Program Analysis Tools

Steven J Zeil

Last modified: Apr 07, 2015

Contents:

Abstract

In this lesson we look at a variety of code analysis tools available to the practicing software developer. These include *static* analysis tools that examine code without executing it, and *dynamic* analysis tools that monitor code while it is being run on tests or in operation.

We will look at the kinds of information that developers can obtain from these tools, the potential value offered by this information, and how such tools can be integrated into an automated build or a continuous integration setup.

Classifying Analysis Tools

- Static Analysis
  - style checkers
  - data flow analysis
- Dynamic Analysis
  - Memory use monitors
  - Profilers

Analysis Tools and Compilers

Analysis tools, particularly static, share a great deal with compilers

- Need to parse code & understand at least some language smeantics
- Data flow techniques originated in compiler optimization

1. Representing Programs

Most static analysis is based upon one of these graphs

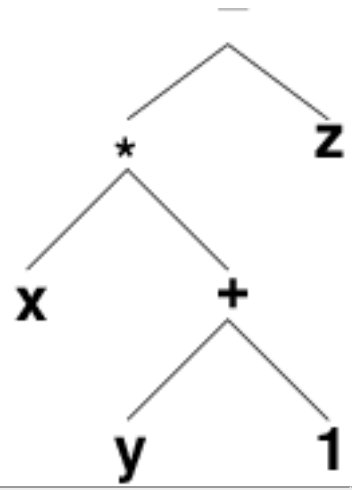
That's "graphs" in the discrete mathematics (CS 381) or data structures (CS 361) sense: a collection of nodes connected by edges, not

the sense of points plotted on X-Y axes.

- Abstract syntax trees
- Control Flow Graphs

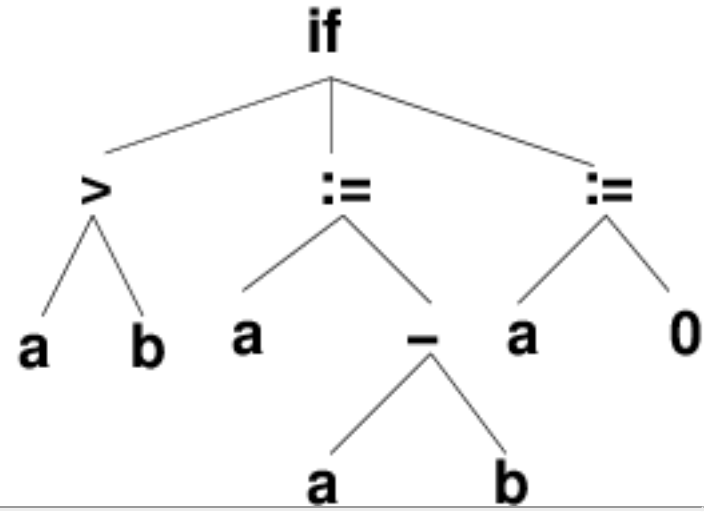
1.1 Abstract Syntax Trees (ASTs)

- Output of a language parser
  - Simpler than parse trees
- Generally viewed as a generalization of operator-applied-to-operands



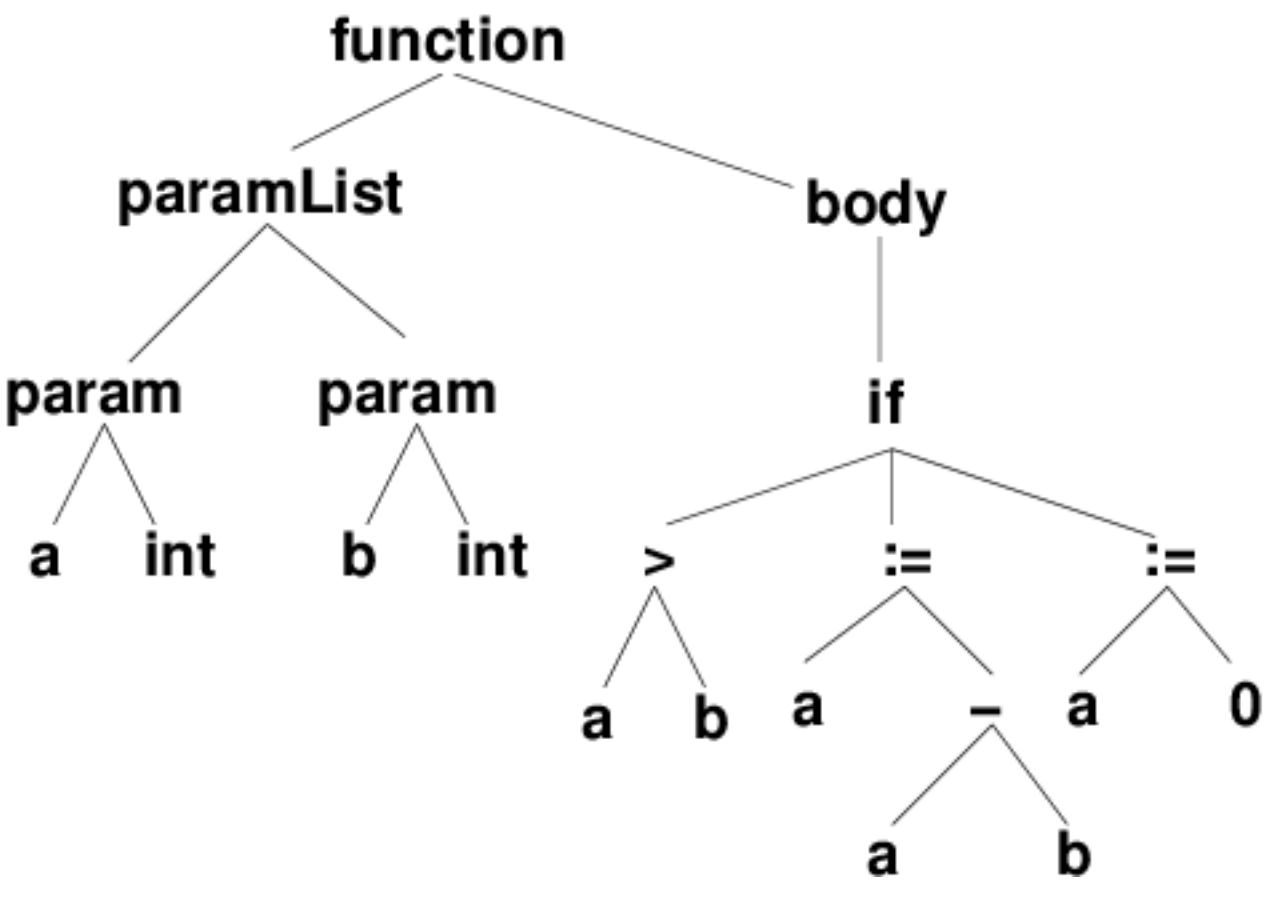
Abstract Syntax Trees (cont.)

- ASTs can be applied to larger constructions than just expressions



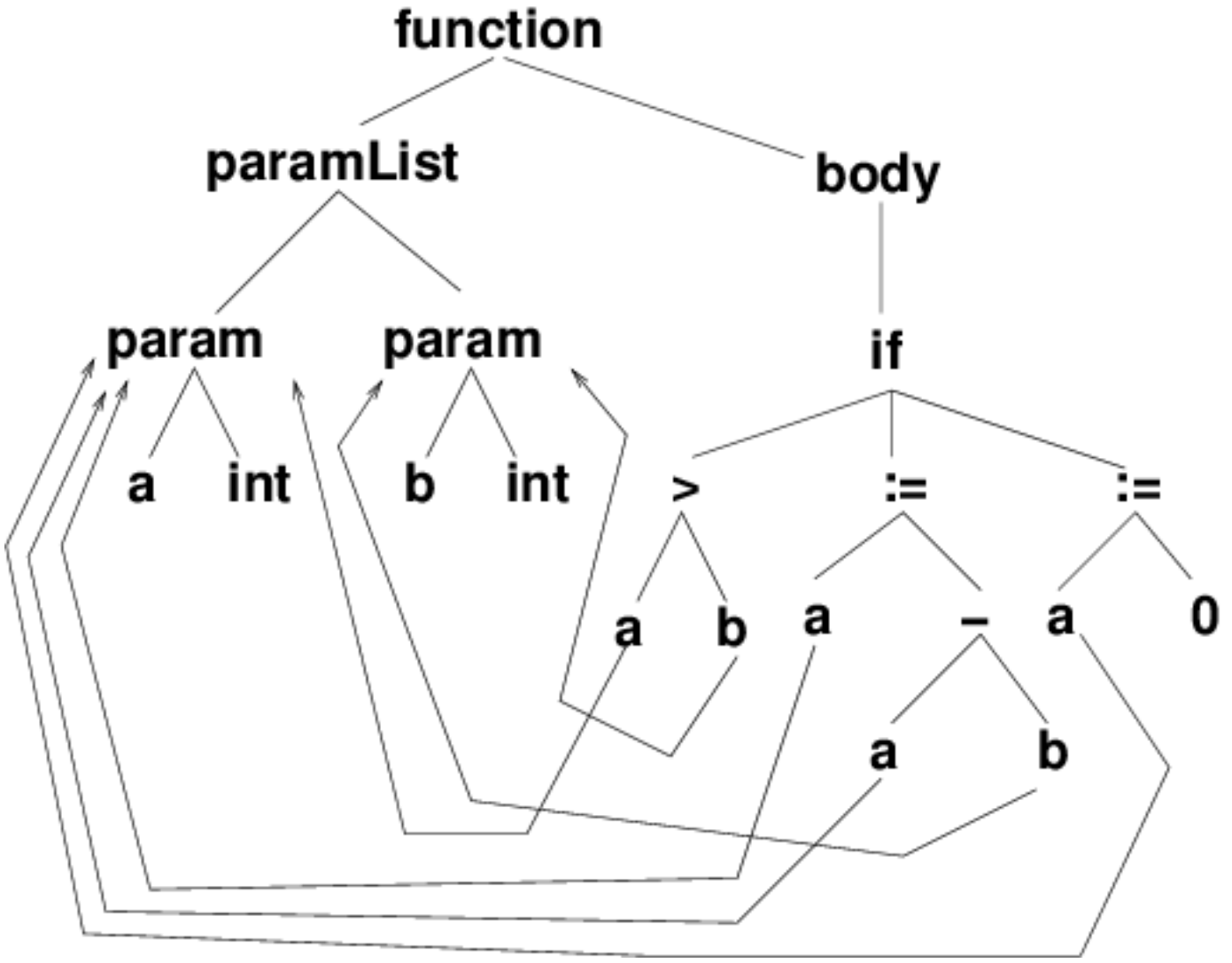
Abstract Syntax Trees (cont.)

- In fact, generally reduce entire program or compilation unit to one AST



Abstract Syntax Graphs

- In most programming languages, any given variable name (e.g., `i`, `x`) could actually refer to many different objects depending upon the *scope* rules of the language.
- The *semantic analysis* portion of a compiler pairs uses of variable names with the corresponding declarations
  - What we have left is no longer a tree, but a graph.



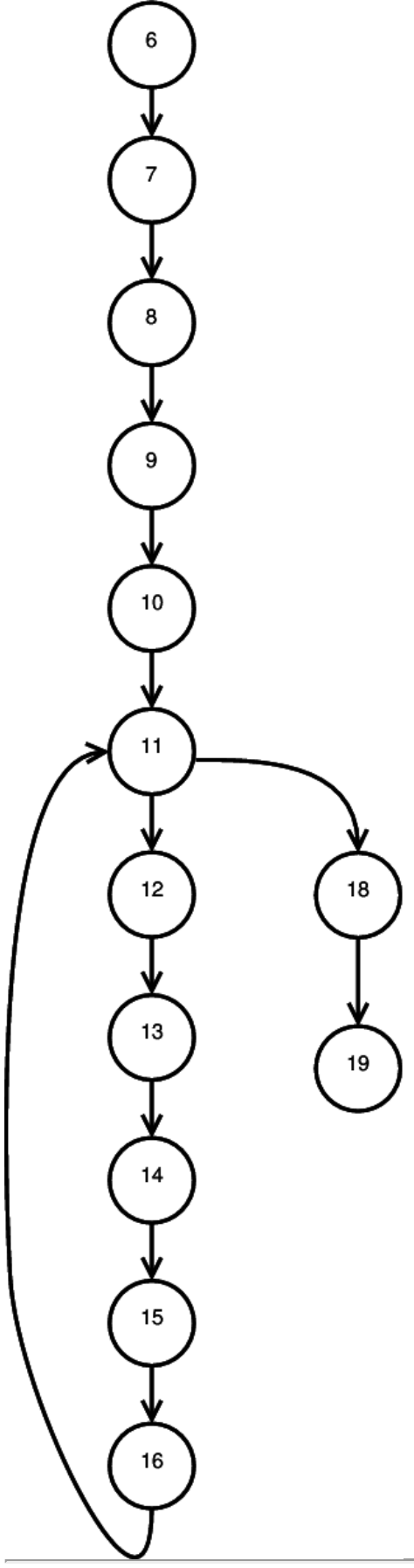
1.2 Control Flow Graphs

Represent each executable statement in the code as a node,

- with edges connecting nodes that can be executed one after another.
  - Nodes for conditional statements have two or more outgoing edges.

Sample CFG

```
01: procedure SQRT (Q, A, B: in float;
02:                 X: out float);
03: // Compute X = square root of Q,
04: //    given that A <= X <= B
05:    X1, F1, F2, H: float;
06: begin
07:    X1 := A;
08:    X2 := B;
09:    F1 := Q - X1**2
10:    H := X2 - X1;
11:    while (ABS(H) >= 0.001) loop
12:       F2 := Q - X2**2;
13:       H := - F2 * ((X2-X1)/(F2-F1));
14:       X1 := X2;
15:       X2 := X2 + H;
16:       F1 := F2
17:    end loop;
18:    X := (X1 + X2) / 2.;
19: end SQRT;
```



Simplifying CFGs: Basic Blocks

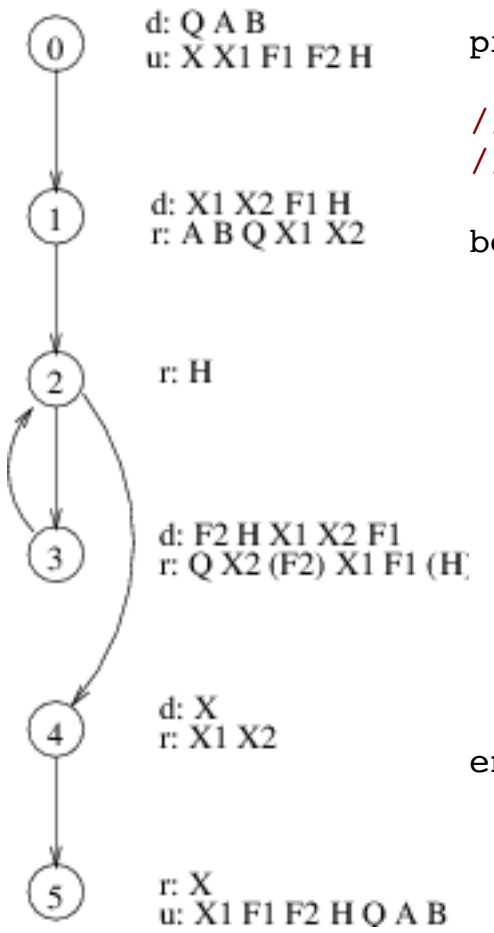
```
procedure SQRT (Q, A, B: in float; //  node 0
                X: out float);
// Compute X = square root of Q,
//    given that A <= X <= B
   X1, F1, F2, H: float;
begin
   X1 := A;
   X2 := B;                          // node 1
   F1 := Q - X1**2
   H := X2 - X1;
   while (ABS(H) >= 0.001) loop      // node 2
      F2 := Q - X2**2;
      H := - F2 * ((X2-X1)/(F2-F1));
      X1 := X2;                      // node 3
      X2 := X2 + H;
      F1 := F2
   end loop;
   X := (X1 + X2) / 2.;              // node 4
end SQRT;                            // node 5
```


Data Flow Analysis

- All data-flow information is obtained by propagating data flow markers through the program.
- The usual markers are
  - $d_i(x)$: a **def**inition of variable `x` (any location where `x` is assigned a value) at node `i`
  - `r<sub>i</sub>(x)`: a **ref**erence to `x` (any location where the value of `x` is used) at node `i`
  - `u<sub>i</sub>(x)`: an **undef**inition of `x` (any location where `x` becomes undefined/illegal) at node `i`
- Data flow problems are solved by propagating markers around a control flow graph

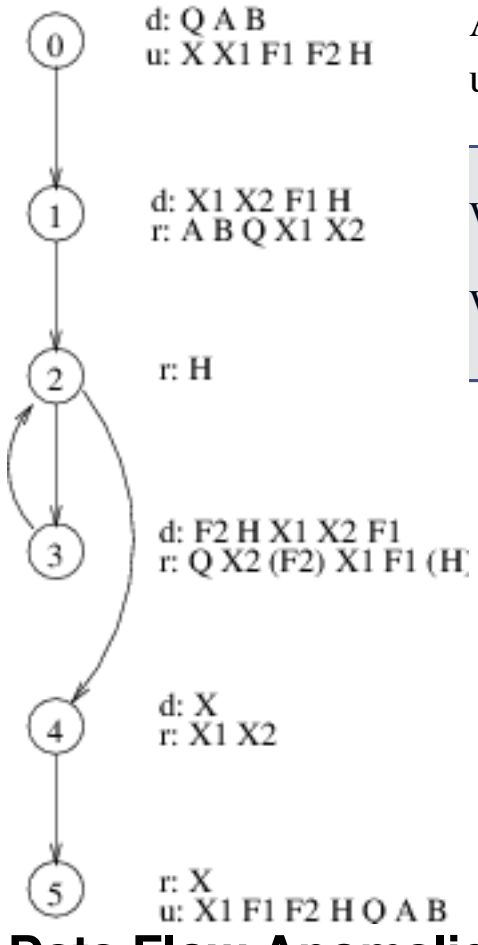
---

Data-Flow Annotated CFG




```
procedure SQRT (Q, A, B: in float; //  node 0
                X: out float);
// Compute X = square root of Q,
//    given that A <= X <= B
   X1, F1, F2, H: float;
begin
   X1 := A;
   X2 := B;                          // node 1
   F1 := Q - X1**2
   H := X2 - X1;
   while (ABS(H) >= 0.001) loop      // node 2
      F2 := Q - X2**2;
      H := - F2 * ((X2-X1)/(F2-F1));
      X1 := X2;                      // node 3
      X2 := X2 + H;
      F1 := F2
   end loop;
   X := (X1 + X2) / 2.;              // node 4
end SQRT;                            // node 5
```

Reaching Definitions



A definition $d_i(x)$ *reaches* a node n_j iff there exists a path from n_i to n_j on which `x` is neither defined nor undefined.

> What definitions reach the reference to `x1` in node 4?
>
> What definitions reach the reference to `H` in node 2?

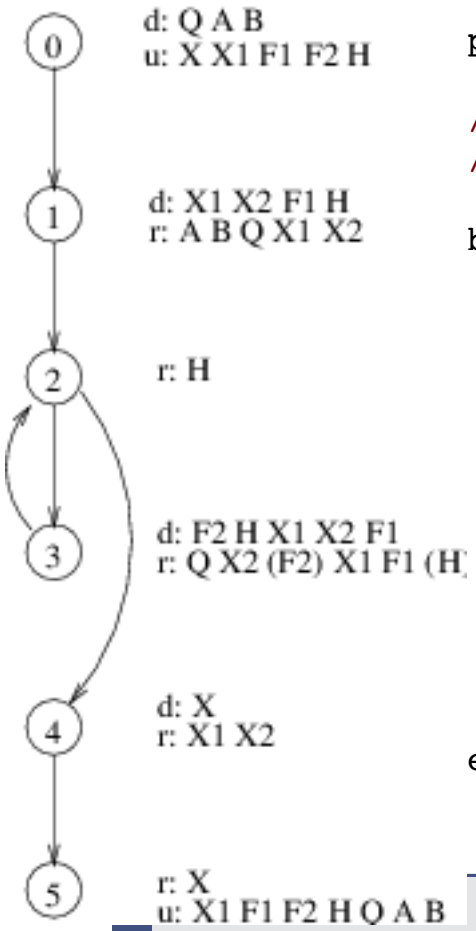
Data Flow Anomalies

The reaching definitions problem can be used to detect anomolous patterns that *may* reflect errors.

- *ur anomalies*: if an undefinition of a variable *reaches* a reference of the same variable
- *dd anomalies*: if a definition of a variable *reaches* a definition of the same variable
- *du anomalies*: if a definition of a variable *reaches* an undefinition of the same variable

Available Expressions

An expression `e` is *available* at a node `n` iff every path from the start of the program to `n` evaluates `e`, and iff, after the last evaluation of `e` on each such path, there are no subsequent definitions or undefinitions to the variables in `e`.



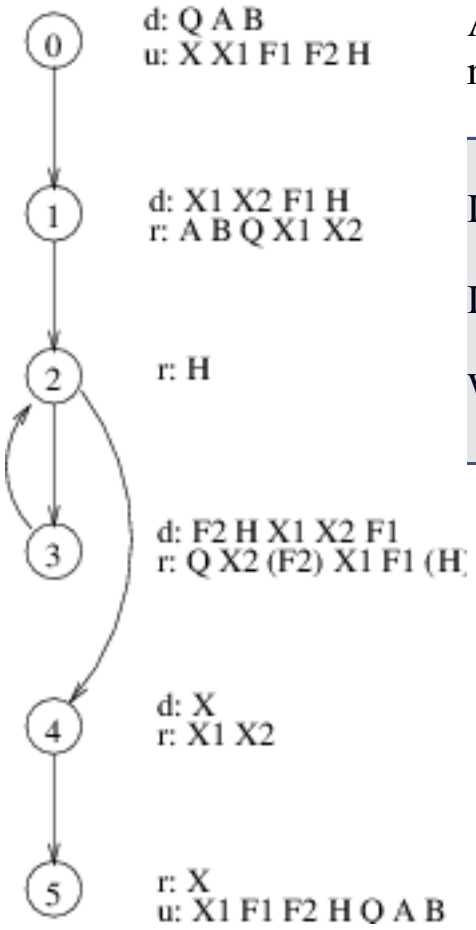

```
procedure SQRT (Q, A, B: in float; //  node 0
                X: out float);
// Compute X = square root of Q,
//    given that A <= X <= B
   X1, F1, F2, H: float;
begin
   X1 := A;
   X2 := B;                          // node 1
   F1 := Q - X1**2
   H := X2 - X1;
   while (ABS(H) >= 0.001) loop      // node 2
      F2 := Q - X2**2;
      H := - F2 * ((X2-X1)/(F2-F1));
      X1 := X2;                      // node 3
      X2 := X2 + H;
      F1 := F2
   end loop;
   X := (X1 + X2) / 2.;              // node 4
end SQRT;                            // node 5
```

Is the expression `x2 - x1` available at the start of node 3?

At the end of node 3?

Same questions for `Q - X2**2`

Live Variables



A variable `x` is *live* at node `n` iff there exists a path starting at `n` along which `x` is used without prior redefinition.

In what nodes in H live?

In what nodes is X1 live?

What does this tell you about memory allocation within this function?

Data Flow and Optimization

Optimization Technique	Data-Flow Information
Constant Propagation	reach
Copy Propagation	reach
Elimination of Common Subexpressions	available
Dead Code Elimination	live, reach
Register Allocation	live

Anomaly Detection	reach
Code Motion	reach

2. Style and Anomaly Checking

A common form of static analysis:

2.1 Lint

Perhaps the first such tool to be widely used, lint (1979) became a staple tool for C programmers.

Combines static analysis with style recommendations, e.g.,

- data flow anomalies
- potential arithmetic overflow
  - e.g., storing an int calculation in a char
- conditional statements with constant values
- potential = versus == confusion

Is there room for lint-like tools?

- lint was a response, in part, to the weak capabilities of early C compilers
- Much of what lint does is now handled by optimizing compilers
  - However compilers seldom do cross-module or even cross-function analysis

2.2 Static Analysis by Compilers

- Over time, compilers offer more and more static analysis features.
  - E.g., GNU `g++`
- One caution is that these are often not turned on by default, but need to be added as command line flags.
  - IDEs often do not use these flags by default.

Analysis Options for g++

`g++` offers several "collections" flags that turn on multiple warnings (which could have been turned on individually).

You explored these in an earlier lab.

- `-Wall`: warnings that GNU considers "useful" and "easily avoidable"

  Examples include:

  - out of bound accesses to arrays, (when compiled with `-O2`)
  - use of C++11 features when the explicit option for these has not been supplied
  - use of `char` in array subscripts
  - abuse of enumeration types in comparisons
  - bad formats in `printf`
  - possibly uninitialized variables

- `-Wextra`: warnings that GNU considers "useful" but that can create *false positives* that can be hard to avoid.

  Examples:

  - empty loop and if bodies,
  - comparisons between signed and unsigned integers,

    - unused function parameters.

- `-pedantic`: warnings required by ISO C++ as for non-standard code.
    - e.g., non-standard file extensions
- `-Weffc++`: warnings about violations of Scott Meyer's *Effective C++*

  Examples:

    - Failing to implement your own version of the copy constructor and assignment operator for classes that have dynamic allocation.
    - `operator=` implementations that fail to return `*this`,
    - the use of assignment rather than initialization within constructors

2.3 CheckStyle

checkstyle is a tool for enforcing Java coding standards.

- Focus is on the more cosmetic aspects of coding, e.g.:
    - Non-empty Javadoc comments for all classes and members.
    - No `*` imports.
    - Lines, function bodies, files not too long.
    - `{ }` used around even single-statement loop bodies and if-then-else bodies.
    - Whitespace used uniformly.
- Can be run via
    - ant
    - maven
    - eclipse
        - Checkstyle messages appear as warnings/errors in the Java editor.

2.4 FindBugs

- Open source project from U.Md.
- Works on compiled Java bytecode
- Sample report
- Can be run via
    - GUI
    - ant
    - Eclipse
    - maven

What Bugs does FindBugs Find?

Unlike Checkstyle, FindBugs goes well beyond cosmetics:

- "Bugs" categorized as
    - Correctness bug: an apparent coding mistake
    - Bad Practice: violations of recommended coding practices.
    - Dodgy: code that is "confusing, anomalous, or written in a way that leads itself to errors"
- Bugs are also given "priorities" (p1, p2, p3 from high to low)
- Bug list

2.5 PMD

Another good tool for finding non-cosmetic problems in your code:

- PMD, source analysis for Java, JavaScript, XSL
    - CPD, "copy-paste-detector" for many programming languages

Can find large repeated code segments that might be better pulled out into a single function.

- Works on source code
- Sample reports (PMD & CPD)
- Can be run via
  - ant
  - maven
  - [eclipse](http://pmd.sourceforge.net/pmd–4.3.0/integrations.html]

PMD Reports

- Configured by selecting "rule set" modules
  - Otherwise, appears to lack categories & priorities
- Reports provide cross reference to source location

3. Reverse-Engineering Tools

Reverse engineering makes heavy use of static analysis, and is even more closely tied to compiler technology than the tools we have looked at so far.

3.1 Reverse Compilers

a.k.a. "uncompilers"

- Generate source code from object code
- Originally clunky & more of a curiosity than usable tools
  - Improvements based on
    - "deep" knowledge of compilers (aided by increasingly limited field of available compilers)
    - Information-rich object codes (e.g., Java bytecode formats)
- Legitimate uses include
  - reverse-engineering
  - generating input for source-based analysis tools
- But also great tools for plagiarism

Java and Decompilation

- Java is a particularly friendly field for decompilers
  - Rich object code format
  - Nearly monopolistic compiler suite

Example of Java Decompilation

For example, I might write the following code:

```
void drawGraphics(Graphics g, Point[] pts)
{
  double xMin = pts[0].x;
  double xMax = pts[0].x;
  double yMin = pts[0].y;
  double yMax = pts[0].y;
```

- If I compile this with the `-g` debugging option on (which saves variable names and other internal information so that a debugger can access them), and run any of several well-known decompilers, I would get back the same code, with only formatting changes.
- If I compile this code without debugging info, one well-known decompiler would give me this:

```
void drawGraphics(Graphics g, Point[] pts)
```

```
{
  double d0 = pts[0].x;
  double d1 = pts[0].x;
  double d2 = pts[0].y;
  double d3 = pts[0].y;
```

Compiled Java `.class` files always preserve the API info.

Defending Against Decompilers

- Options for "protecting" programs compiled in Java:
  - gjc: compile into native code with a far less popular compiler
  - obfuscators…

3.2 Java Obfuscators

Work by a combination of

- Renaming variables, functions, and classes to meaningless, innocuous, and very similar name sets
  - Challenge is to preserve those names of entry points needed to execute a program or applet or make calls upon a library's public API
- Stripping away debugging information (e.g., source code file names and line numbers associated with blocks of code)
- Applying optimization techniques to reduce code size while also confusing the object-to-source mapping
- Replacing some expressions by calls to "dummy" functions that actually simply compute the replaced expression.

3.3 Obfuscation Example

Example, given the compiled code from

```
void drawGraphics(Graphics g, Point[] pts)
{
  double xMin = pts[0].x;
  double xMax = pts[0].x;
  double yMin = pts[0].y;
  double yMax = pts[0].y;
```

the obfuscator yguard will rewrite the code so that the best that a decompiler could produce is:

```
void a(Graphics a, Point[] b)
{
  double d0;
  double d1;
  double d2;
  double d3;
  _mthfor(d0, _mthdo(b, 0));
  _mthfor(d1, _mthdo(b, 0));
  _mthfor(d2, _mthif(b, 0));
  _mthfor(d3, _mthif(b, 0));
```

4. Dynamic Analysis Tools

Not all useful analysis can be done statically

- Profiling
- Memory leaks, corruption, etc.
- Data structure abuse

Abusing Data Structures

- Traditionally, the C++ standard library does not check for common abuses such as over-filling and array or accessing non-

existent elements
    - Various authors have filled in with "checking" implementations of the library for use during testing and debugging

- In a sense, the assert command of C++ and Java is the language's own extension mechanism for such checks.

4.1 Pointer/Memory Errors

Memory Abuse

- Pointer errors in C++ are both common and frustrating
    - Traditionally unchecked by standard run-time systems
- Monitors can be added to help catch these
    - In C++, link in a replacement for malloc & free

How to Catch Pointer Errors

- Use *fenceposts* around allocated blocks of memory
    - check for unchanged fenceposts to detect over-writes
    - Check for fenceposts before a delete to detect attempts to delete addresses other than the start of an allocated block
- Add tracking info to allocated blocks indicating location of the allocation call
    - Scan heap at end of program for unrecovered blocks of memory
    - Report on locations from which those were allocated
- Add a "freed" bit to allocated blocks that is cleared when first allocated and set when the block is freed
    - Detect when a block is freed twice

Memory Analysis Tools

- Purify is a well-known commercial (pricey) tool

- At the other end of the spectrum, LeakTracer is a small, simple, but capable open source package that I've used for many years
    - Works with gcc/g++/gdb compiler suite

** Sample of Leaktracer Output **

```
Gathered 8 (8 unique) points of data.
(gdb)
Allocations: 1 / Size: 36
0x80608e6 is in NullArcableInstance::NullArcableInstance(void) (Machine.cc:40).
39      public:
40          NullArcableInstance() : ArcableInstance(new NullArcable) {}

Allocations: 1 / Size: 8
0x8055b02 is in init_types(void) (Type.cc:119).
118 void init_types() {
119 Type::Integer = new IntegerType;

Allocations: 1 / Size: 132 (new[])
0x805f4ab is in Hashtable<NativeCallable, String, false, true>::Hashtable(unsigned int) (ea/h/Hashtable.h:15).
14 Hashtable (uint _size = 32) : size(_size), count(0) {
15 table = new List<E, own> [size];
```

4.2 Profilers

Profilers provide info on where a program is speding most of its execution time

- May express measurements in
    - Elapsed time
    - Number of executions
- Granularity may be at level of
    - functions
    - individual lines of code
- Measurement may be via
    - Probes inserted into code

- Statistical sampling of CPU program counter register

Profiling Tools

- gprof for C/C++, part of the GNU compiler suite
  - Refer back to earlier lesson on statement and branch coverage
  - gprof is, essentially, the generalization of gcov
- jvisualm for Java, part of the Java SDK
- Provides multiple monitoring tools, including both CPU and memory profiling

---