞ Example of a future compiler warning:

```
/**
 * @param args
 */
public static void main(String[] args) throws IOException {

    //Declarations
    HuffmanEncoder huffman = null;
    double averageA0 = 0.0;
    int totalAtomicOperation = 0;

    // Setup File for writing
    File f = new File ("./road_not_taken.txt");
    Scanner inFile = new Scanner(f);
    String roadNotTakenString = "";

    while (inFile.hasNextLine()) {
        String line = inFile.nextLine();
        roadNotTake
```



The value of the local variable huffman is not used

4 quick fixes available:

- Remove 'huffman' and all assignments
- Remove 'huffman', keep side-effect assignments
- Fix 2 problems of same category in file
- Add @SuppressWarnings 'unused' to 'huffman'
- Add @SuppressWarnings 'unused' to 'main()'

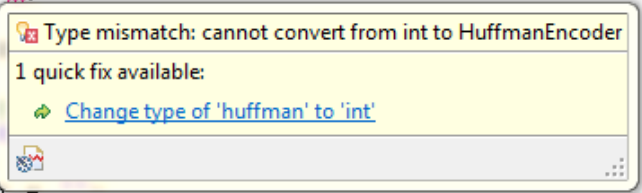
Press 'F2' for focus

☞ Example of a future compiler error:

```
/**
 * @param args
 */
public static void main(String[] args) throws IOException {

    //Declarations
    HuffmanEncoder huffman = 1;
    double averageA0 = 0.0;
    int totalAtomicOperation = 0;

    // Setup File for writing
    File f = new File ("./road_not_taken.txt");
    Scanner inFile = new Scanner(f);
    String roadNotTakenString = "";
```



Type mismatch: cannot convert from int to HuffmanEncoder

1 quick fix available:

- Change type of 'huffman' to 'int'

Conclusions

- ⌘ Program Analysis is a deep sub-field straddled between software engineering and programming language research with lots of open problems.
 - ⌘ To determine if a program will halt (or not) is undecidable. How does this affect our analysis?
 - ⌘ Static, dynamic, and human analyses help us be more informed of our program behavior and how we can improve it.
 - ⌘ As newer technologies (cloud computing for example), we must find ways of breaking down their complexity to address security, performance, and other thematic concerns. Program analysis provides a means of breaking down (analyzing) that complexity.
 - ⌘ Visit the Tools page (next slide) to learn more about the tools referenced throughout this presentation.

List of Tools

☞ List of tools:

- ☞ Coverity [2] : www.coverity.com
 - ☞ C/C++ static analyzer for commercial use
- ☞ Grammatech Codesurfer/Codesonar [1]: www.grammatech.com
 - ☞ C/C++ static analyzer for commercial use
- ☞ Eclipse development environment [9]: www.eclipse.org
 - ☞ Open source programming environment for a host of supported languages and technologies. Very popular and comes with integrated static analyzer for select languages.
- ☞ Soot [5]: <http://www.sable.mcgill.ca/soot/>
 - ☞ Java Optimization framework (static analysis), although not straightforward to set up.
- ☞ Xcode [7]: <https://developer.apple.com/technologies/tools/>
 - ☞ Apple's very own integrated development environment that comes with a built-in static analyzer and performance monitoring tools.
- ☞ Avalanche [8]: <http://code.google.com/p/avalanche/>
 - ☞ An open-source dynamic analyzer. Again, very hard to configure and use.

References

- ⌘ [1]: Static Analysis for C and C++ | GrammaTech. (n.d.). Retrieved March 21, 2012, from <http://www.grammatech.com/>
- ⌘ [2]: Coverity Static Analysis Tools for C/C++, C#, and Java | Coverity. (n.d.). Retrieved March 21, 2012, from <http://coverity.com/products/static-analysis.html>
- ⌘ [3]: Clarke, L. A. (2010). Introduction to Dynamic Analysis. Presentation.
- ⌘ [4]: Floyd, R. (1967). Assigning meanings to programs. *Mathematical aspects of computer science*. Retrieved from <http://books.google.com/books?hl=en&lr=&id=ynigSICiflYC&oi=fnd&pg=PA19&dq=Assigning+Meanings+to+Programs&ots=i1HMoZeLAd&sig=9mSUp4i49ntWJW1G7rjm8KFBQh8>
- ⌘ [5]: Soot: a Java Optimization Framework. (n.d.). Retrieved March 22, 2012, from <http://www.sable.mcgill.ca/soot/>
- ⌘ [6]: Porter, A., Siy, H., & Toman, C. A. (1995). An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development, 92-103.
- ⌘ [7]: Apple Xcode Tools. <https://developer.apple.com/technologies/tools/>
- ⌘ [8]: Avalanche Dynamic Analysis. <http://code.google.com/p/avalanche/>
- ⌘ [9]: The Eclipse Foundation. <http://www.eclipse.org>
- ⌘ [10]: Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 258-287. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5388086
- ⌘ And [WP]: Wikipedia for general acquaintance with the body of knowledge.

Program Analysis

Mario Barrenechea

- ⌘ Software and its complexity has increasingly convinced researchers and engineers to utilize program analysis as a form of verification, both through formal and practical methods.
- ⌘ This presentation will walk through three types of analyses {static, dynamic, human} in order to give a well-rounded glimpse of how program analysis can help you in future development!

Static techniques:

Data-flow analysis
Symbolic Execution
Dependence Analysis

Dynamic techniques:

Testing
Assertions
Coverage Analysis

Human techniques:

Code Inspection
Program Comprehension
Code Reviews