School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 2a: Mathematical Preliminaries Introduction

CSI3105:
Software Testing

Review Questions

1. Explain your understanding of the term “*software testing*”
2. List the different activities in software testing.
3. Define the terms: *errors*, *faults* and *failure*.
4. What is your understanding of the terms: *testing* and *debugging.*
5. Statement: “It is impossible to test a program completely”. Do you think that this statement is true and provide a rationale for your answer.
6. What is a test oracle?

Explain your understanding of the term “*software testing*”

Explain your understanding of the term "*software testing*"

Definition: Testing is the process of executing a program with the intention of finding errors (Myers,1976).

Testing - an unnatural process as its aim is to make the program fail.

- Errors, faults, failure

To be successful in his/her testing activities, the tester must construct his/her test case such that; if faults are present in the software, these faults will be exposed.

It is not possible to guarantee the absence of errors by testing the software using a large set of test cases. The optimal result of testing - the maximum exposure of errors present. **Testing can demonstrate the presence of errors but not their absence.**

Effective testing is a result of adequate preparations being made before hand:

- to arrange the system such that it easy to test and
- by preparing a plan of testing so that the sequence of testing is well organised.

List the different activities in software testing

List the different activities in software testing.

List the different activities in software testing.

Steps in carrying out testing:

Establish test objectives.

Design test cases and writing test cases.

Evaluate the test cases.

Execute the tests using the developed program.

Examine test results (output from the previous step).

Define the terms: *errors*, *faults* and *failure*.

Define the terms:

- *errors*,
- *faults* and,
- *failure*

Define the terms: *errors*, *faults* and *failure*.

Error, Faults, Failures

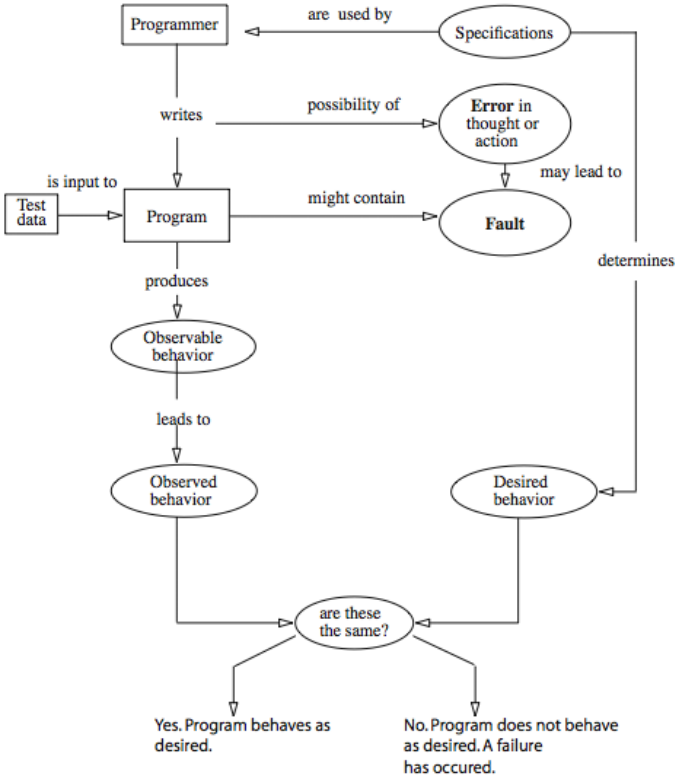


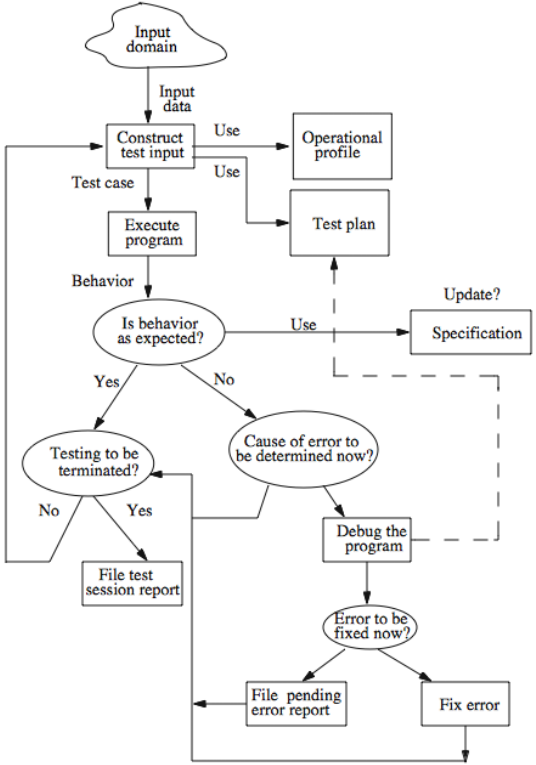
Diagram from Mathur(2014)

What is your understanding of the terms: *testing* and *debugging.*

What is your understanding of the terms:

- *testing*
- d*ebugging*

What is your understanding of the terms: *testing* and *debugging*.



Test/debug cycle

Diagram from Mathur(2014)

What is a test oracle?

Test Oracle?

What is a test oracle?

Test Oracle

- The entity that performs the task of checking the correctness of the observed behavior of a program

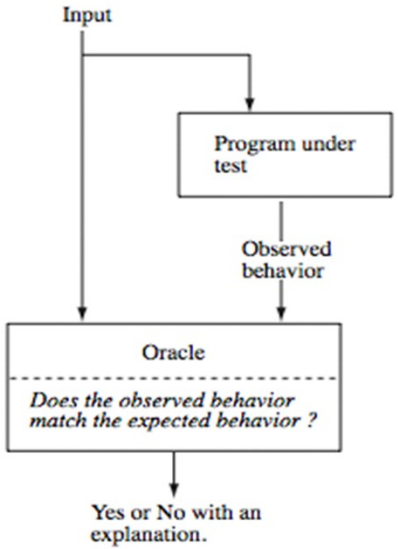


Diagram from Mathur(2014)

Lecture Summary

The learning goal for this week is to familiarize the students with basic mathematical concepts associated with software testing:

- To learn about Predicates and Boolean expressions
- To learn about Control Flow Graph
- To learn about Strings, Languages and Regular Expressions

This module relates to Chapter 2 of the textbook

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 2b: Predicates/ Boolean Expressions

CSI3105:
Software Testing

Predicates and Boolean Expressions

Relational operators

- <, >, ==, !=

Boolean operators/ connectors

- ^ (AND), v (OR), ¬ (NOT)

True ^ False = False
True v False = True
¬ True = False
¬ False = True

Predicates and Boolean Expressions

A Boolean variable has two possible values (true/false) (i.e.1/0).

A Boolean function has a number of Boolean input variables and has a Boolean valued **output** (1/0).

For n variables, there are 2^n Boolean functions.

A Predicate is a Boolean function.

P(a,b) = (a + b < 10)

- Let a = 3 and, b = 2
- P(3,2) = (3 + 2 < 10)
- P = 5 < 10
- P = TRUE

Predicates and Boolean Expressions

Item	Examples	Comment
Simple predicate	p q ^ r A + b < c	p, q, r = Boolean variables
Compound predicate	¬(a + b < c) (a + c < c) ^ (¬p)	Parentheses explicitly group simple predicates
Boolean expression	P ¬p P ^ q v r	Boolean expression is a predicate that doesn't contain any relational operators. Composed of ONE or MORE Boolean variables joined by Boolean operators
Singular boolean expression	p ^ q v ¬r ^ s	p q r s Are all Boolean variables All variables appear once
Non- singular boolean expression	**p** ^ q v ¬r ^ **p**	P appears twice, therefore not singular

Predicates and Boolean Expressions

RECAP!

Boolean operators

Relational operators

Examples:

- Simple Predicates
- Compound Predicates

Predicates and Boolean Expressions

Boolean expression

- Predicates can be converted to Boolean expression
- P = (a + b < c) ^ (!d)
- Boolean v1 = (a + b < c) , Boolean v2 = (!d)
- P = Boolean v2 ^ Boolean v2 ← No more relational operators

  - Singular – example ($x \wedge y \wedge p$) where each literal appears only once
    - ($x \wedge y \vee (x + y)$) y appears more than once therefore non-singular
  - Mutually Singular
    - E= $((a + b > 10) \wedge (c + d < 20) \wedge (!p) \vee (p + q > 5))$
      - if e1 = a + b > 10; e2 = c + d < 20; e1 and e2 are mutually singular
      - If e3 = !p and e4 = p + q >5 ; e3 and e4 are not mutually singular

Precedence of Operators

Typical precedence levels

- parentheses
- unary operators (!, ++)
- ** (exponentiation, if the language supports it)
- *, /, %
- +, -

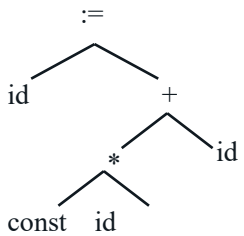
x OR y AND z	is	x OR (y AND z)
x AND y OR z AND t	is	(x AND y) OR (z AND t)
x AND y AND z OR t	Is	((x AND y) AND z) OR t
!x AND y OR z	is	((!x) AND y) OR z

Operator precedence

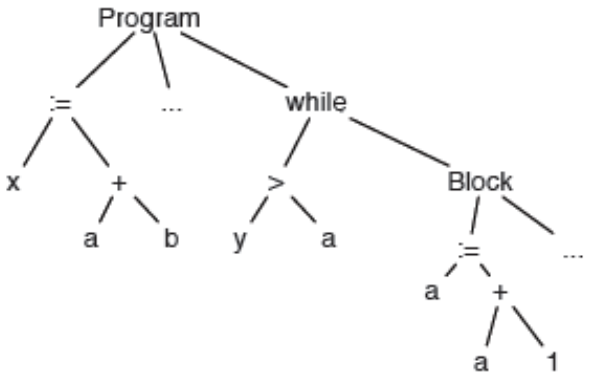
NOT
AND
OR

An Abstract Syntax tree:

abstract syntax tree

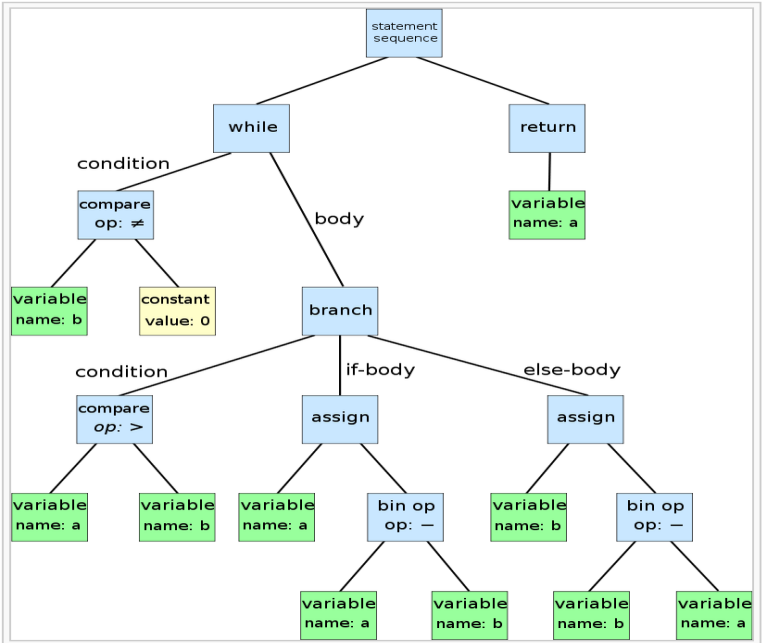



```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```



AST – Example

From:
http://cecs.wright.edu/~pmateti/Courses/7100/Lectures/Syntax/ast-notes.html



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 2c: Control Flow Graphs

CSI3105:
Software Testing

Control Flow Graph

An abstract representation of a program/method that models all executions of a program/method by describing control structures

- Nodes : Statements or sequences of statements (basic blocks)
- Directed Edges : Transfers of control (branch; alternate paths)

Basic Block : A sequence of statements such that if the first statement is executed, all statements will also be executed (no branches). It has unique entry and exit points.

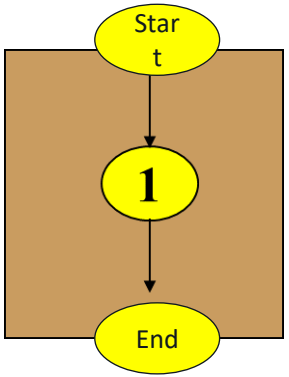
Path : A collection of *Nodes* linked with *Directed Edges*

Control flow graph G= (N, E)

- Finite set of N nodes and finite set of E edges
- Block bi = node ni. An edge (I,j) connects blocks bi and bj

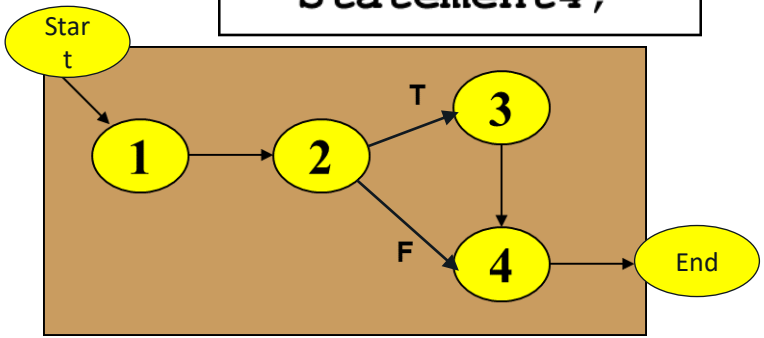
Control Flow Graph (CFG)

Simple Basic block
represented as one node



CFG

```
Statement1;
Statement2;
Statement3;
Statement4;
```




```
Statement1;
Statement2;

if X < 10 then
   Statement3;

Statement4;
```

1
2
3
4

Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

The if Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```

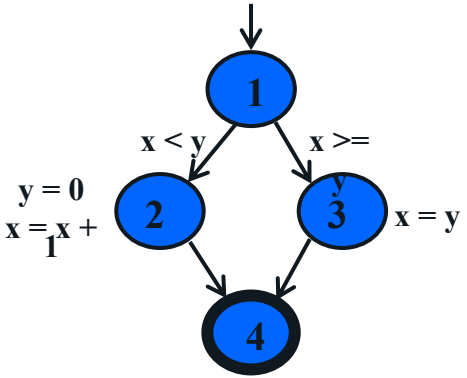
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```

Control Flow Graph (CFG)

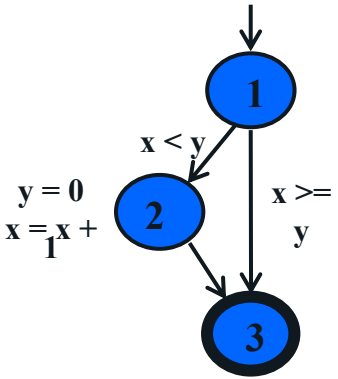
Example – Taken from Amman & Offutt (2008)

The if Statement (CFG)

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```




```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```

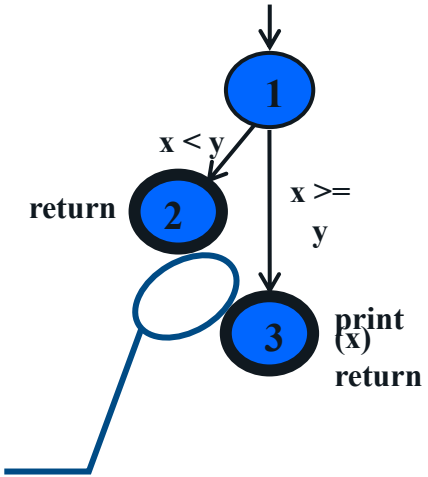


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

The if- Return Statement

```
if (x < y)
{
   return;
}
print (x);
return;
```



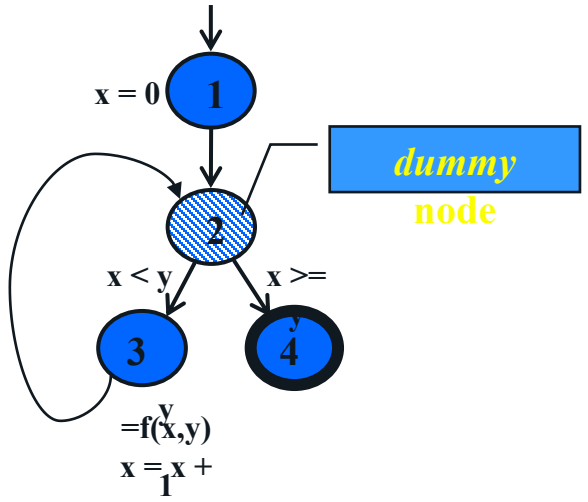
No edge from node 2 to 3.
The return nodes must be distinct.

Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

While Loop

```
x = 0;
while (x <
y)
{
   y = f (x,
y);
   x = x + 1;
}
```

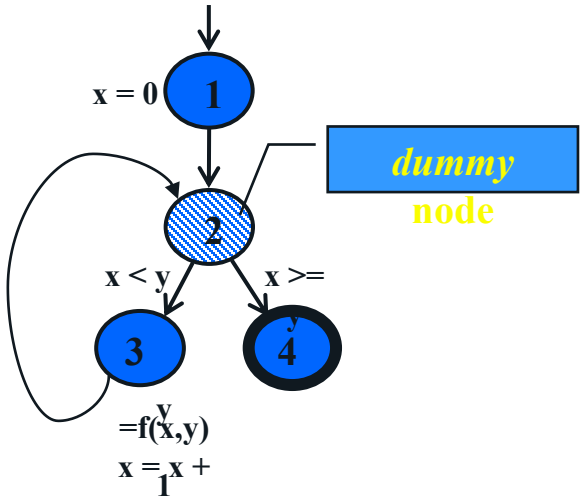


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

While Loop

```
x = 0;
while (x <
y)
{
   y = f (x,
y);
   x = x + 1;
}
```

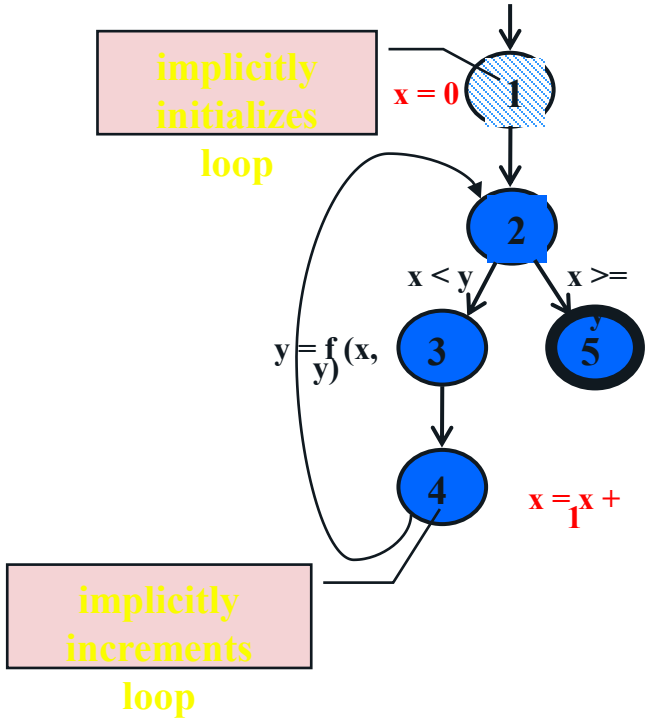


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

For loop

```
for (x = 0; x < y;
x++)
{
   y = f (x, y);
}
```

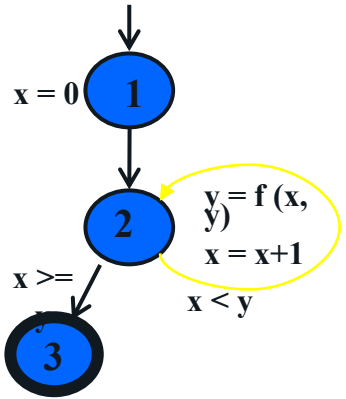


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

Do While loop

```
x = 0;
do
{
   y = f (x, y);
   x = x + 1;
} while (x < y);
println (y)
```

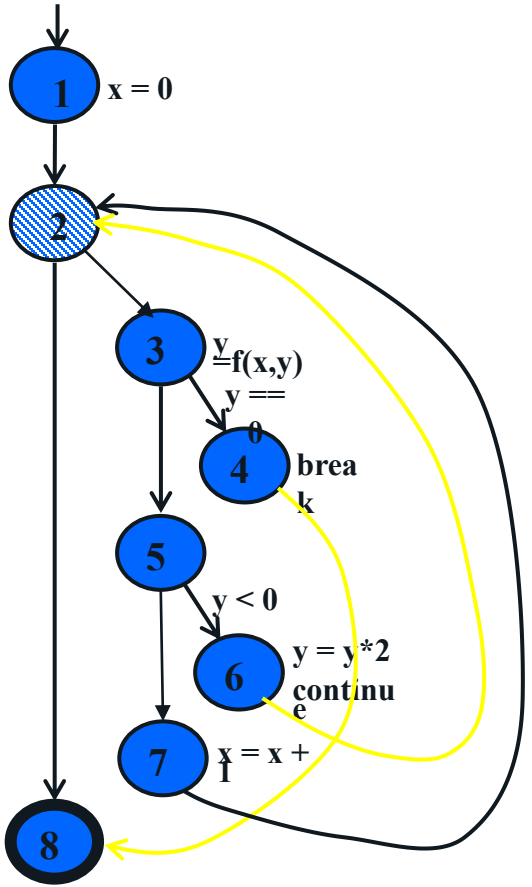


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

Loop with break and Continue

```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y <
0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```

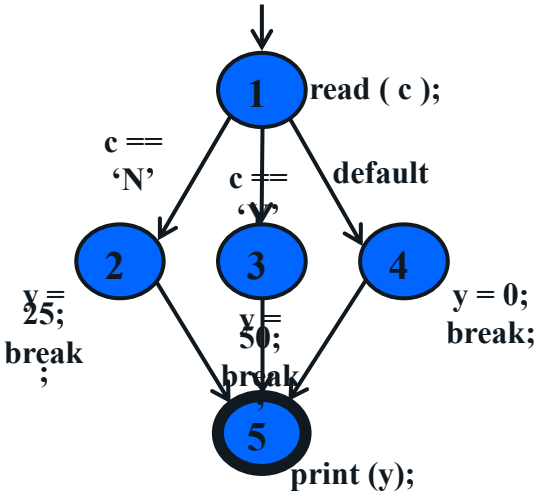


Control Flow Graph (CFG)

Example – Taken from Amman & Offutt (2008)

The case (Switch) Statement

```
read ( c) ;
switch ( c )
{
   case 'N':
      y = 25;
      break;
   case 'Y':
      y = 50;
      break;
   default:
      y = 0;
      break;
}
print (y);
```

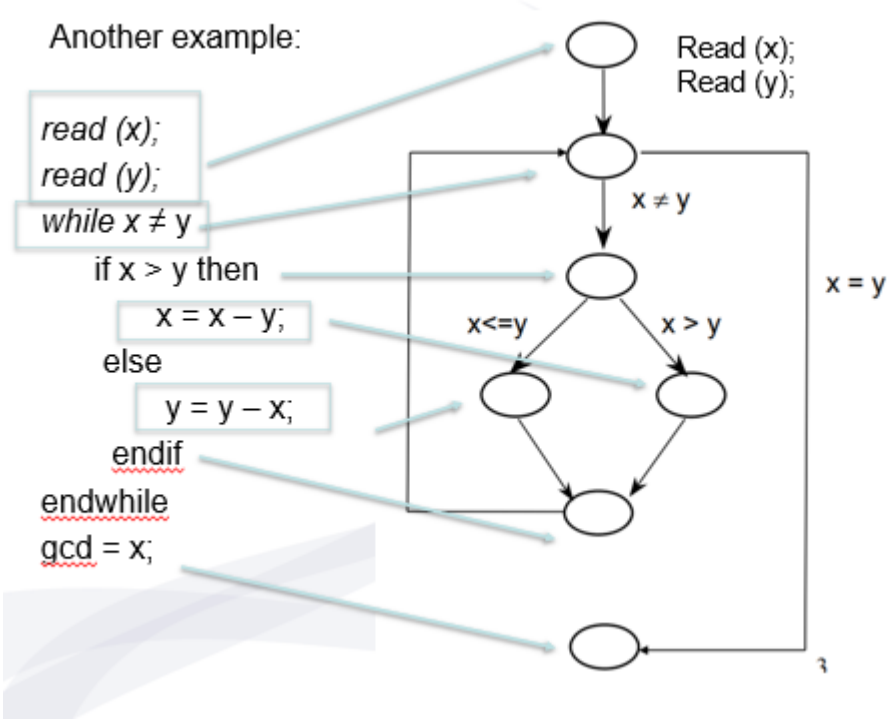


Control Flow Graph (CFG)

Another example:

```
read (x);
read (y);
while x ≠ y
      if x > y then
            x = x – y;
       else
             y = y – x;
        endif
endwhile
gcd = x;
```

Control Flow Graph (CFG)

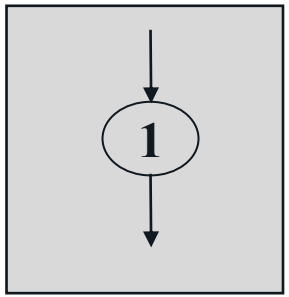


Number of Paths through CFG

Given a program, how do we exercise all statements and branches at least once?

Steps:

- Translate the program into a CFG, work out the set of paths that can cover all arcs and nodes in the CFG
  - Path: A sequence of nodes – [n_1, n_2, …, n_M]
  - Each pair of nodes is an edge (e.g. n_1, n_2 is an edge)
  - Length of path: The number of edges
- Example:



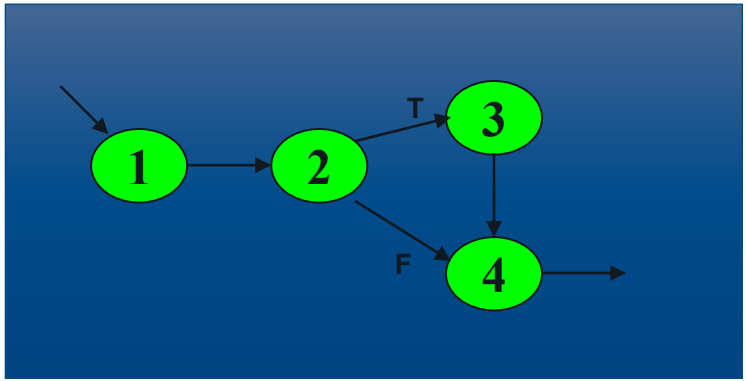
Only one path is needed for this CFG -- **[1]**

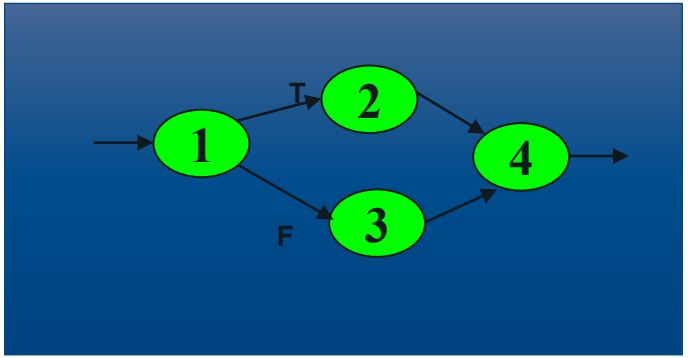
A single node is a path of length 0

Number of Paths through CFG

Two Paths:

- **[1 – 2 - 4]**
- **[1 - 2 - 3 - 4]**





Two Paths:

- **[1 – 2 - 4]**
- **[1 - 3 - 4]**

Paths through CFG

A **Simple** path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same

- No internal loops
- A loop is a simple path

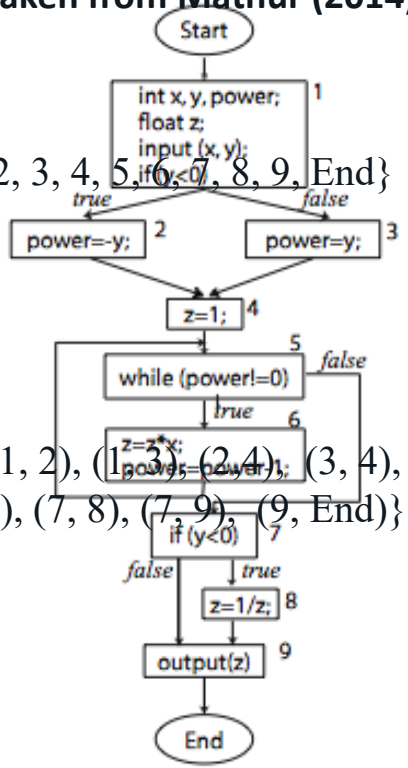
A **Complete** path – if the first node along the path is **Start** and the terminating node is **End**.

A **Feasible** path – there exist at least one testcase with inputs that will transverse the path *p*

An **Infeasible** path – No testcase with inputs that will transverse the path *p*

Control Flow Graph (CFG)

Example – Taken from Mathur (2014)



N={Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End}

E={(Start,1), (1, 2), (1, 3), (2,4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)}

Sample Paths Through a CFG

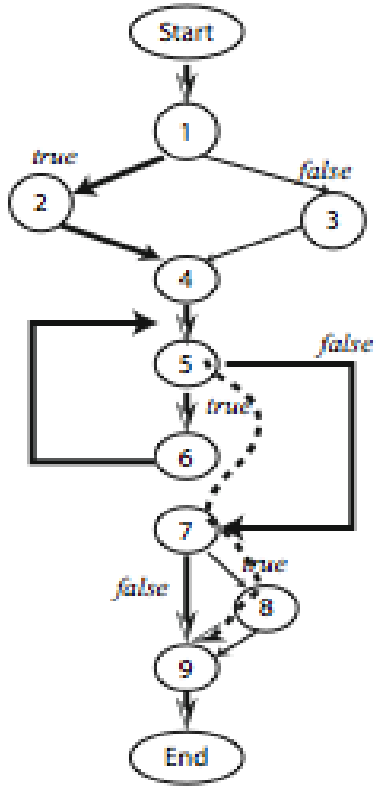
Two feasible and complete paths:

p_1= (Start, 1, 2, 4, 5, 6, 5, 7, 9, End)

Bold edges: complete path.
Dashed edges: subpath.

Specified unambiguously using edges:
p_1= ((Start, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, End))

p_2= (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)



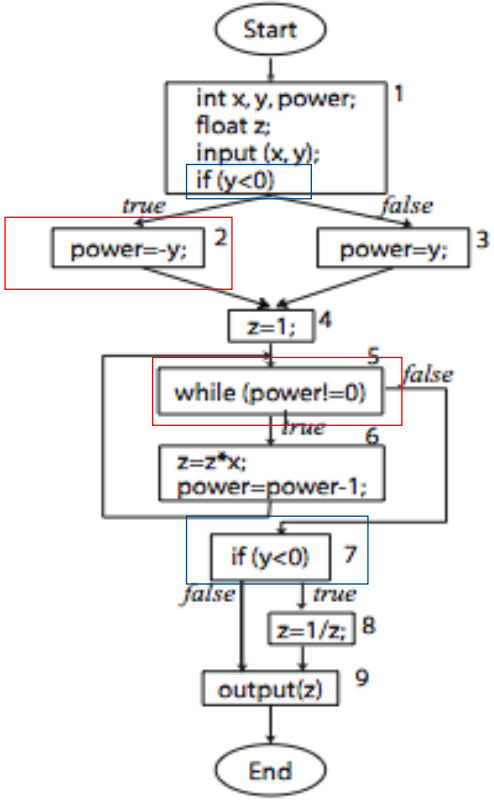
Taken from Mathur ,2014

Sample Paths Through a CFG

Paths: infeasible

p_1= (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)

p_2= (Start, 1, 2, 4, 5, 7, 9, , End)



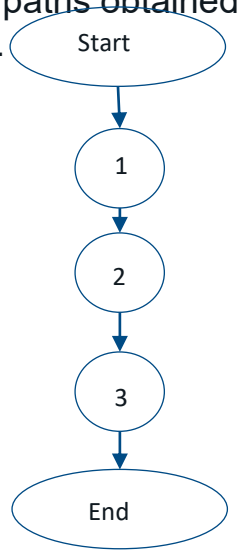
Taken from Mathur ,2014

Basis Paths

Given a Program **P** and its associated CFG, **G**, a basis path set (also known as basis set) consists of one or more linearly independent complete paths through **G**. Each path in this set is known as basis path.

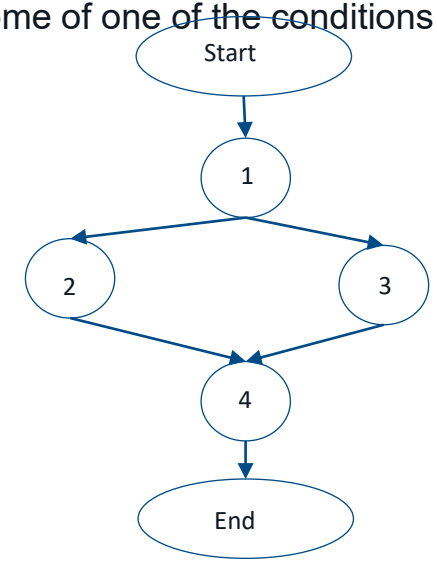
The number of basis paths is equal to the cyclomatic complexity of **G** (V(G)).

The first path could be any complete path through G, starting with the start node, ends with the end node and does not iterate more than once. Subsequent paths obtained by changing the outcome of one of the conditions in any paths.



Edges = 4
Nodes = 5
V(G) = 1

Basis path = 1
Start, 1, 2, 3 , End



Edges = 6
Nodes = 6
V(G) = 2

Basis path = 2
Start, 1, 2 ,4, end
Start, 1, 3 ,4, end

Basis Paths

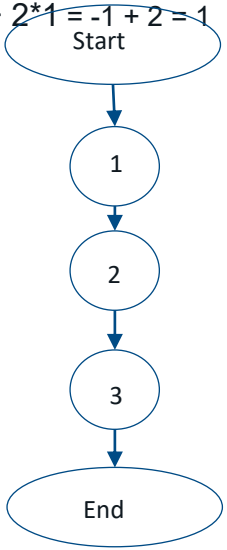
Calculating the cyclomatic complexity

Cyclomatic complexity = V(G) = e – n + 2p

E = edges, n = nodes, p = the no of connected components in G

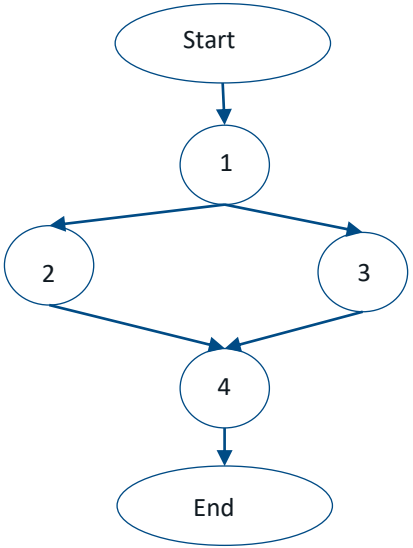
Diagram one

- 4 – 5 + 2*1 = -1 + 2 = 1



Edges = 4
Nodes = 5
V(G) = 1

Basis path =
Start, 1, 2, 3 , End



Edges = 6
Nodes = 6
V(G) = 2

Basis path =
Start, 1, 2 ,4, end
Start, 1, 3 ,4, end

School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 2d: Strings/ Languages/ Reg Ex

CSI3105:
Software Testing

String, Languages and Regular Expressions

1. Read the material from the slides (from the textbook) on BB as well as pages 98 and 99 of the textbook.

String – Slides from the textbook

Strings play an important role in testing. A string serves as a test input.

- Examples: 1011; AaBc; “Hello world”.

Alphabet

- A collection of symbols is known as an alphabet. We use an upper case letter such as X and Y to denote alphabets.
- Though alphabets can be infinite, we are concerned only with finite alphabets. For example, X={0, 1} is an alphabet consisting of two symbols 0 and 1. Another alphabet is Y={dog, cat, horse, lion}that consists of four symbols ``dog", ``cat", ``horse", and ``lion".

A string over an alphabet X is any sequence of zero or more symbols that belong to X. For example, 0110 is a string over the alphabet {0, 1}. Also, dog cat dog dog lion is a string over the alphabet {dog, cat, horse, lion}.

We will use lower case letters such as p, q, r to denote strings. The length of a string is the number of symbols in that string.

- Given a string s, we denote its length by |s|. Thus, |1011|=4 and |dog cat dog|=3. A string of length 0, also known as an empty string, is denoted by ε.

String – Slides from the textbook

String Concatenation

- Let s1 and s2 be two strings over alphabet X. We write s1.s2 to denote the concatenation of strings s1 and s2.
- For example, given the alphabet X={0, 1}, and two strings 011 and 101 over X, we obtain 011.101=011101. It is easy to see that |s1.s2|=|s1|+|s2|. Also, for any string s, we have s. ε =s and ε.s=s.

Languages - Slides from the textbook

A set L of strings over an alphabet X is known as a language. A language can be finite or infinite.

For example, a set of all strings consisting of zeros and ones is the language of binary numbers.

- The following sets are finite languages over the binary alphabet {0, 1}:
  - ∅: The empty set
  - {ε}: A language consisting only of one string of length zero
  - {00, 11, 0101}: A language containing three strings

Regular Expressions - Slides from the textbook

A pattern of special characters (called metacharacters) which is used to match strings in a search. Any non-meta character matches itself

Metacharacter	Matches…
.	**Any one character, except new line**
[a-z]	**Any one of the enclosed characters (e.g. a-z)**
*	**Zero or more of preceding character**
?	**Zero or one of the preceding characters**
+	**One or more of the preceding characters**

Examples:

- /a|x / -- matches a, x, aa, aax, aaxx, aaa, xx, etc
  (Note that | -- metacharacter for logical OR)
- abc* --- ab, abc, abcc, abccc ……
- (abc)* -- “ ”, abc, abcabc, abcabcabc, ….
- (a|b)(c|d) -- ac, ad, bc, bd
- (0|1)* --- all binary strings

Regular Expressions - Metacharacter

RE Metacharacter	Matches…
^	**beginning of line**
$	**end of line**
char	**Escape the meaning of *char* following it**
[^]	**One character not in the set**
\<	**Beginning of word anchor**
\>	**End of word anchor**
( ) or \(\)	**Tags matched characters to be used later (max = 9)**
| or |	**Or grouping**
x\{m\}	**Repetition of character x, m times (x,m = integer)**
x\{m,\}	**Repetition of character x, at least m times**
x\{m,n\}	**Repetition of character x between m and m times**

Review Questions

1. Explain the following terms associated with Control Flow Graphs:
   - Nodes
   - Directed Edges
   - Basic Block
   - Path
2. Draw the Control Flow Graph for the following code segments:

```
Q= 15;
 P = 5;
   while  (Q > P)
 {
                P = R (Q, P);
                Q = Q – 1;
 }
```

```
for ( y = 100; y > x; x--)
{
    y = y – 5;
    x = x + 1;
}
```

3. Assume a predicate P(x) that represents the statement:
   - X is a prime number

What is the truth values for the following:

P(2)

(P3)

P(4)

P(5)

References

- A. P. Mathur, *Foundations of Software Testing, 2nd Edition*, Pearson Education, 2014.

P. Ammann & J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.