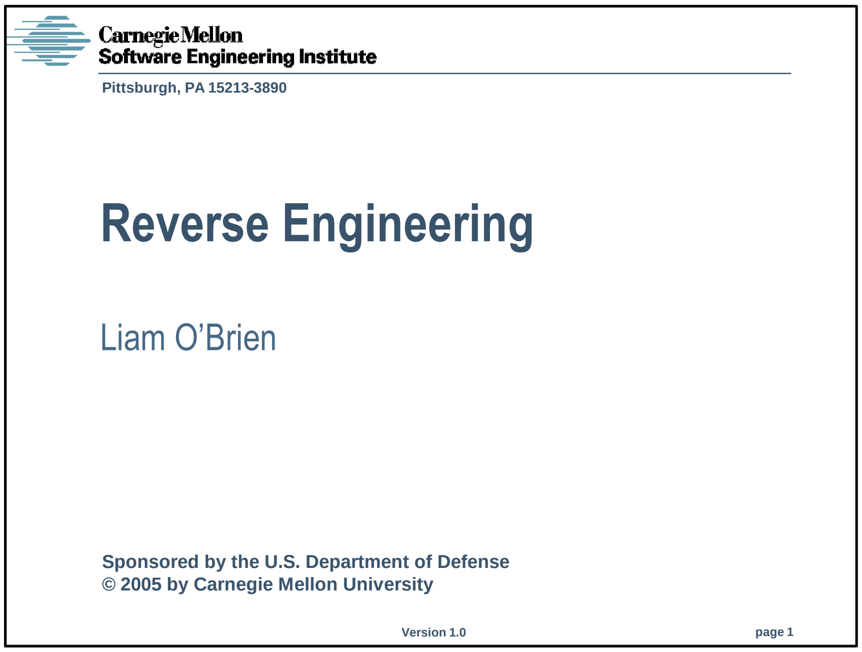
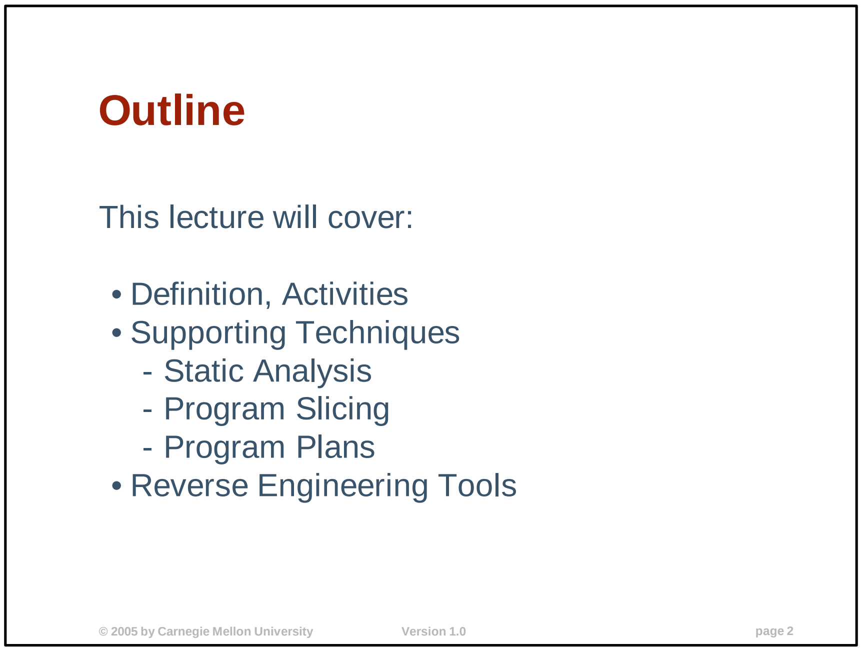


Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Reverse Engineering

Liam O'Brien



Outline

This lecture will cover:

- Definition, Activities
- Supporting Techniques
  - Static Analysis
  - Program Slicing
  - Program Plans
- Reverse Engineering Tools

Reverse Engineering

Reverse Engineering supports understanding of a system through identification of the components or artifacts of the system, discovering relationships between them and generating abstractions of that information.

The goal of reverse engineering is not to alter the system in any way.

Reference:
E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery - A Taxonomy," IEEE Software, Jan. 1990, 13-17.

Reverse Engineering Activities

The three main Reverse Engineering activities:

- Data Gathering
- Knowledge Organization
- Information Exploration

Data Gathering

Raw data is used to identify a system's artifacts and relationships

Techniques include:

- Static source code analysis (parsing)
- Dynamic Analysis (profiling)
- Informal extraction (interviewing)

Knowledge Organization

Goals of Knowledge Organization are:

- Efficient storage of knowledge
- Permit automated analysis
- Reflect user's perspective

Classical data models

- Hierarchical
- Network
- Relational

Abstraction Mechanisms

Abstraction: Selective emphasis on detail

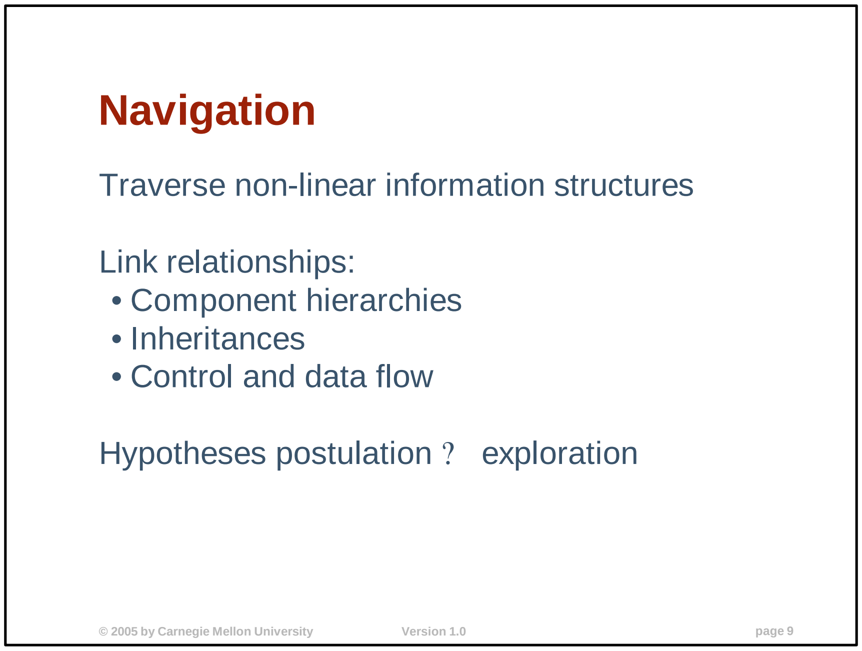
Common Mechanisms:
- Classification
- Aggregation
- Generalization

Conceptual Modeling

Information Exploration

Probably the most important activity:
- Data gathering: necessary to begin
- Knowledge organization: structure model
- Information Exploration: understanding

Composite Activities:
- Navigation
- Analysis
- Presentation



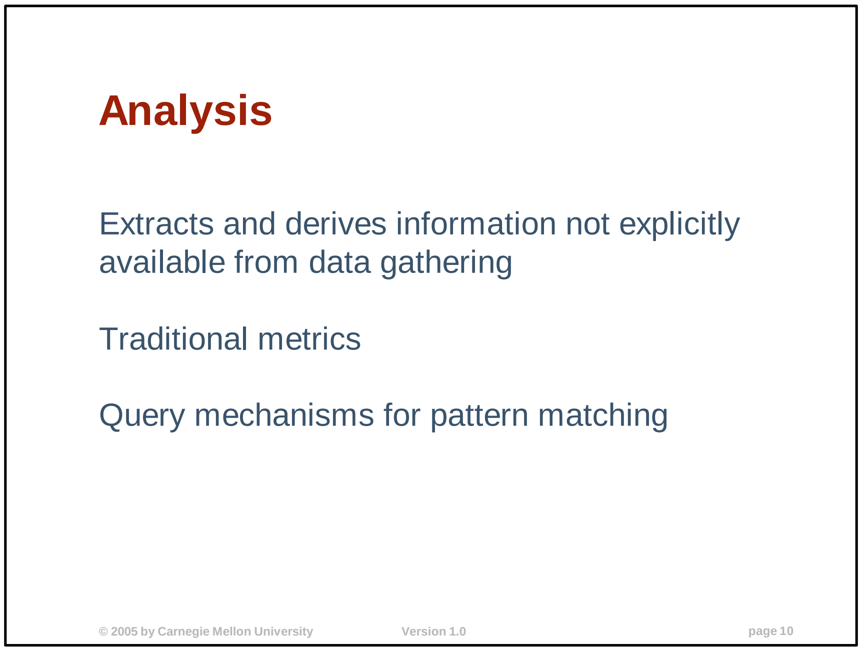
Navigation

Traverse non-linear information structures

Link relationships:

- Component hierarchies
- Inheritances
- Control and data flow

Hypotheses postulation ? exploration



Analysis

Extracts and derives information not explicitly available from data gathering

Traditional metrics

Query mechanisms for pattern matching

Presentation

Spatial and visual data

Descriptive and depictive information

Sense integration: look, feel, etc

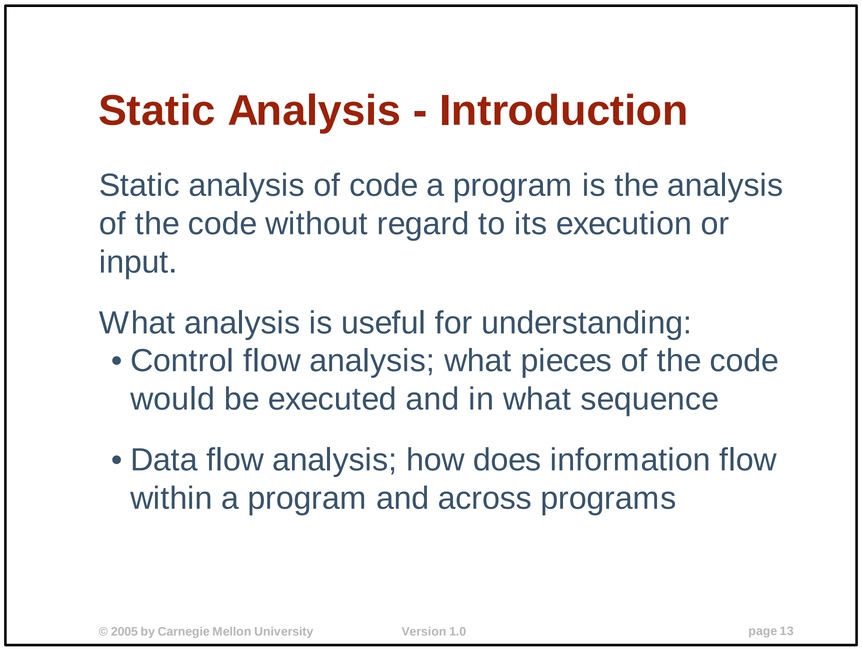
Some current issues:

- Integration of various visual representations
- High flexibility
- Context based visualization
- Integration of various visualization techniques

Reverse Engineering – Supporting Techniques

Techniques

- Static Analysis
  - Control Flow
  - Data Flow
- Program Slicing
- Program Plans

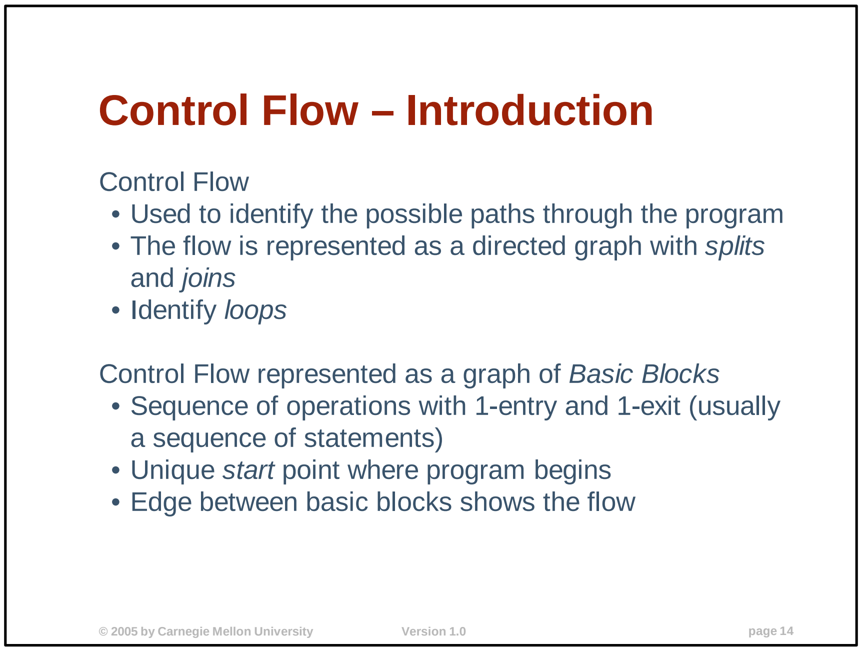


Static Analysis - Introduction

Static analysis of code a program is the analysis of the code without regard to its execution or input.

What analysis is useful for understanding:

- Control flow analysis; what pieces of the code would be executed and in what sequence
- Data flow analysis; how does information flow within a program and across programs



Control Flow – Introduction

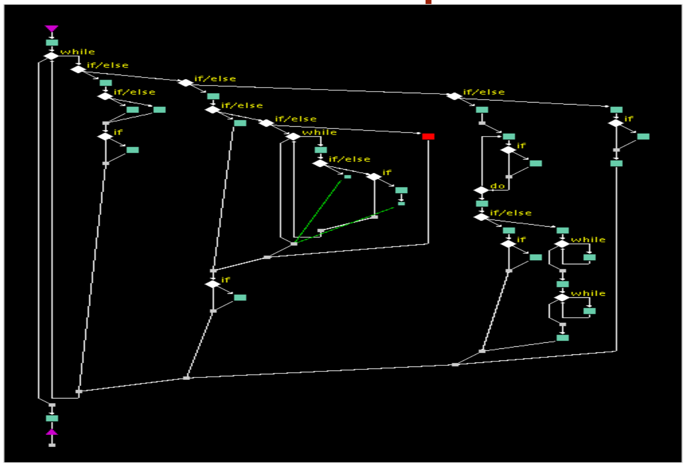
Control Flow

- Used to identify the possible paths through the program
- The flow is represented as a directed graph with *splits* and *joins*
- Identify *loops*

Control Flow represented as a graph of *Basic Blocks*

- Sequence of operations with 1-entry and 1-exit (usually a sequence of statements)
- Unique *start* point where program begins
- Edge between basic blocks shows the flow

Control Flow – Example



Imagix 4D representation of control flow: http://www.imagix.com

Control Flow – Code View

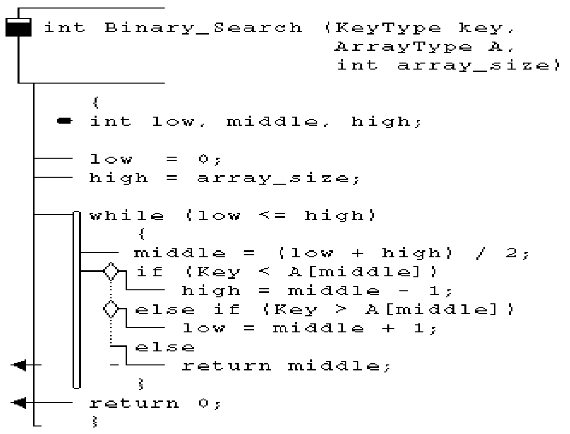
Another example of visualizing the control flow of a program is using a Control Structure Diagram (CSD). CSD is a algorithmic level graphical representation for software.

The following notations are used:

- Sequential flow – straight line
- If/The/Else/Switch statements – diamonds
- For/While – elongated loop
- Loop exit – arrow
- Function – open-ended box

The GRASP project at Auburn University
http://www.eng.auburn.edu/department/cse/research/grasp/

Control Flow – Example - CSD



```
int Binary_Search (KeyType key,
                   ArrayType A,
                   int array_size)
{
int low, middle, high;

low  = 0;
high = array_size;

while (low <= high)
   {
   middle = (low + high) / 2;
   if (Key < A[middle])
      high = middle - 1;
   else if (Key > A[middle])
      low = middle + 1;
   else
      return middle;
   }
return 0;
}
```

Data Flow - Introduction

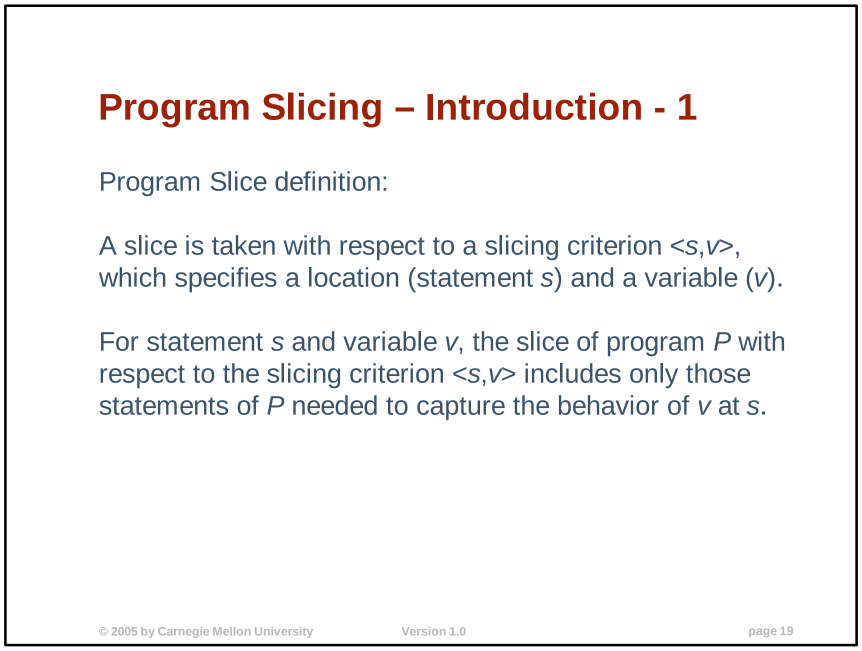
Data Flow is used to analyze the flow of data throughout a program and between program

Local Data Flow Analysis

- Analyze the effects of each statement
  - variable(s) defined
  - set of variable(s) referenced
- Compose the effects to derive information from beginning of each basic block to the statement

Data Flow Analysis

- Propagate basic block information over entire Control Flow Graph

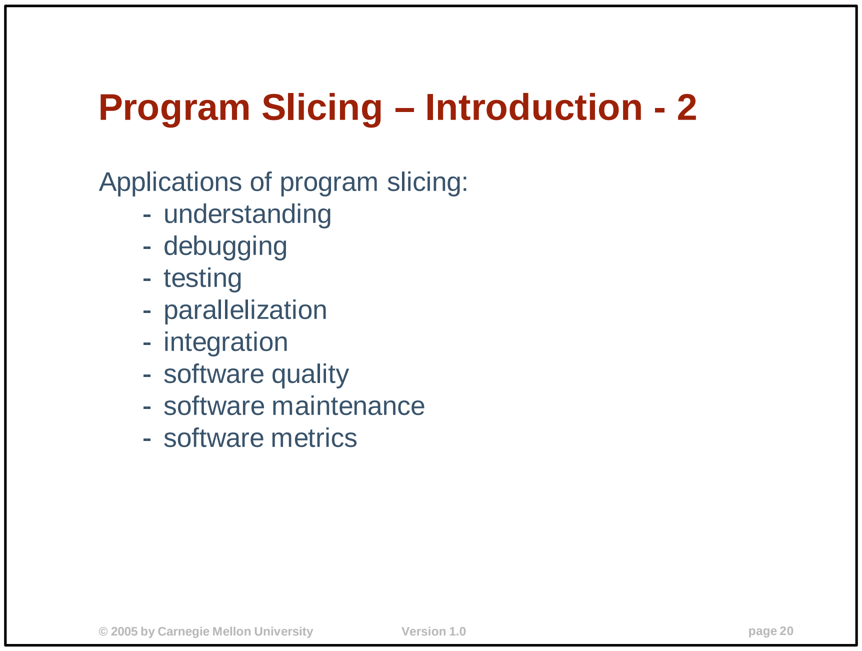


Program Slicing – Introduction - 1

Program Slice definition:

A slice is taken with respect to a slicing criterion <*s*,*v*>, which specifies a location (statement *s*) and a variable (*v*).

For statement *s* and variable *v*, the slice of program *P* with respect to the slicing criterion <*s*,*v*> includes only those statements of *P* needed to capture the behavior of *v* at *s*.



Program Slicing – Introduction - 2

Applications of program slicing:

- understanding
- debugging
- testing
- parallelization
- integration
- software quality
- software maintenance
- software metrics

Program Slicing – Introduction - 3

Program Slicing was first introduced by Weiser. He introduced the concept of an *executable backwards static* slice.

- *executable* - slice is required to be an executable program
- *backwards* – because of the direction the edges are traversed when computing the slice using a dependence graph
- *static* because they are computed as the solution to a static analysis problem (without considering the program’s input)

Many applications of program slicing (such as debugging) do not require executable slices.

M. Weiser, *Program Slicing*, Proceedings of ICSE 1981, 439-449.

Program Slicing – Introduction - 4

Forward slicing (introduced by Horwitz)

- “What statements are affected by the value of *v* at statement *s*?”.

S. Horwitz and T. Reps and D. Binkley, *Interprocedural Slicing using dependence graphs*, Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, 1988.

Dynamic Slicing (introduced by Korel and Laski)

- A slice is computed for a particular fixed input.

B. Korel and J. Laski, *Dynamic Program Slicing*, Information Processing Letters, 29(3), Oct 1988, 155-163

Program Slicing – Flow Graphs - 1

Slicing of a flow graph is a two-step process:

1. Compute the data flow information
2. Use this information to extract a slice

To obtain the data flow information for statement *n* we first obtain:

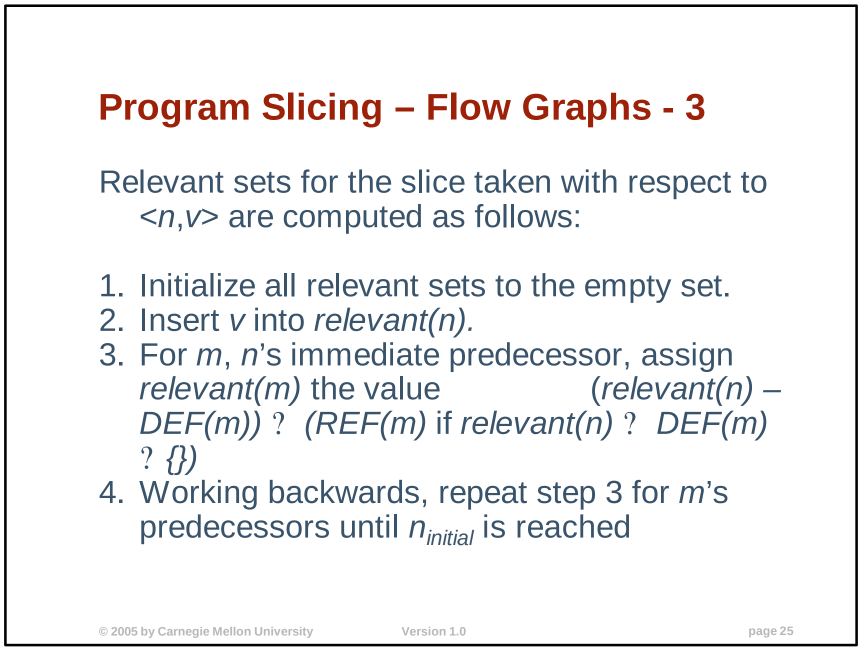
- *REF(n)* – the set of variables that are referenced in *n*
- *DEF(n)* – the set of variables defined (given a value) in *n*

The data flow information is the set of relevant variables at each node *n.*

Program Slicing – Flow Graphs - 2

For the slice with respect to <*s*,*v*> the relevant set for each node contains the variables whose values transitively affect the computation of *v* at *s*.

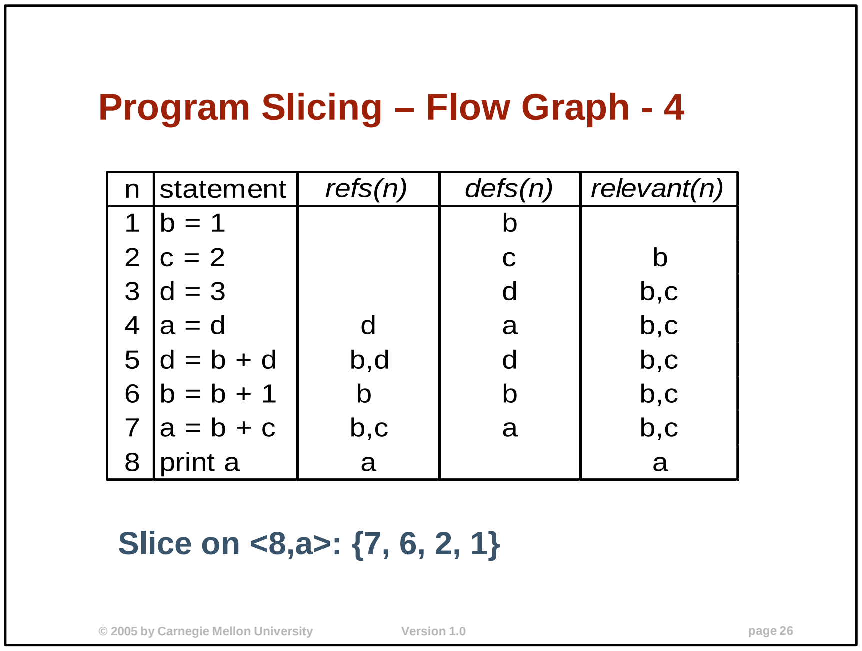
A statement *n* is in the slice if it assigns a value to a variable relevant at *n* and the slice taken with respect to any predicate node that directly controls *n*'s execution.



Program Slicing – Flow Graphs - 3

Relevant sets for the slice taken with respect to <*n*,*v*> are computed as follows:

1. Initialize all relevant sets to the empty set.
2. Insert *v* into *relevant(n).*
3. For *m*, *n*'s immediate predecessor, assign *relevant(m)* the value (*relevant(n)* – *DEF(m))* ? (*REF(m)* if *relevant(n)* ? *DEF(m)* ? *{})*
4. Working backwards, repeat step 3 for *m*'s predecessors until $n_{initial}$ is reached



Program Slicing – Flow Graph - 4

n	statement	*refs(n)*	*defs(n)*	*relevant(n)*
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b,c
4	a = d	d	a	b,c
5	d = b + d	b,d	d	b,c
6	b = b + 1	b	b	b,c
7	a = b + c	b,c	a	b,c
8	print a	a		a

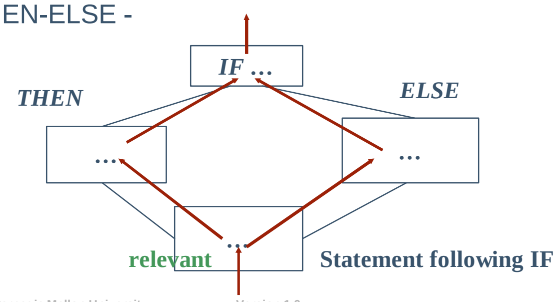
Slice on <8,a>: {7, 6, 2, 1}

Program Slicing – Flow Graph - 5

Slicing with control statements such as:

- IF-THEN-ELSE
- Loop Statements

IF-THEN-ELSE -



Program Slicing – Flow Graph - 6

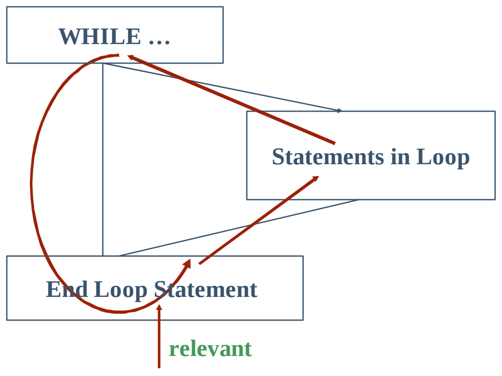
n	statement	*refs(n)*	*defs(n)*	*control(n)*	*relevant(n)*
1	b = 1		b		
2	c = 2		c		b
3	d = 3		d		b,c
4	a = d	d	a		b,c,d
5	if (a) then	a			a,b,c,d
6	d = b + d	b,d	d	5	b,d
7	c = b + d	b,d	c	5	b,d
8	else			5	b,c
9	b = b + 1	b	b	8	b,c
10	d = b + 1	b	d	8	b,c
11	endif				b,c
12	a = b + c	b,c	a		b,c
13	print a	a			a

Slice on <13,a>: {12, 11, 9, 8, 7, 6, 5, 4, 3, 2, 1}

Note: a is included in the relevant set at 5. Otherwise another column.

Program Slicing – Flow Graph - 7

Loop Statements:



Version 1.0

Program Slicing – Flow Graph - 8

Loop Example:

n	statement	*refs(n)*	*defs(n)*	*control(n)*	*relevant(n)* *Iter 1*	*relevant(n)* *Iter 2*
1	b = 1		b			
2	c = 2		c			b
3	d = 5		d			b,c
4	a = 3		a			b,c
5	While (a < 10)	a			a,b,c	a,b,c
6	b = b + c	b,c	b	5	b,c	b,c
7	c = c + 1	c	c	5	b	b,c
8	a = b	b	a	5	b	b,c
9	EndWhile			5	a	
10	print a	a			a	

Slice on <10,a>: {9, 8, 7, 6, 5, 4, 2, 1}

Version 1.0

Program Slicing – Flow Graph - 9

n	statement
0	While (a)
1	$x_n = x_{n-1}$
2	$x_{n-1} = x_{n-2}$
	...
n	$x_2 = x_1$
n + 1	EndWhile

If x_n in slicing criteria - need n passes through loop

Program Slicing - Dynamic

- Dynamic Program Slicing
  - Only the dependencies that occur in a specific execution of the program are taken into account.
  - A *dynamic slicing criterion* specifies the input, occurrence of a statement and a variable
  - dynamic slicing assumes a fixed input for a program whereas a static slice does not make assumptions about the input.
- Hybrid approaches that use both static and dynamic slicing also exist.

Program Slicing – Dynamic e.g. -1

n	statement
1	read(n)
2	i := 1
3	while (i <= n) do
4	begin
5	if (i mod 2 = 0) then
6	x := 17
7	else
8	x := 18;
9	i := i + 1
10	end;
11	write(x);

What is dynamic slice with criterion (n = 2, 11, x)?

Program Slicing – Dynamic e.g. -2

n	statement
1	read(n)
2	i := 1
3	while (i <= n) do
4	begin
5	if (i mod 2 = 0) then
6	x := 17
7	else
8	
9	i := i + 1
10	end;
11	write(x);

Dynamic slice with criterion (n=2, 11, x) is entire program without line 8.

Static slice (11, x) is the entire program.

Program Plans – Introduction

The goal is to recognize clichés using plans.

A cliché is a pattern that appears frequently in programs (e.g., algorithms, data structures, domain-specific patterns).

A plan is an abstract representation of a cliché.

Representation is at the semantic level rather than at the syntactic level.

Clichés - Examples

Data structure clichés: lists, trees, tables, vectors, matrices

Algorithmic clichés: list, tree, graph traversals; iterators, applicators, manipulators; linear, binary, hash searches; event handler; exception handler

ADT clichés: dictionary, priority queue, heaps

Plan Recognition

Approaches:

- Top-down: Start with set of goals to be achieved; determine what plans can achieve these goals; connect these plans to source code patterns.
  - Problem: Requires detailed advance knowledge otherwise connection to code is unrealistic
- Bottom-up: Start with source code; identify plans that match source code; infer higher-level goals from these plans.
  - Problem: Combinatorial explosion of alternatives
- Hybrid: top-down and bottom-up

Plan Recognition – Method

Typical method of plan recognition

- An effective (language-independent) program representation
- A translator to transform source text into this program representation
- A library of programming plans representing clichés at various levels of abstraction
- A plan recognizer which parses the program to recognize plans stored in the library
- The result is a tree or lattice with program components at the leaves, programming plans, and the goals of the program at the root
- Bottom-up program understanding

Plan Recognition - Issues

Syntactic variations: Recognizer works on the basis of structure information only; syntactic variations lead to the same paraphrase, modulo identifiers

Non-contiguousness: Recognizer works with graph structures, can accommodate equivalent sequences of statements

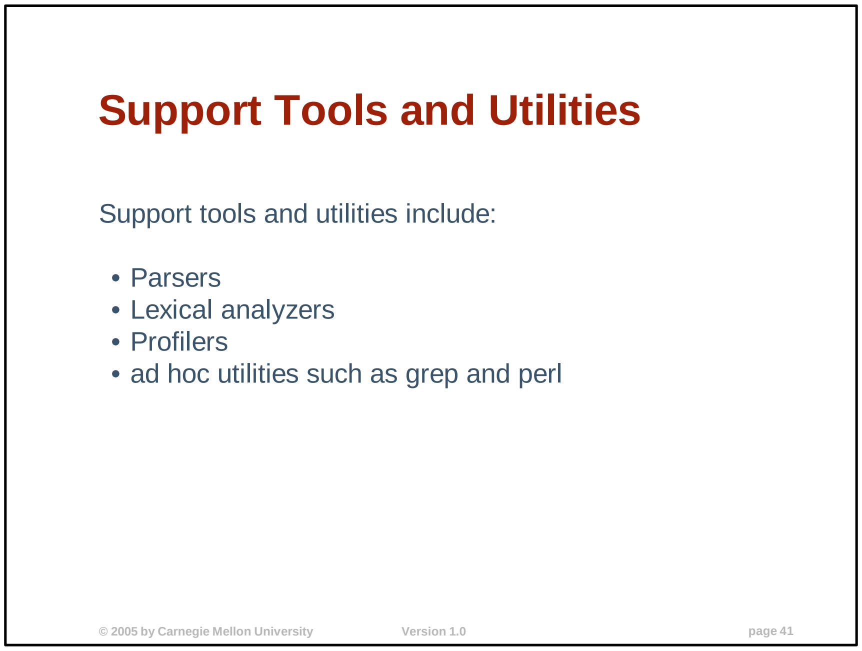
Implementation variations: Similar programs are matched against the same plans, lead to the same paraphrases

Recognition algorithm depends polynomially on size of the program and plan library; graph grammars and graph recognition algorithms deployed

Reverse Engineering - Tools

Types of tools:

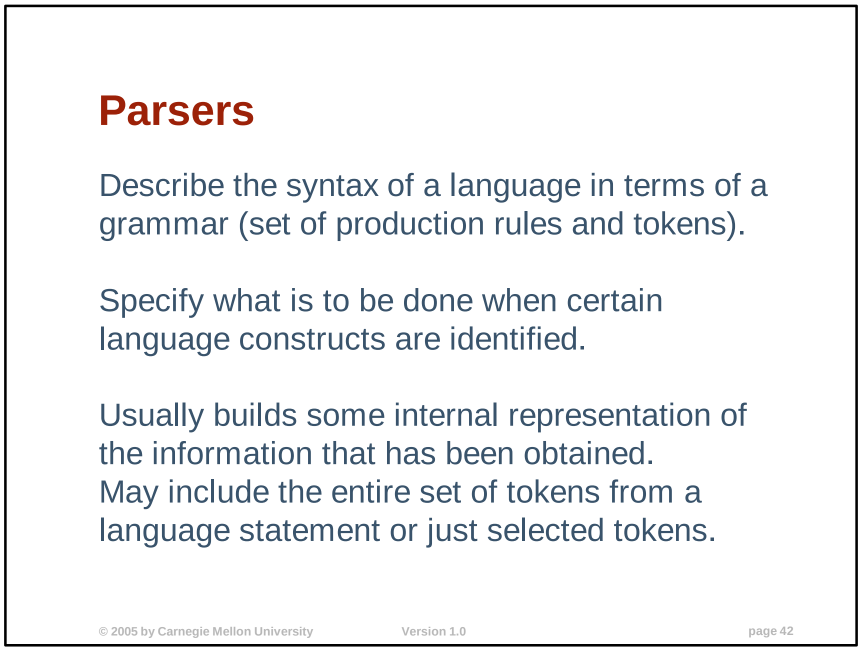
- Support tools and utilities
- Analysis tools
- Reverse Engineering environments
- Tool environments for building tools
- Integrated forward and reverse engineering environments



Support Tools and Utilities

Support tools and utilities include:

- Parsers
- Lexical analyzers
- Profilers
- ad hoc utilities such as grep and perl



Parsers

Describe the syntax of a language in terms of a grammar (set of production rules and tokens).

Specify what is to be done when certain language constructs are identified.

Usually builds some internal representation of the information that has been obtained.
May include the entire set of tokens from a language statement or just selected tokens.

Parsers: Grammar example

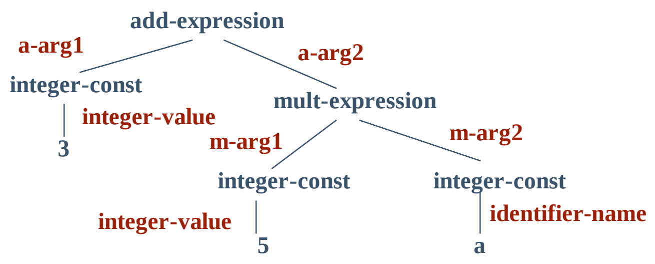
```
!!in-grammar(‘ syntax)
grammar CALC
  productions
    identifier   ::= [identifier-name]       builds identifier,
    int-const    ::= [integer-value]         builds integer-const,
    add-expr     ::= [a-arg1 “+” a-arg2]   builds add-expression,
    mult-expr   ::= [m-arg1 “*” m-arg2]   builds mult-expression,

  precedence
     for calc-expression brackets “(“ matching “)”
         (same-level “+” associativity left),
         (same-level “*” associativity left)
end
```

From Refine Programming Language – Reasoning.com

Parsers: Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a representation of the grammar that’s populated by the parser.
We parse the expression (3 + (5 * a)) and get the following AST.



Lexical Analyzers

Identify some set of tokens from a statement within a program.

Examples would be identify “#include” token and the name of the file included.

Useful for extracting call graphs, file dependencies, etc.

Lexical Analyzers vs Parsers

Why do lexical analysis rather than parsing?

Parsing is expensive in terms of time and space

Each programming language requires a new parser to be written (even different versions of the same language require new parsers)

Parsers usually require complete code that can be compiled whereas lexical analyzers do not.

Profilers

Obtain dynamic (run-time) information about a system.

Some information may not available statically, due to *late binding*:
- polymorphism
- function pointers
- run-time parameterization

Other ways of obtaining dynamic information?

- **Code instrumentation**
- **Use of debugging tool (less efficient)**

Analysis Tools

Analysis Tools:
- Extract software artifacts including
  - control flow graphs
  - call graphs (structure)
  - cross-references,
  - global variables,
  - types and constants,
  - pointer analyses (aliasing),
  - metrics

Tool Demo

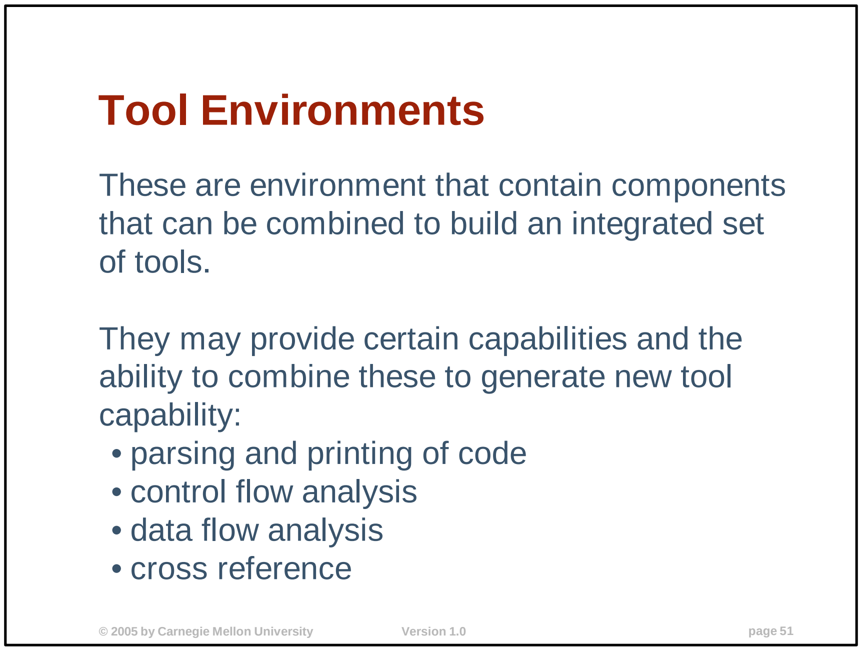
Understand for C/C++ from Scientific Toolworks, Inc

www.scitools.com

Understanding Environments

Understanding Environments:

- Parsing engine, repository, user interface
- Store extracted software artifacts in a repository
- Interactive tool to navigate, browse, explore search, query repository
- Syntactic, functional, behavioral pattern matching and filters
- Forming abstract representations

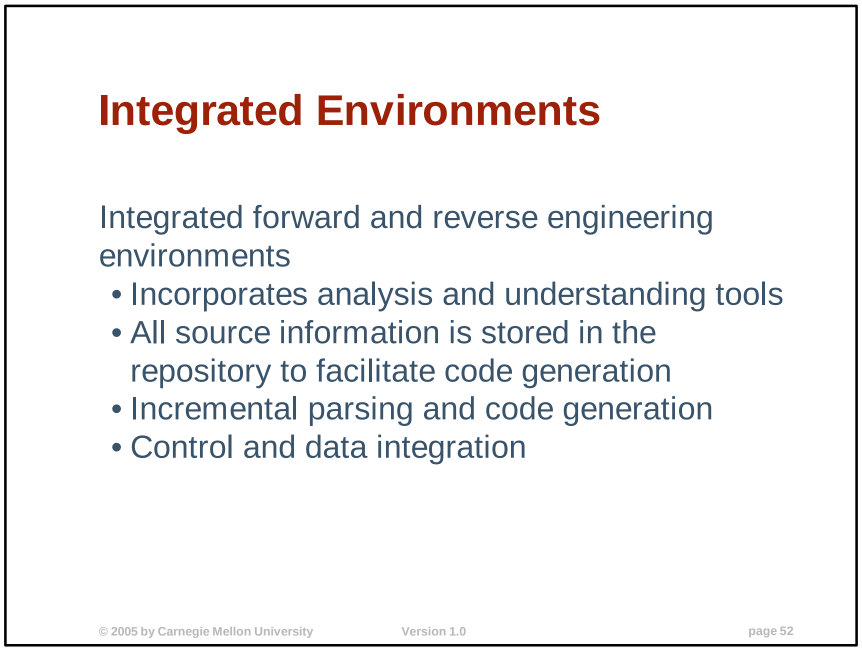


Tool Environments

These are environment that contain components that can be combined to build an integrated set of tools.

They may provide certain capabilities and the ability to combine these to generate new tool capability:

- parsing and printing of code
- control flow analysis
- data flow analysis
- cross reference

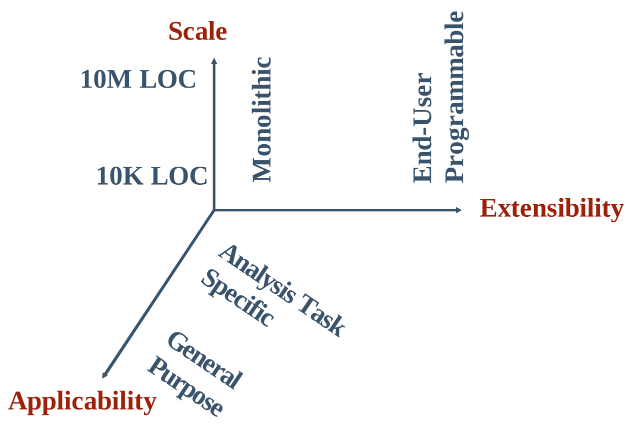


Integrated Environments

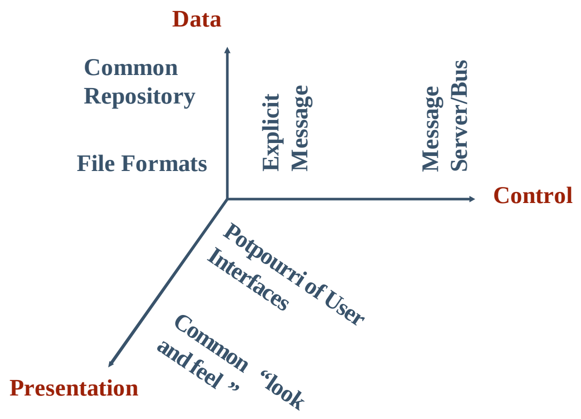
Integrated forward and reverse engineering environments

- Incorporates analysis and understanding tools
- All source information is stored in the repository to facilitate code generation
- Incremental parsing and code generation
- Control and data integration

Design Space for Tools



Tool Integration Dimensions



Tool Integration - GXL

GXL (Graph eXchange Language) is designed to be a standard exchange format for graphs. It is an XML sublanguage.

This exchange format offers an adaptable and flexible means to support interoperability between graph-based tools such as parsers, control-flow analyzers, program slicers, etc.

Represent the data to be shared as a graph and a schema that describes the graph format.

More information: http://www.gupro.de/GXL/

Information sources

Information can be extracted from various sources:

Static
- Source code
- Makefiles
- Documentation
- Interviewing

Dynamic
- Profiling
- Code instrumentation

Some Available Tools

There are lots of tools out there. Examples:

- Imagix 4D (www.imagix.com) - analysis
- Rigi (www.rigi.csc.uvic.ca) - analysis
- SNiFF+ (www.takefive.com) - analysis tool
- Hindsight (www.testersedge.com) - analysis tool
- McCabe IQ (www.mccabe.com) - analysis tool
- Surgeon’s Assistant (Unravel) - program slicing (http://www.cs.loyola.edu/~kbg/Surgeon/)
- …

Dedicated conference:
Working Conference on Reverse Engineering (WCRE)

Summary

In this lecture we talked about Reverse Engineering

Outlined some techniques that support Reverse Engineering
- Static Analysis
- Program Slicing
- Program Plans

Outlined examples of tools and some of the issues with them.