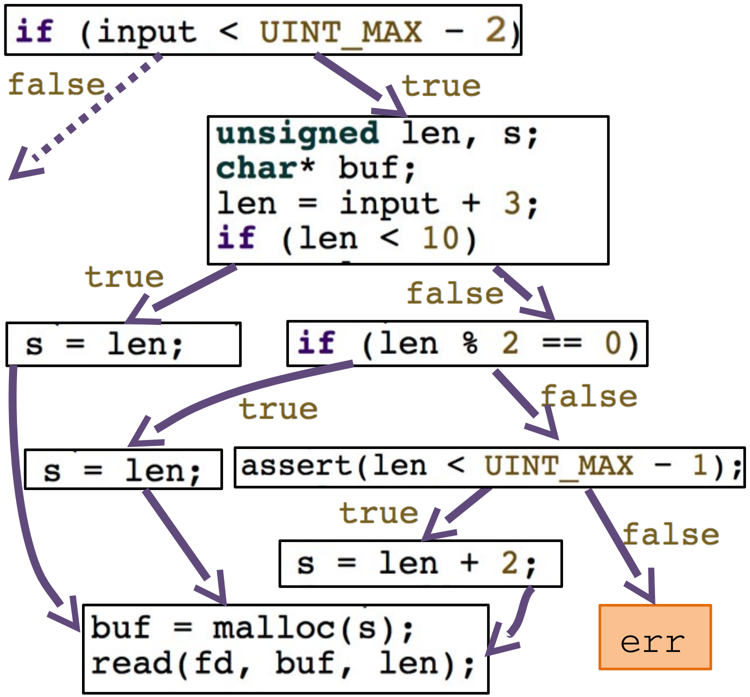
Vulnerability Analysis (IV): Program Verification

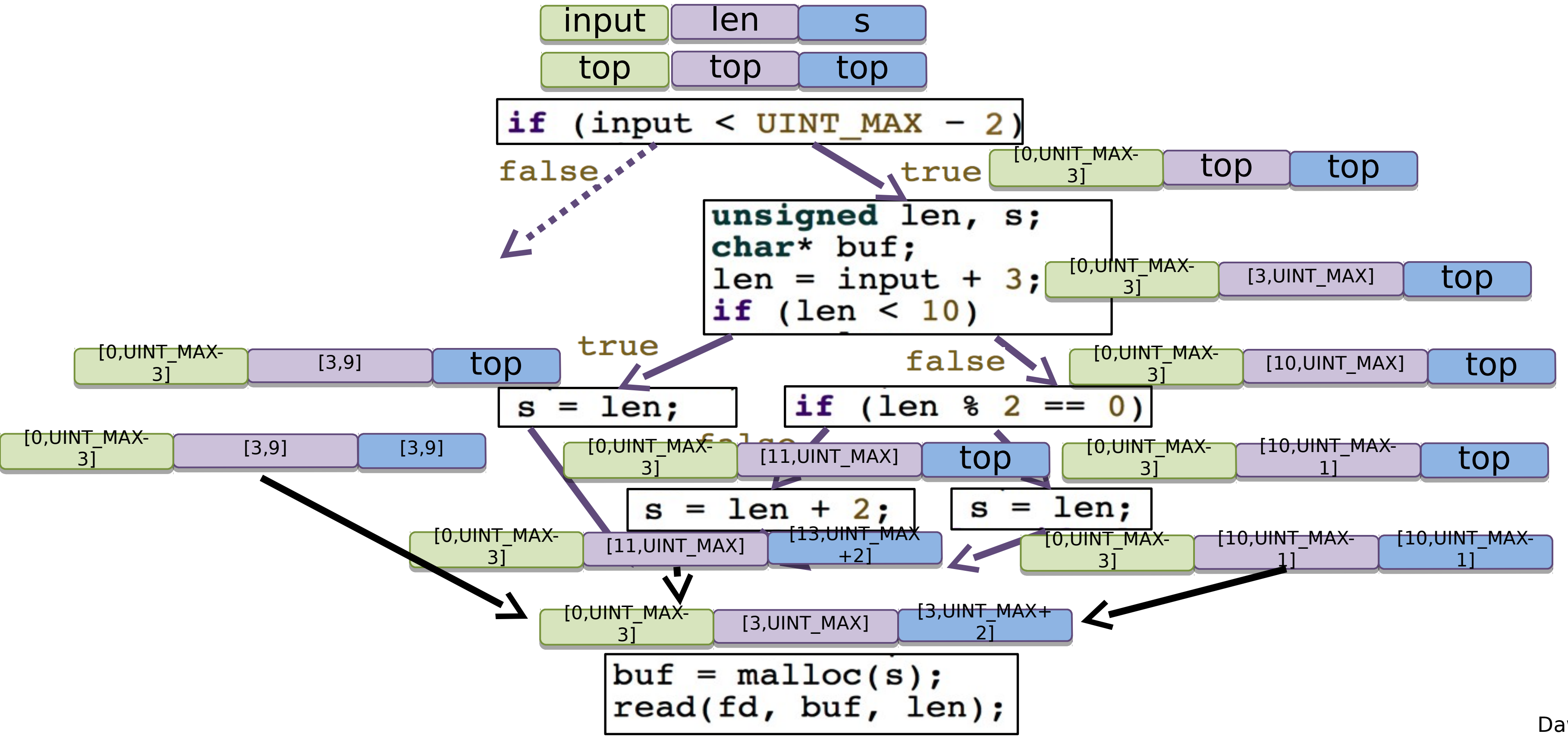
Slide credit: Vijay D'Silva

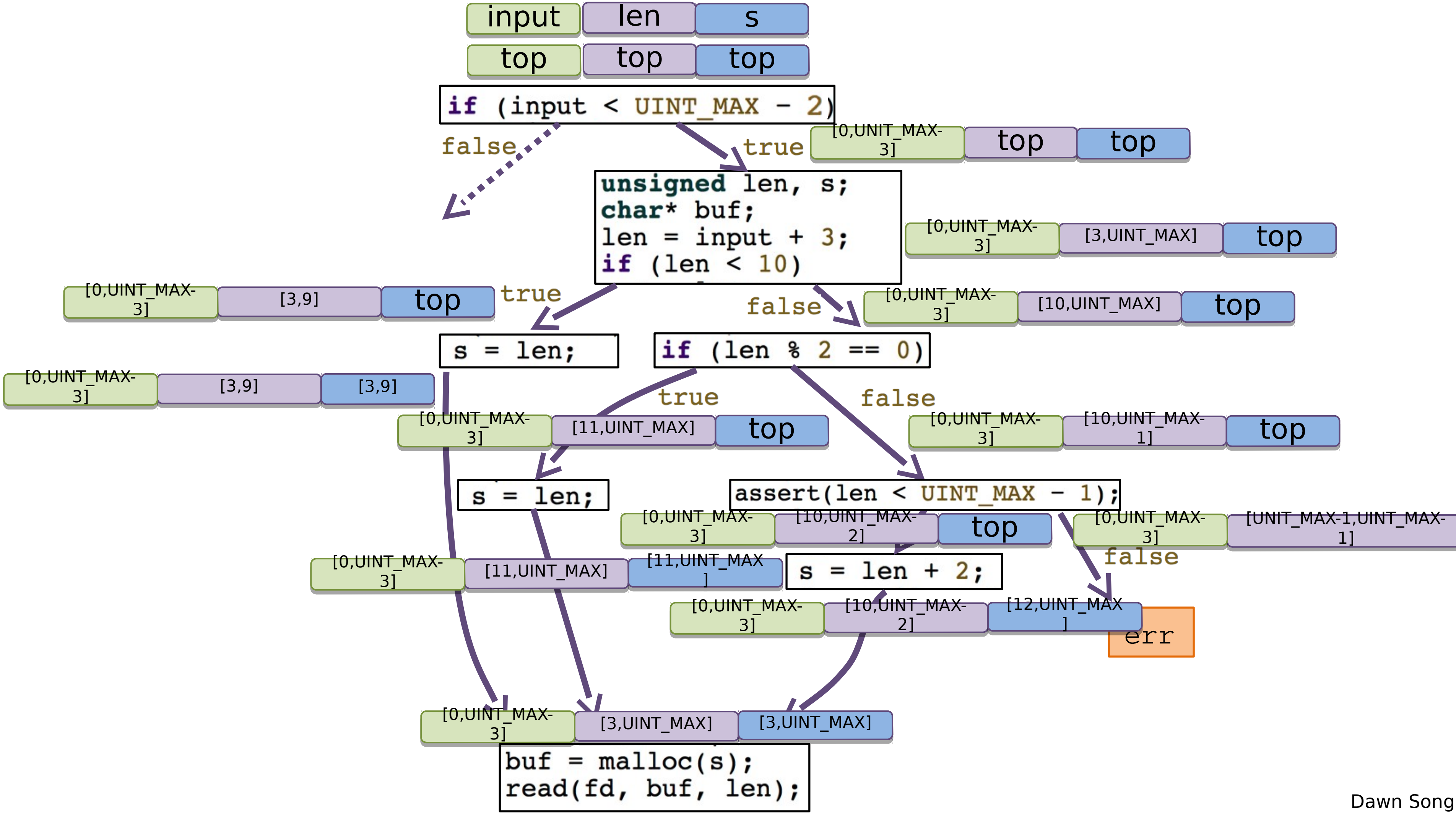
Interval Analysis: Example

```
foo(unsigned input){

  if (input < UINT_MAX - 2){
    unsigned len, s;
    char* buf;
    len = input + 3;
    if (len < 10)
      s = len;
    else if (len % 2 == 0)
      s = len;
    else {
      assert(len < UINT_MAX - 1);
      s = len + 2;
    }
    buf = malloc(s);
    read(fd, buf, len);
     ....
  }
}
```

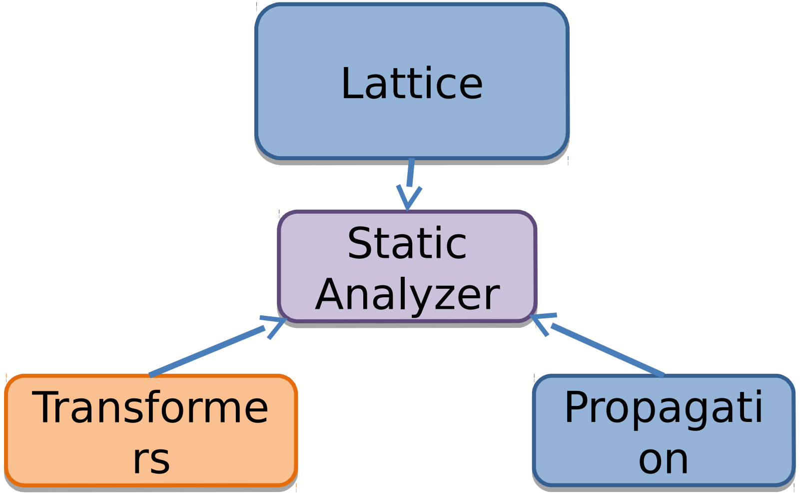


Interval Analysis: Example




input
len
s
top
top
top
if (input < UINT_MAX - 2)
false
true
[0,UNIT_MAX-3]
top
top
unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
[0,UINT_MAX-3]
[3,UINT_MAX]
top
true
[0,UINT_MAX-3]
[3,9]
top
false
[0,UINT_MAX-3]
[10,UINT_MAX]
top
s = len;
if (len % 2 == 0)
[0,UINT_MAX-3]
[3,9]
[3,9]
true
false
[0,UINT_MAX-3]
[11,UINT_MAX]
top
[0,UINT_MAX-3]
[10,UINT_MAX-1]
top
s = len;
assert(len < UINT_MAX - 1);
[0,UINT_MAX-3]
[10,UINT_MAX-2]
top
[0,UINT_MAX-3]
[UNIT_MAX-1,UINT_MAX-1]
false
[0,UINT_MAX-3]
[11,UINT_MAX]
[11,UINT_MAX]
s = len + 2;
[0,UINT_MAX-3]
[10,UINT_MAX-2]
[12,UINT_MAX]
err
[0,UINT_MAX-3]
[3,UINT_MAX]
[3,UINT_MAX]
buf = malloc(s);
read(fd, buf, len);

Transformers in a Static Analyzer



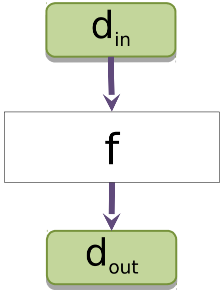
A *transformer* (*or transfer function*) is

- a function on a lattice
- that respects the order (monotone)

Transformers

- abstract the effect of program statements
- may lose precision

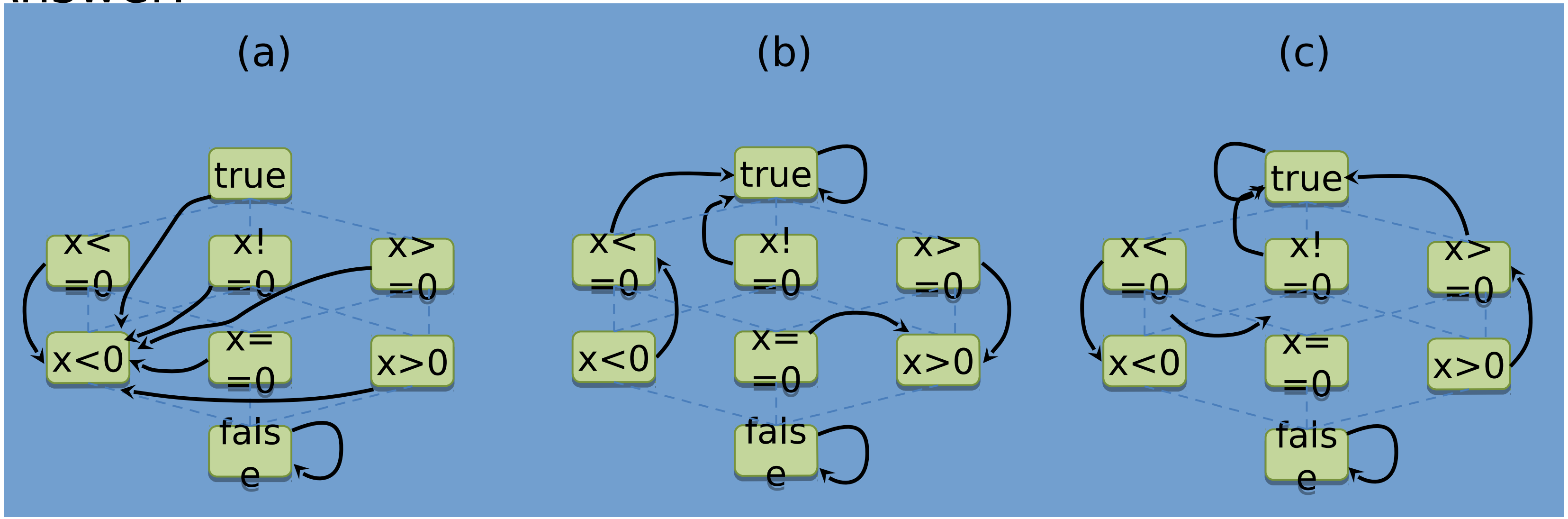
Quiz: Sign Analysis Transformers



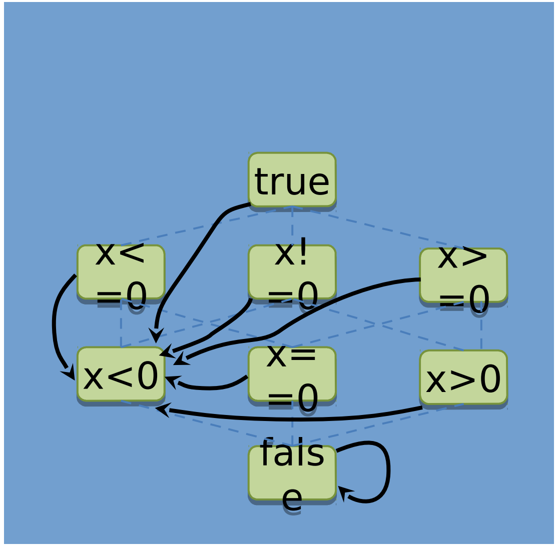
Which of the following is the right transformer for `x=x-1` ?

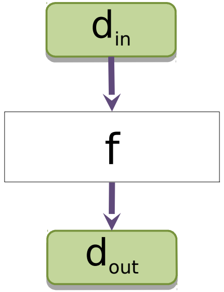
C

Answer:



Quiz: Sign Analysis Transformers





Which of the statements below is best represented by this transformer?

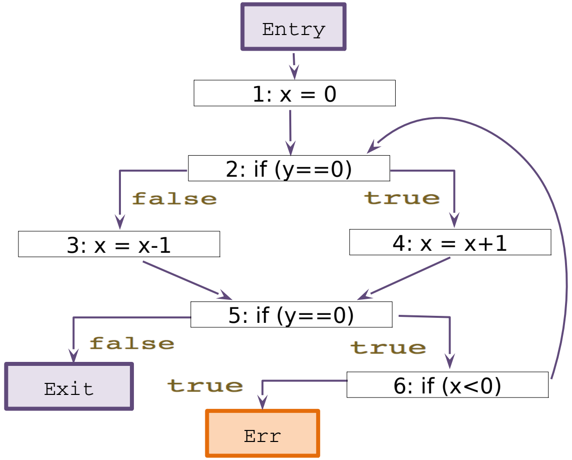
- `x=x-2`
- `if (x<-4)`
- `x=-4`
- `if (x>-4)`

Answer: x=-4

1	Analysis Frameworks

a	Lattices
b	Transformers
c	Systems of Equations
d	Solving Equations

Programs to Equations

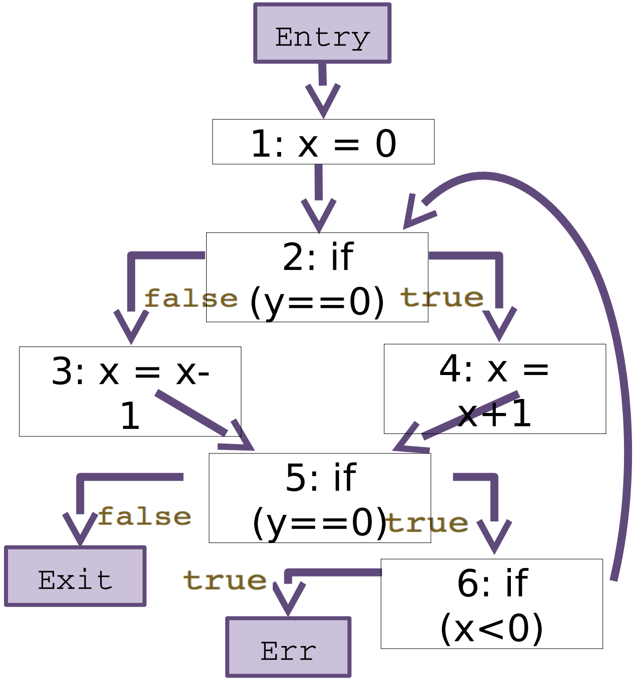


$$
\begin{aligned}
d_{out1} &= f_1(d_{in1}) \\
d_{out2\text{-}f} &= f_{2\text{-}f}(d_{out1} \sqcup d_{out6}) \\
d_{out2\text{-}t} &= f_{2\text{-}f}(d_{out1} \sqcup d_{out6}) \\
d_{Exit} &= f_{5\text{-}f}(d_{out3} \sqcup d_{out4}) \\
d_{out5\text{-}t} &= f_{5\text{-}t}(d_{out3} \sqcup d_{out4}) \\
d_{Err} &= f_{6\text{-}t}(d_{out5\text{-}t})
\end{aligned}
$$

Programs
- convenient to write
- difficult to analyze: datatypes, loops, branches, etc.

Systems of equations
- well-studied in mathematics
- simple compared to programs: expressions and equalities

Example Static Analysis Equations



$$d_{out1} = f_1(d_{Entry})$$

$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out2\text{-}t} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out3} = f_3(d_{out2\text{-}f})$$

$$d_{out4} = f_4(d_{out2\text{-}t})$$

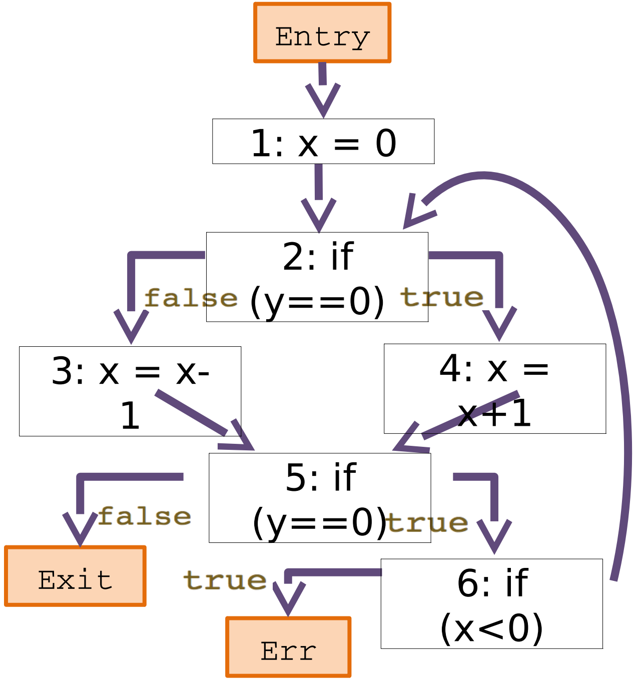
$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$

$$d_{out5\text{-}t} = f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$$

$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

$$d_{out6\text{-}f} = f_{6\text{-}f}(d_{out5\text{-}t})$$

Example Static Analysis Equations



$$d_{out1} = f_1(d_{Entry})$$

$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out2\text{-}t} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out3} = f_3(d_{out2\text{-}f})$$

$$d_{out4} = f_4(d_{out2\text{-}t})$$

$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$

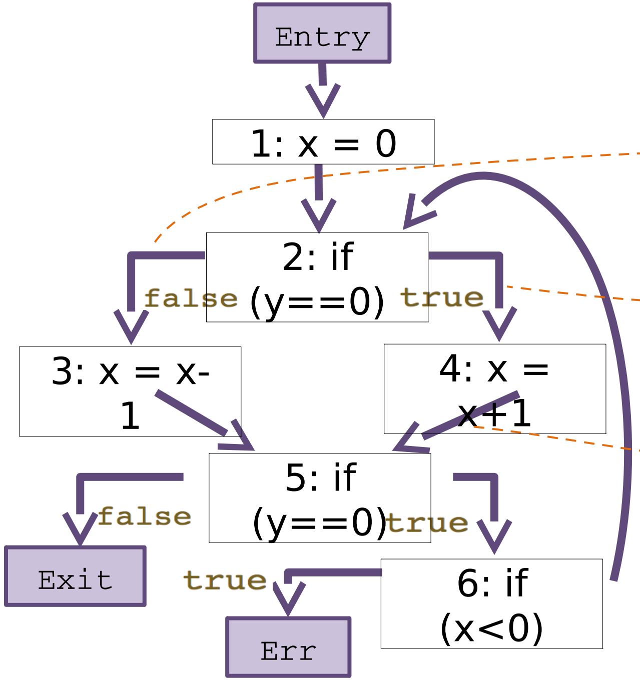
$$d_{out5\text{-}t} = f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$$

$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

$$d_{out6\text{-}f} = f_{6\text{-}f}(d_{out5\text{-}t})$$

Variables represent facts at different program points

Example Static Analysis Equations



$$d_{out1} = f_1(d_{Entry})$$

$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out2\text{-}t} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out3} = f_3(d_{out2\text{-}f})$$

$$d_{out4} = f_4(d_{out2\text{-}t})$$

$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$

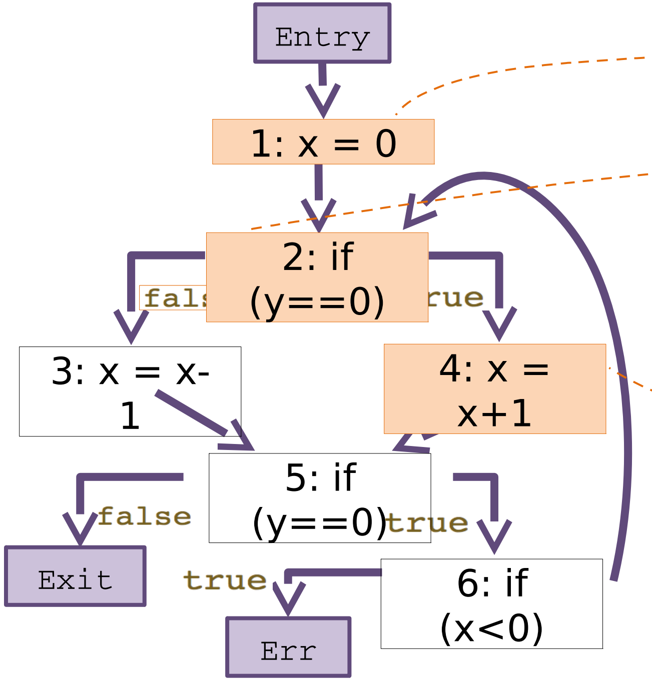
$$d_{out5\text{-}t} = f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$$

$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

$$d_{out6\text{-}f} = f_{6\text{-}f}(d_{out5\text{-}t})$$

Variables represent facts at different program points

Example Static Analysis Equations



$$d_{out1} = f_1(d_{Entry})$$

$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out2\text{-}t} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out3} = f_3(d_{out2\text{-}f})$$

$$d_{out4} = f_4(d_{out2\text{-}t})$$

$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$

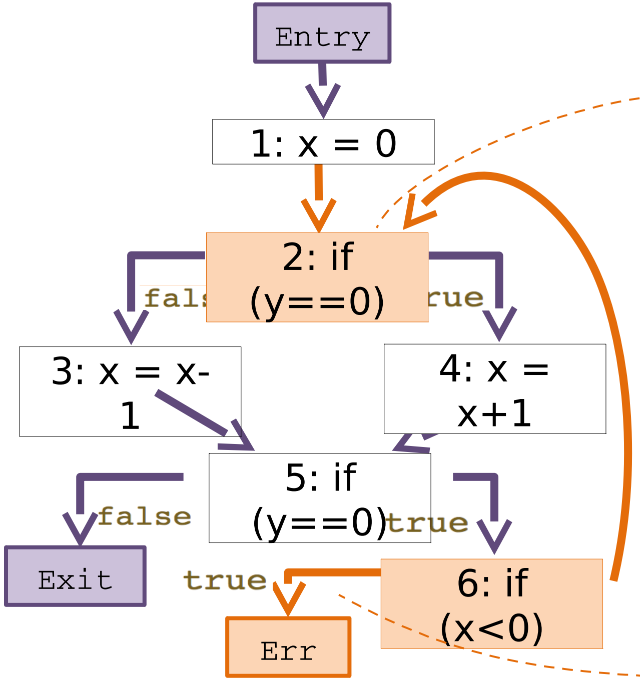
$$d_{out5\text{-}t} = f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$$

$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

$$d_{out6\text{-}f} = f_{6\text{-}f}(d_{out5\text{-}t})$$

Expressions represent how data is transformed

Example Static Analysis Equations



$$d_{out1} = f_1(d_{Entry})$$

$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out2\text{-}t} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$

$$d_{out3} = f_3(d_{out2\text{-}f})$$

$$d_{out4} = f_4(d_{out2\text{-}t})$$

$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$

$$d_{out5\text{-}t} = f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$$

$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

$$d_{out6\text{-}f} = f_{6\text{-}f}(d_{out5\text{-}t})$$

An equation relates the facts flowing in and out of a basic block

Static Analysis Equations

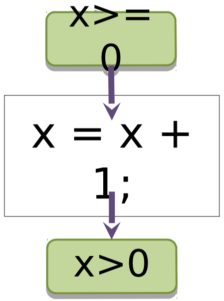
A *static analysis equation* is a set of equalities of the form

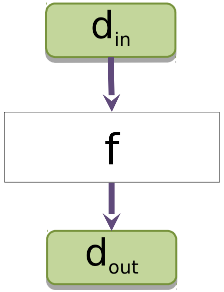
d_1	=	$exp_1(d_1, \ldots, d_k)$
…	=	…
d_k	=	$exp_k(d_1, \ldots, d_k)$

- *variables* d_i represent facts flowing in and out of basic blocks
- *expressions* $exp_i(d_1, \ldots, d_k)$
  - describe how data is transformed
  - are composed of variables, transfer functions, meet, join

Equations for a Single Statement

The relationship between facts that are true at different points in a program can be encoded as an equation.

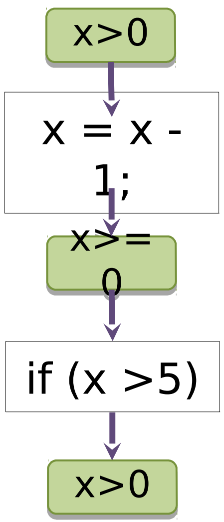


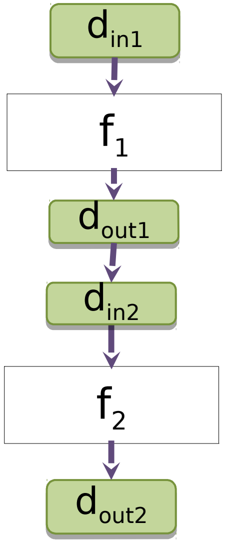


$$d_{out} = f(d_{in})$$

Equations for Sequential Composition

Sequential composition applies the function in one equation to the result of a previous equation





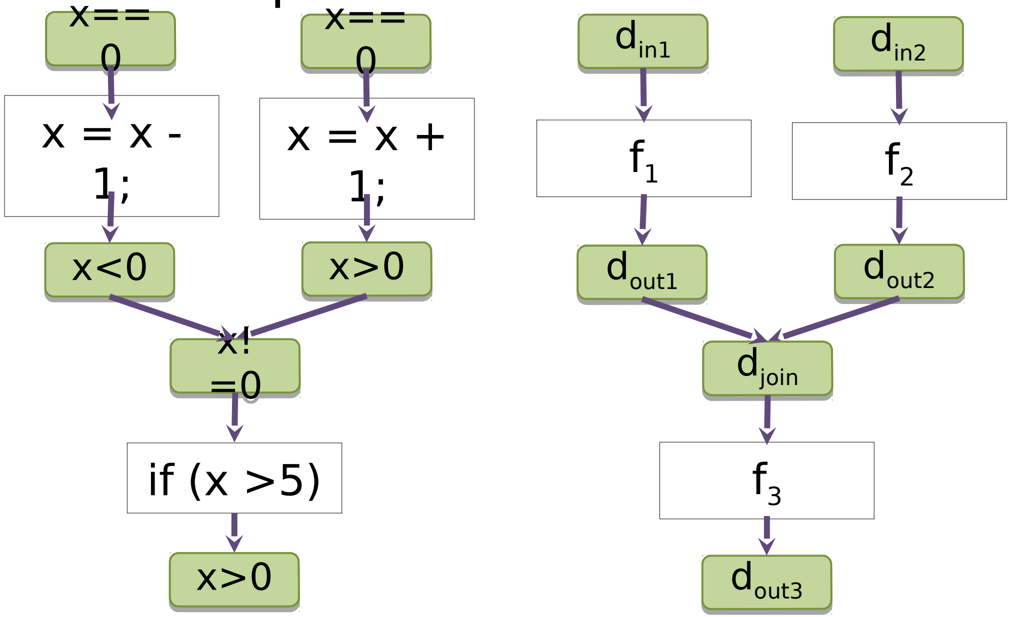
$$d_{out1} = f_1(d_{in1})$$

$$d_{in2} = d_{out1}$$

$$d_{out2} = f_2(d_{in2})$$

Equations at Join Points

The relationship between facts that are true at different points in a program can be encoded as an equation.



$$d_{out1} = f_1(d_{in1})$$

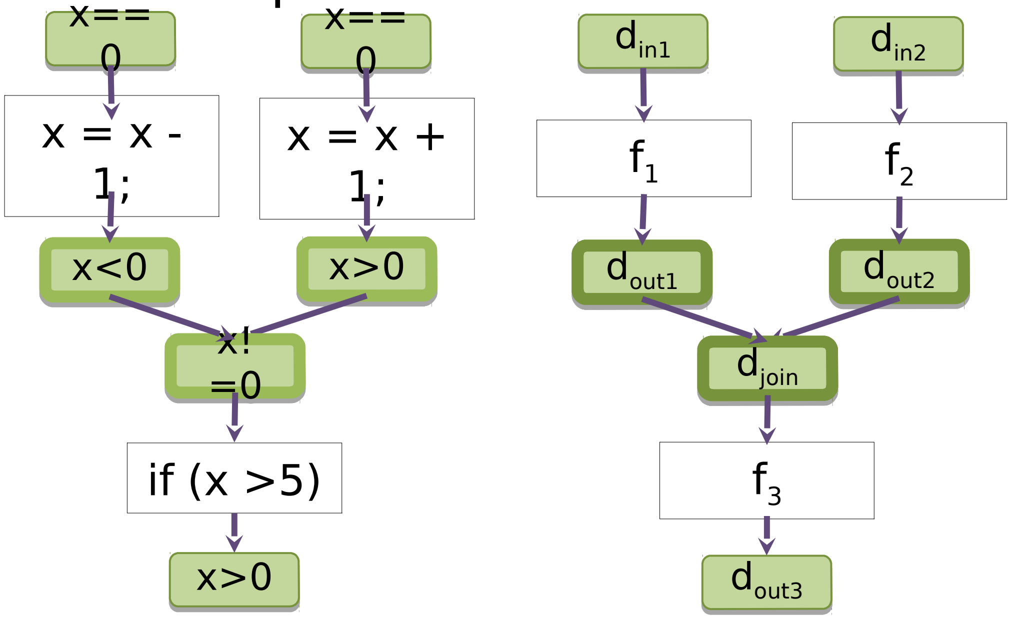
$$d_{out2} = f_2(d_{in2})$$

$$d_{join} = d_{out1} \sqcup d_{out2}$$

$$d_{out3} = f_3(d_{join})$$

Equations at Join Points

The relationship between facts that are true at different points in a program can be encoded as an equation.



$$d_{out1} = f_1(d_{in1})$$

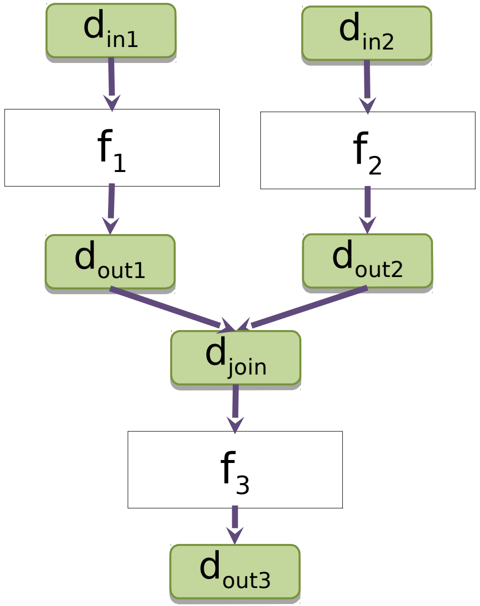
$$d_{out2} = f_2(d_{in2})$$

$$\boldsymbol{d_{join}} = \boldsymbol{d_{out1}} \sqcup \boldsymbol{d_{out2}}$$

$$d_{out3} = f_3(d_{join})$$

Simplifying Equations

It is common to simplify equations by eliminating variables related by equalities.



$$
\begin{aligned}
d_{out1} &= f_1(d_{in1}) \\
d_{out2} &= f_2(d_{in2}) \\
d_{join} &= d_{out1} \sqcup d_{out2} \\
d_{out3} &= f_3(d_{join})
\end{aligned}
$$

$$
\begin{aligned}
d_{out1} &= f_1(d_{in1}) \\
d_{out2} &= f_2(d_{in2}) \\
d_{out3} &= f_3(d_{out1} \sqcup d_{out2})
\end{aligned}
$$

Why Equations?

$$x = \tfrac{1}{2} y - z$$
$$y = x + 2z + 1$$
$$z = 3x + 2y - 1$$

Basic Algebra

$$d_{out1} = f_1(d_{in1})$$
$$d_{out2\text{-}f} = f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$$
$$d_{Exit} = f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$$
$$d_{Err} = f_{6\text{-}t}(d_{out5\text{-}t})$$

Program Analysis Equations

Several properties of equations are well studied

- Existence of solutions
- How to compute solutions when they exist
- How to approximate solutions if finding exact solutions is too difficult

By using equations, program analysis reduces to a well known problem and existing intuition and techniques can be applied

1	Analysis Frameworks

a	Lattices
b	Transformers
c	Systems of Equations
d	Solving Equations

Solutions to Equations

x_1	=	$exp_1(x_1, \dots, x_k)$
…	=	…
x_k	=	$exp_k(x_1, \dots, x_k)$

A *solution* to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?

d_{out1}	=	$f_1(d_{Entry})$
$d_{out2\text{-}f}$	=	$f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$
$d_{out2\text{-}t}$	=	$f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$
d_{out3}	=	$f_3(d_{out2\text{-}f})$
d_{out4}	=	$f_4(d_{out2\text{-}t})$
d_{Exit}	=	$f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$
$d_{out5\text{-}t}$	=	$f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$
d_{Err}	=	$f_{6\text{-}t}(d_{out5\text{-}t})$
$d_{out6\text{-}f}$	=	$f_{6\text{-}f}(d_{out5\text{-}t})$

Solutions to Equations

$\mathbf{x_1}$	=	$\mathbf{exp_1(x_1, \ldots, x_k)}$
…	=	…
x_k	=	$exp_k\ (x_1, \ldots, x_k)$

A *solution* to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?

$\mathbf{d_{out1}}$	=	$\mathbf{f_1(d_{Entry})}$
$d_{out2\text{-}f}$	=	$f_{2\text{-}f}(d_{out1} \sqcup d_{out6})$
$d_{out2\text{-}t}$	=	$f_{2\text{-}f}\ (d_{out1} \sqcup d_{out6})$
d_{out3}	=	$f_3(d_{out2\text{-}f})$
d_{out4}	=	$f_4(d_{out2\text{-}t})$
d_{Exit}	=	$f_{5\text{-}f}(d_{out3} \sqcup d_{out4})$
$d_{out5\text{-}t}$	=	$f_{5\text{-}t}(d_{out3} \sqcup d_{out4})$
d_{Err}	=	$f_{6\text{-}t}(d_{out5\text{-}t})$
$d_{out6\text{-}f}$	=	$f_{6\text{-}f}(d_{out5\text{-}t})$

The Fixed Point Theorem

x_1	=	$exp_1(x_1, \ldots, x_k)$
…	=	…
x_k	=	$exp_k(x_1, \ldots, x_k)$

A *solution* to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?

A *fixed point* of a function is an element satisfying

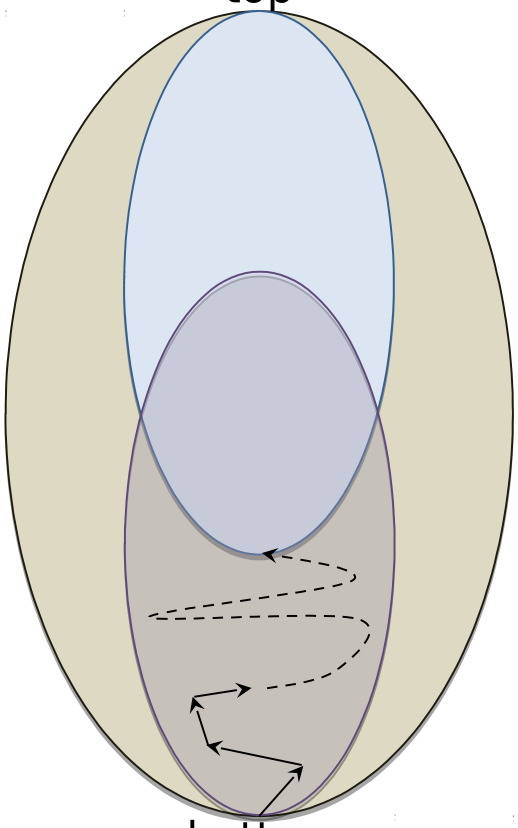
$$x = exp(x)$$

This is an equation and a fixed point is a solution to an equation.

$x = (x_1, x_2, \ldots, x_k) =_{e.g.} (d_{out1}, d_{out2f}, \ldots)$

$exp = (exp_1, \ldots, exp_k) =_{e.g.} (f_1, f_{2f}, \ldots)$

How to Solve Equations



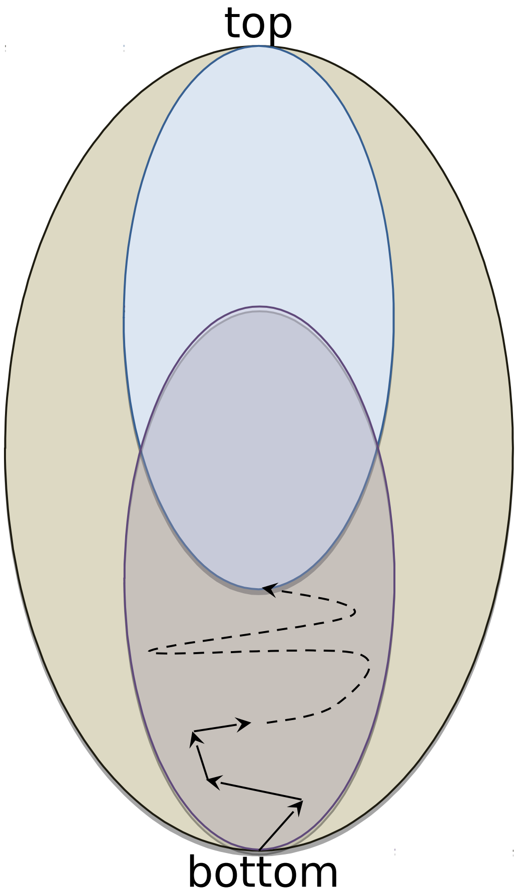
Solving equations by iteration:

- Start from least element
- Apply transformers once: exp(x)
- Update all variables
- Apply transformers again: exp(exp(x))
- Repeat until no variables change

Issues

- wasteful updates to variables
- termination of the iteration
- termination in *reasonable time*

Iteration Strategies



Round robin	Update equations in an apriori fixed order
Topological order	Update equations following the structure of the CFG
Chaotic Iteration	Update equations in arbitrary order making sure all are eventually updated

Many more advanced strategies exist.

Properties of Programs

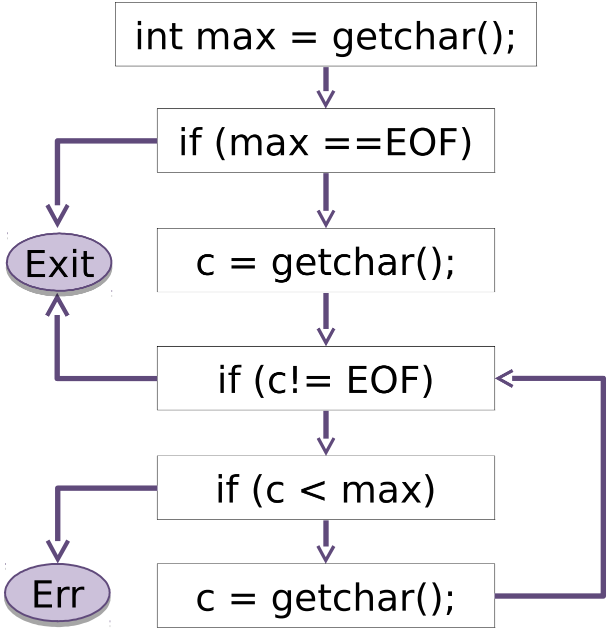
```
int
max=getchar();
if (max == EOF)
  exit(0);
c = getchar();
while (c != EOF)
{
  assert(c <
max);
  c= getchar();
}
```

Consider this program. Some questions that we can ask a program analyzer are:

- Is it possible to violate the assertion?
- What sequence of inputs leads to an assertion violation?

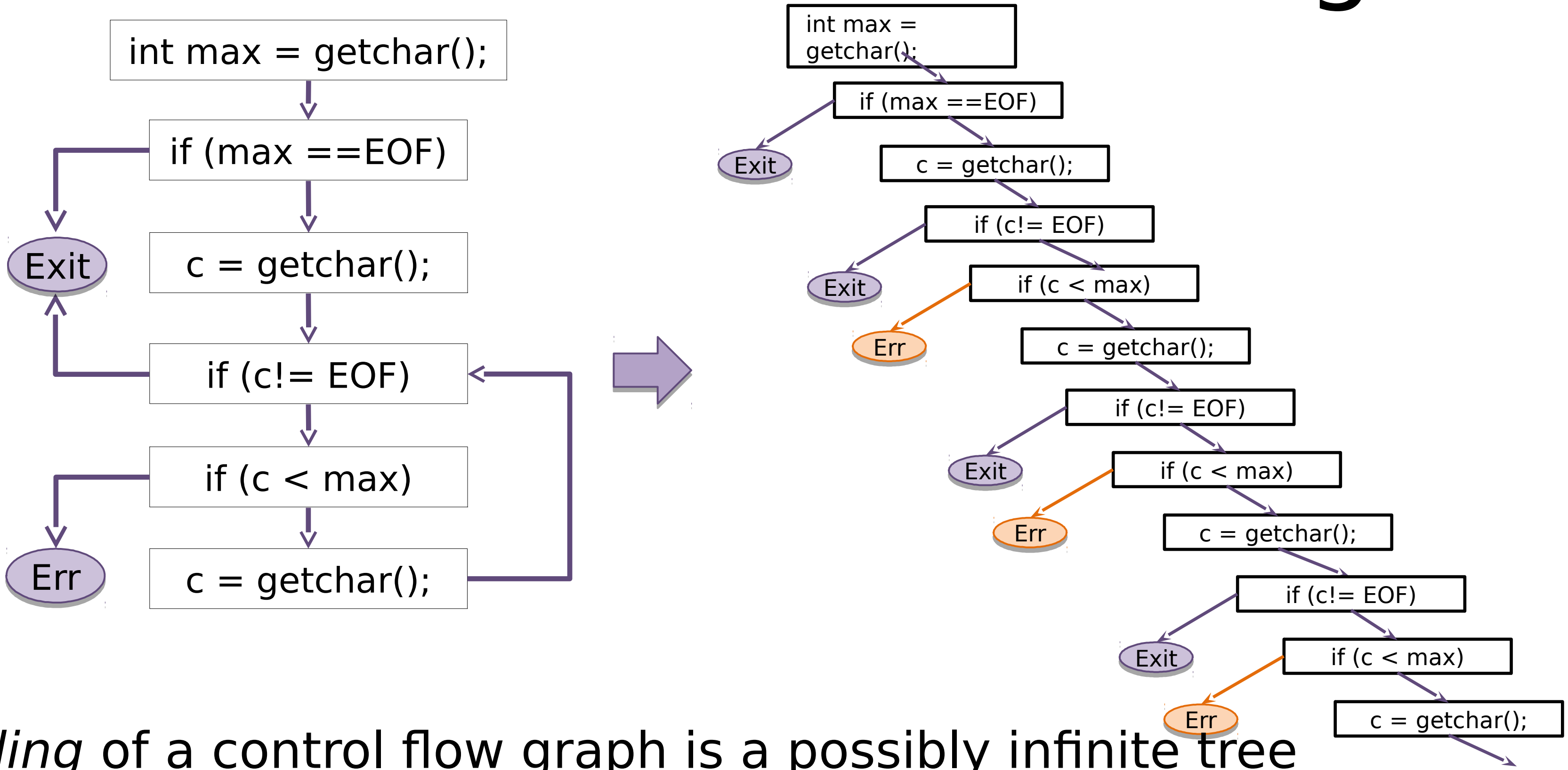
Programs and Control Flow Graphs

```
int
max=getchar();
if (max == EOF)
  exit(0);
c = getchar();
while (c != EOF)
{
  assert(c <
max);
  c= getchar();
}
```



Control Flow Graphs are representations of programs used in program analyzers. The graph structure makes control flow in a program explicit.

Control Flow Unwinding



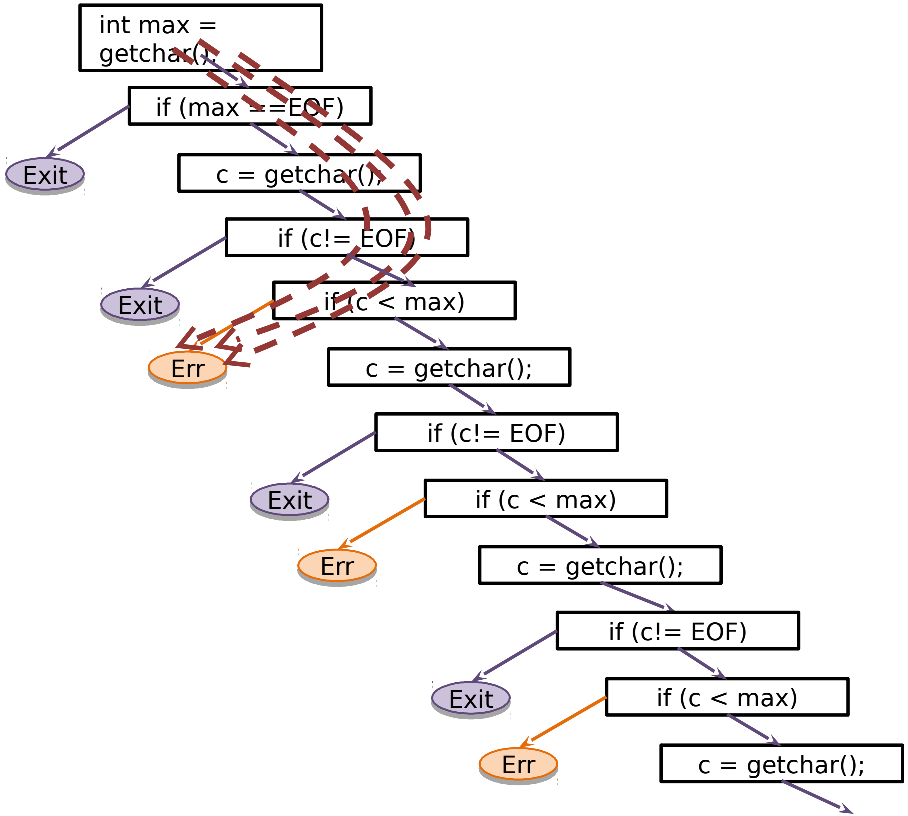
unwinding of a control flow graph is a possibly infinite tree taining every path in the graph.

Executions

An execution corresponds to a path in the tree unwinding.

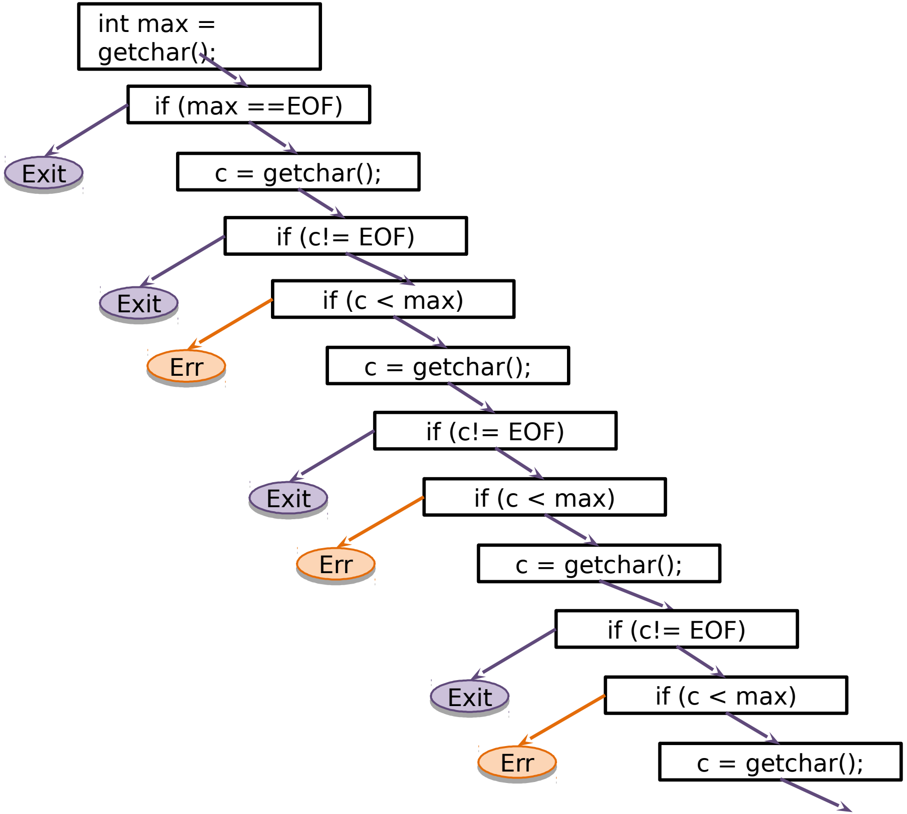
Multiple executions can traverse the same path.

A path is *feasible* if there is an execution that traverses it.



Assertion Violations

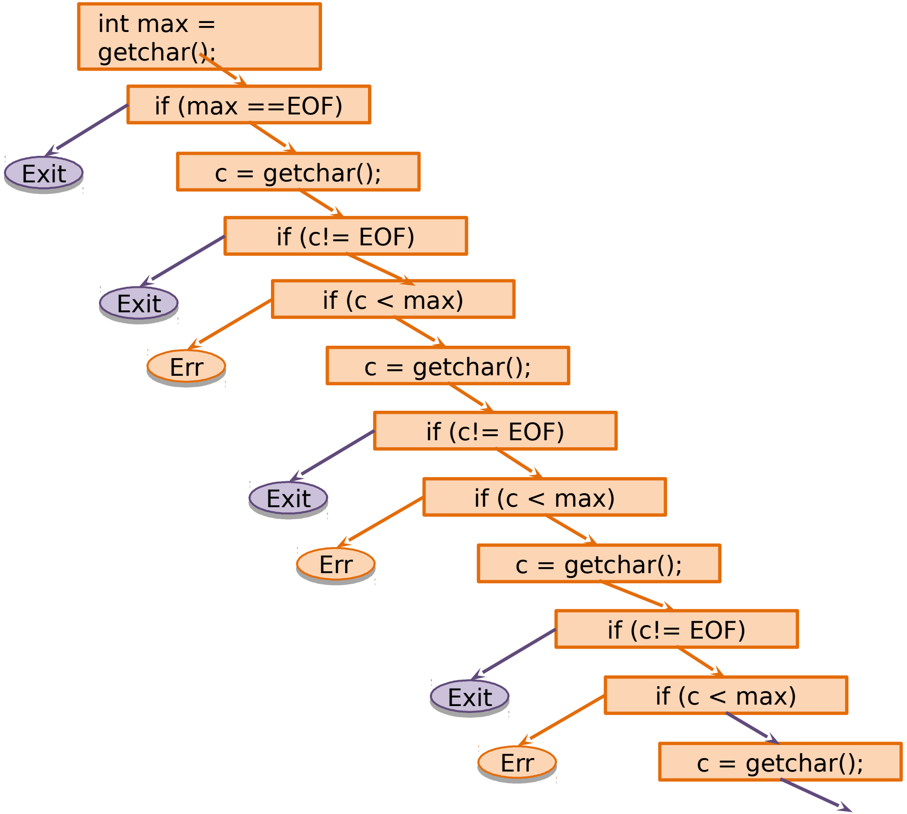
The question of whether an assertion violation exists is equivalent to asking if one of the paths to an error location is feasible.



Assertion Violations

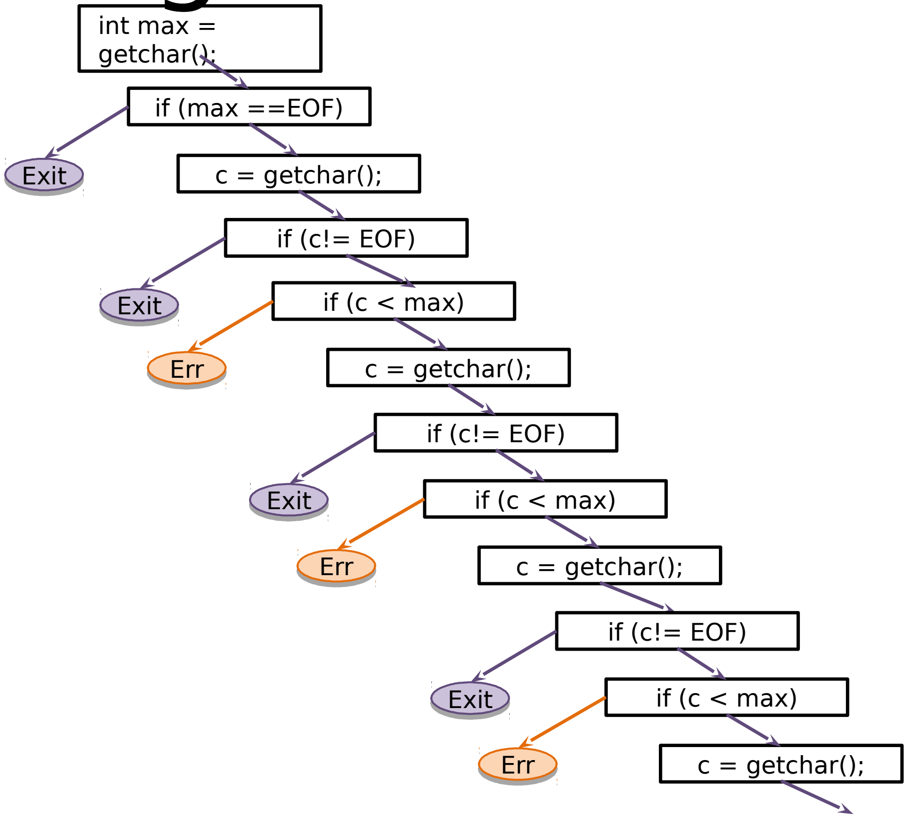
The question of whether an assertion violation exists is equivalent to asking if one of the paths to an error location is feasible.

Vulnerability detection techniques attempt to find if one such feasible path exists.



Fuzzing

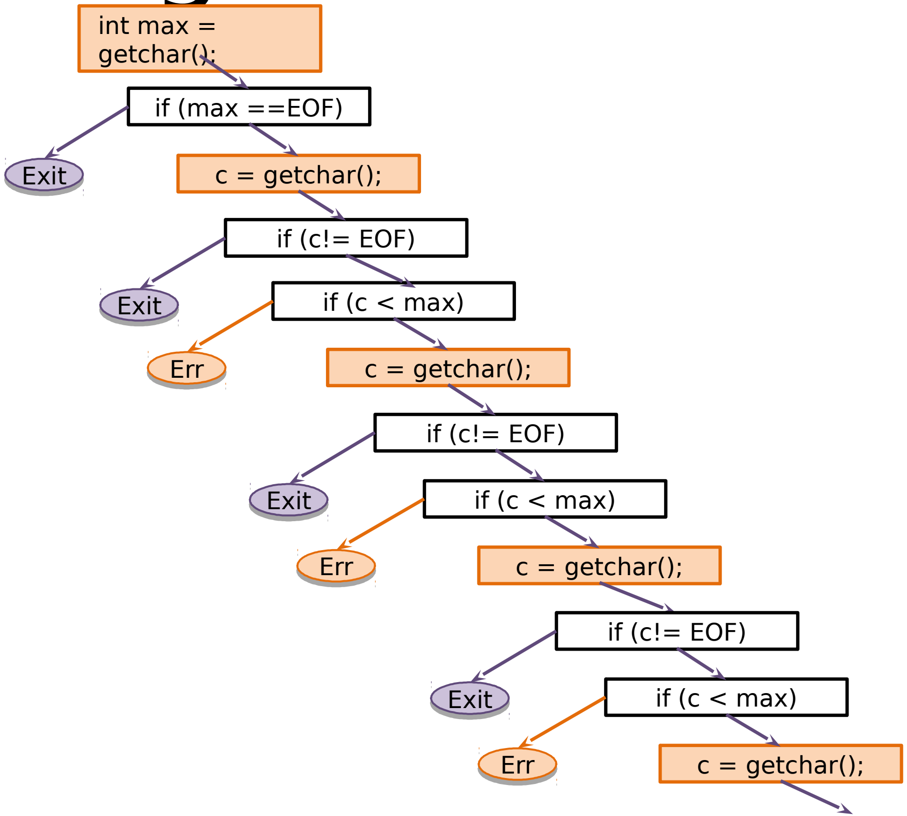
Fuzzing techniques feed inputs to the system and try to trigger a crash.



Fuzzing

Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing

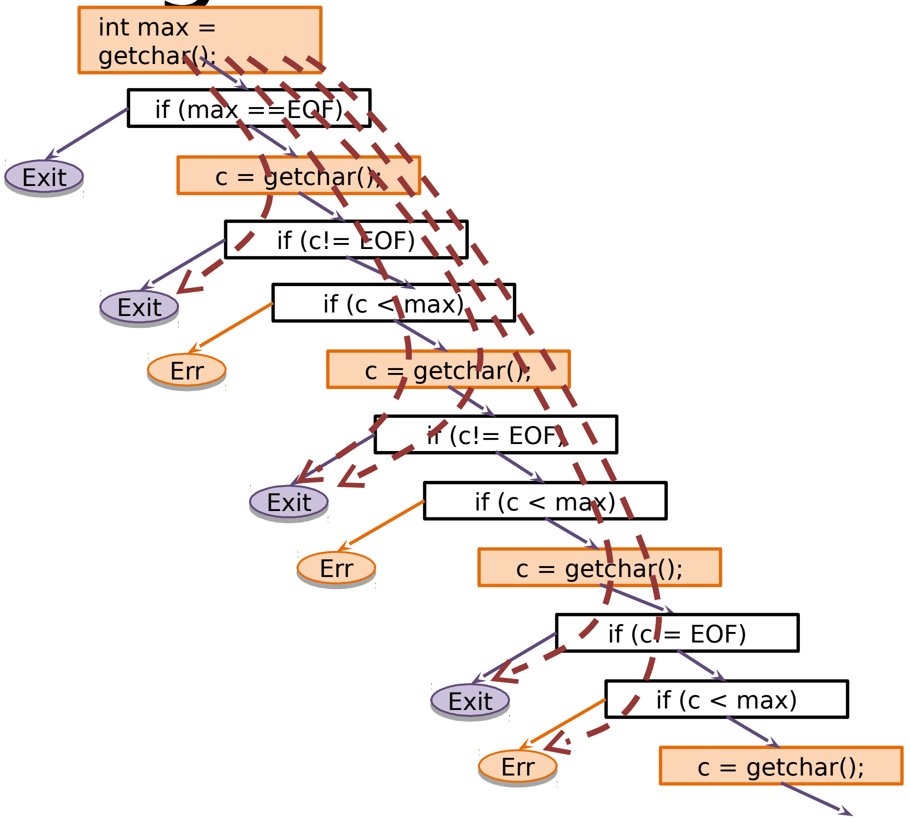
- How to generate inputs?
- How to feed inputs to the system?



Fuzzing

Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing

- How to generate inputs?
- How to feed inputs to the system?

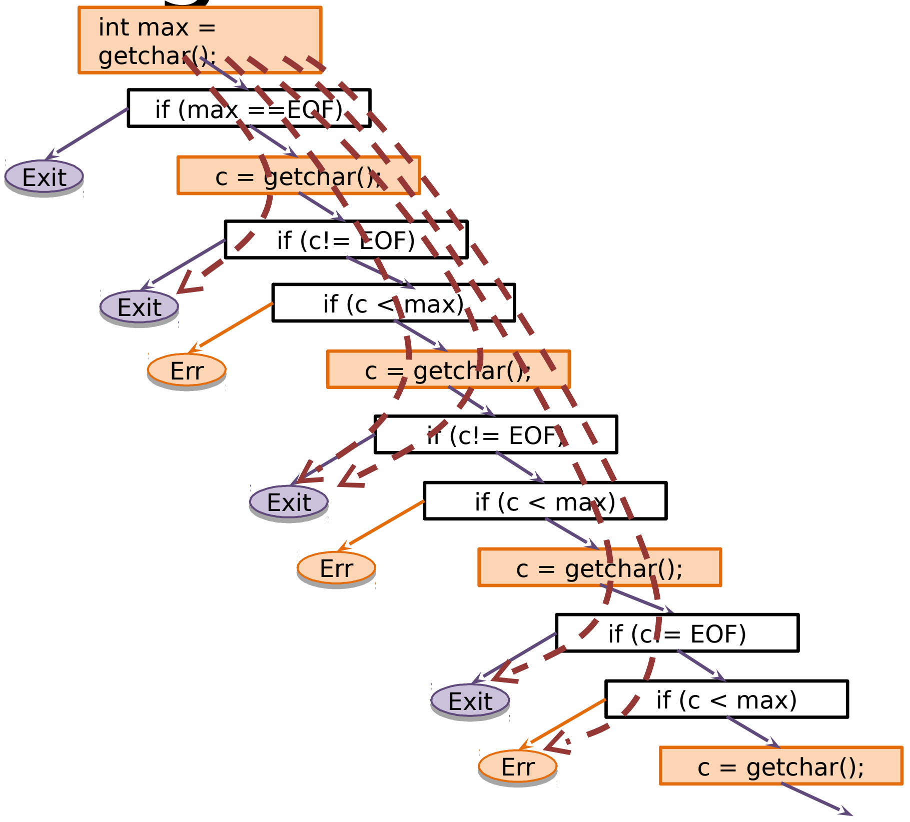


Fuzzing

Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing

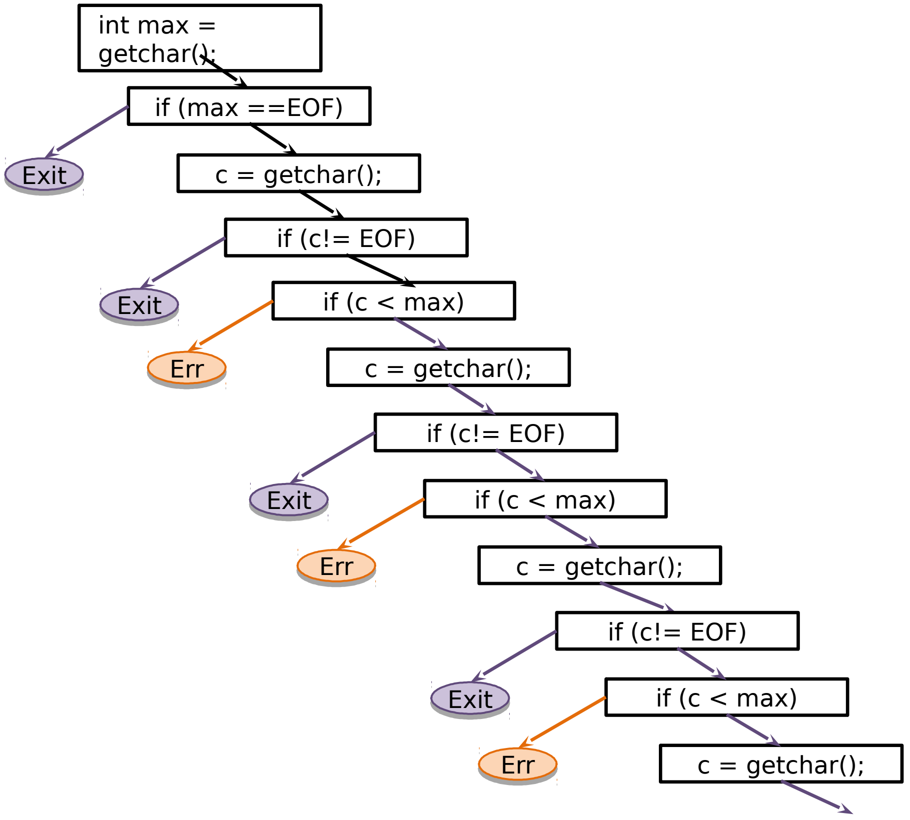
- How to generate inputs?
- How to feed inputs to the system?

Goal: Maximize the likelihood that a set of inputs trigger an error.



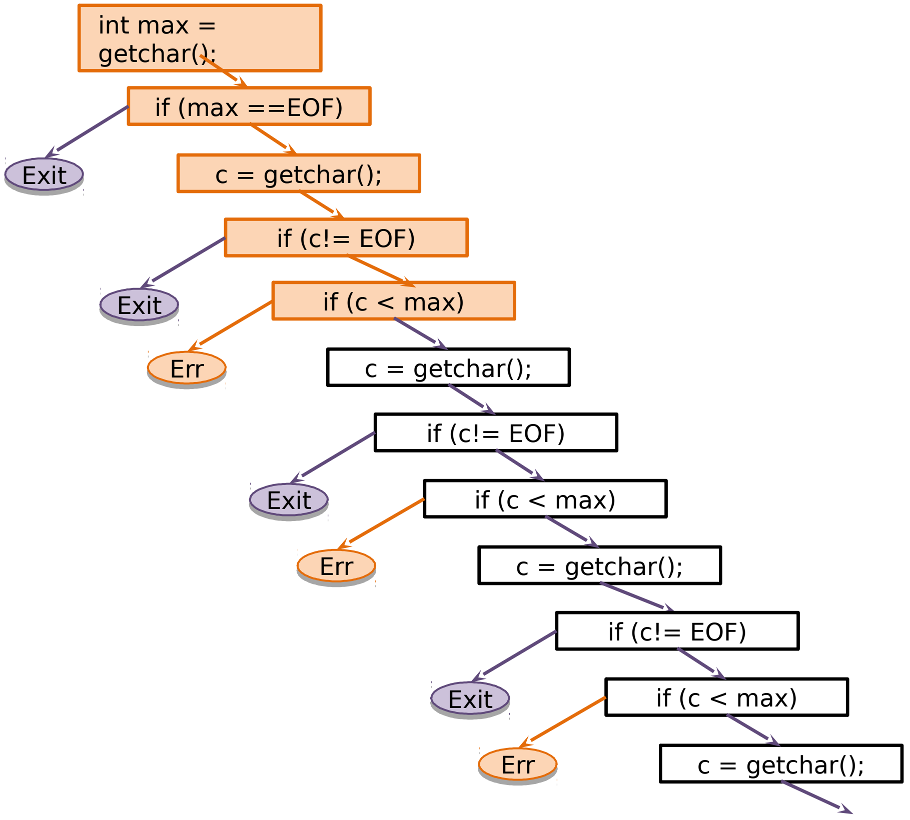
Symbolic Execution

Symbolic execution uses techniques from logic to avoid exploring the same path multiple times.



Symbolic Execution

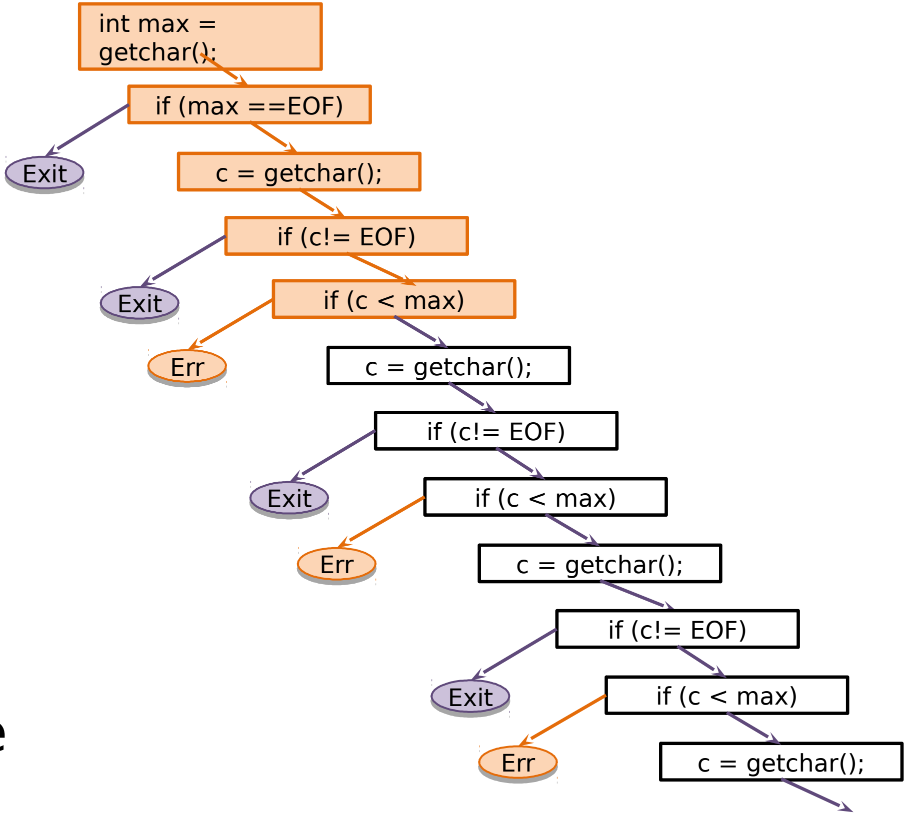
Symbolic execution uses techniques from logic to avoid exploring the same path multiple times.



Symbolic Execution

Symbolic execution uses techniques from logic to avoid exploring the same path multiple times

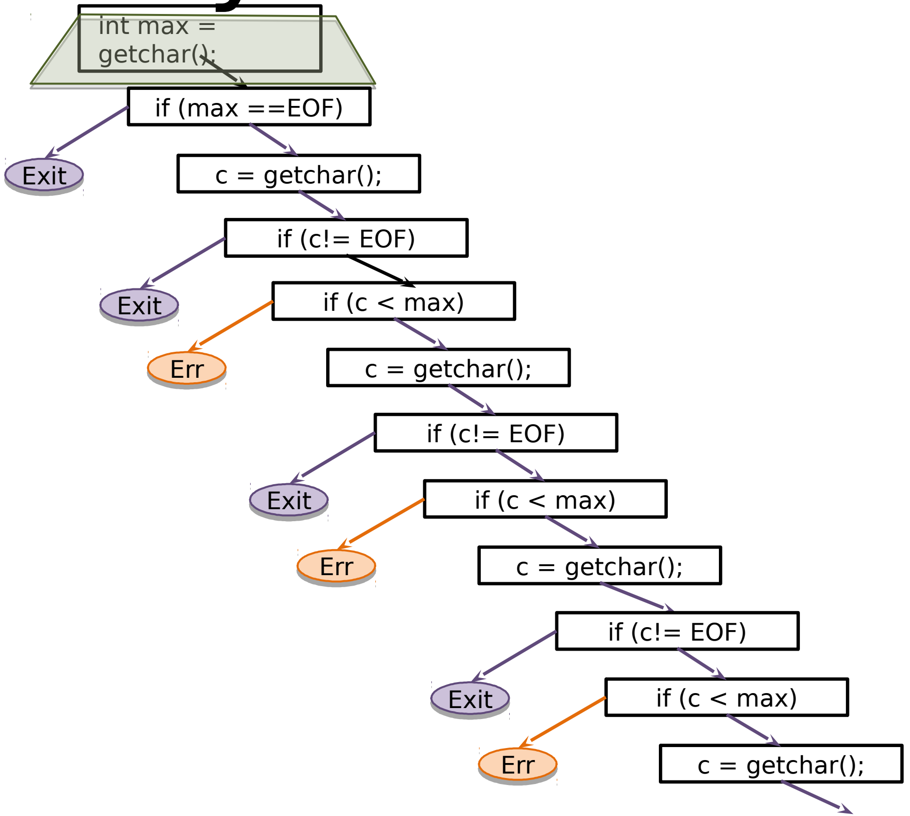
	max == getchar()
&&	max != EOF
&&	c == getchar()
&&	c != EOF
&&	c >= max



The highlighted path is feasible exactly if a certain formula is *satisfiable*.

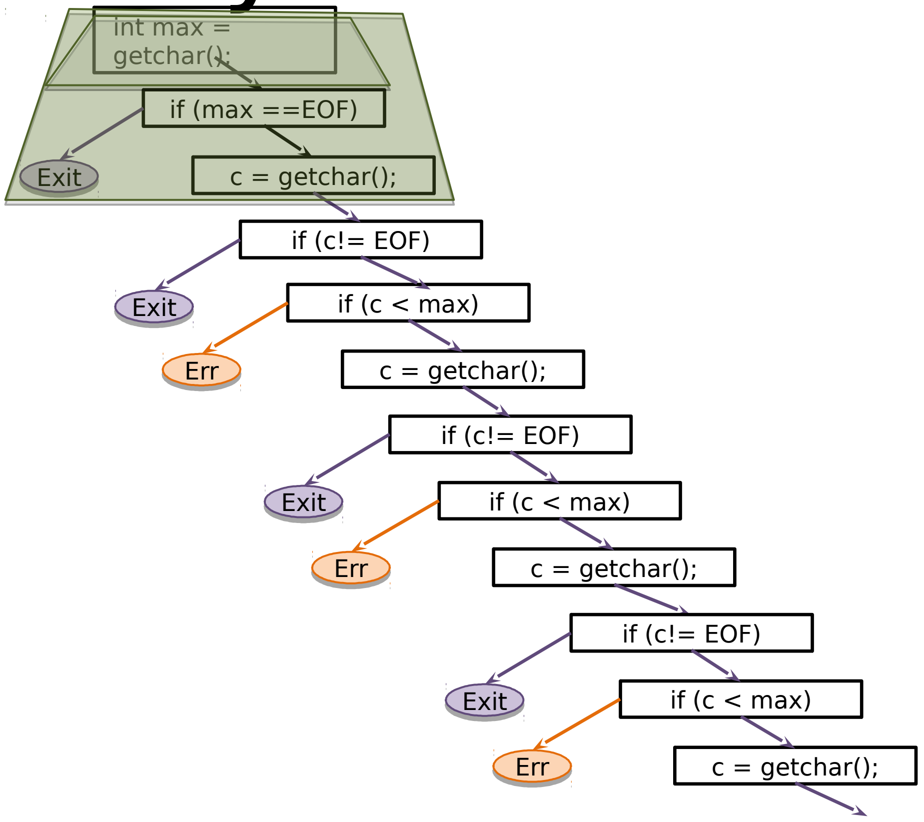
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.



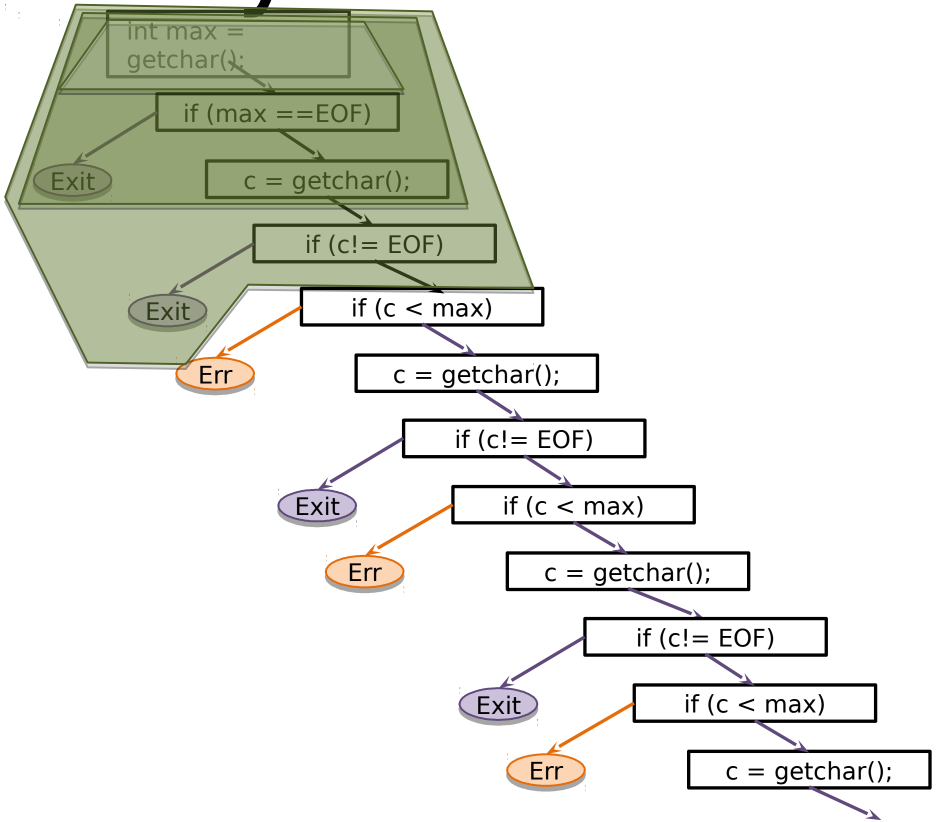
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.



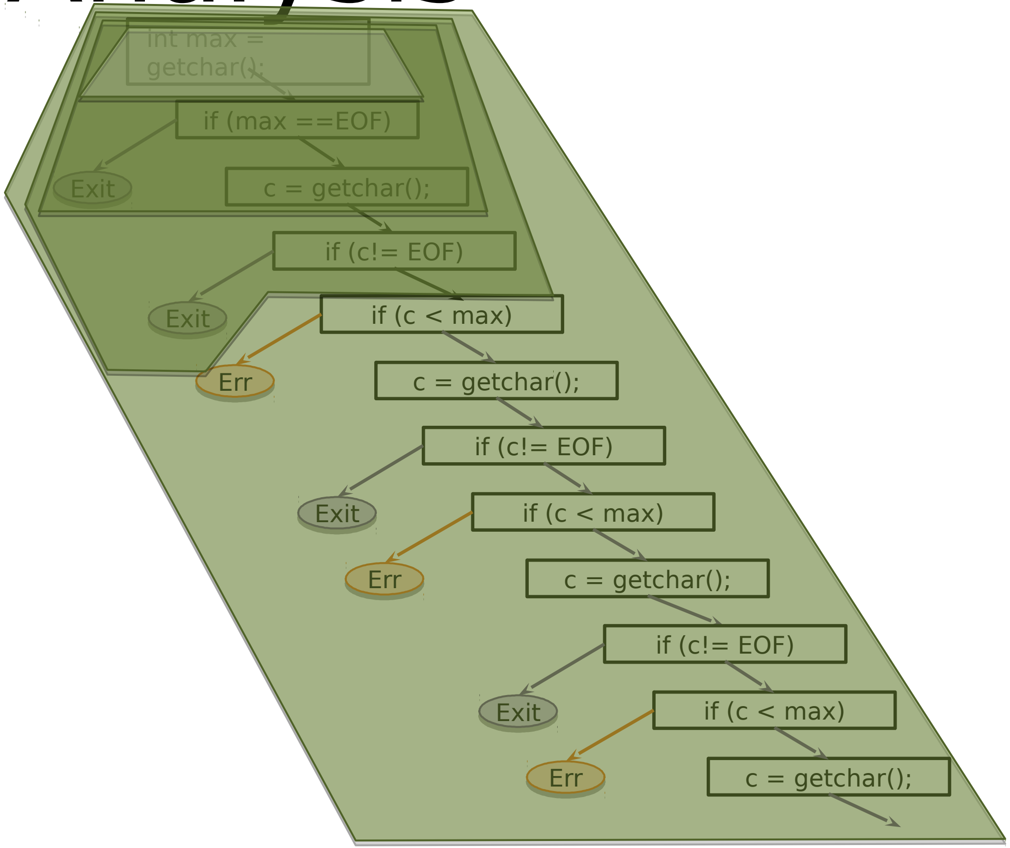
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.



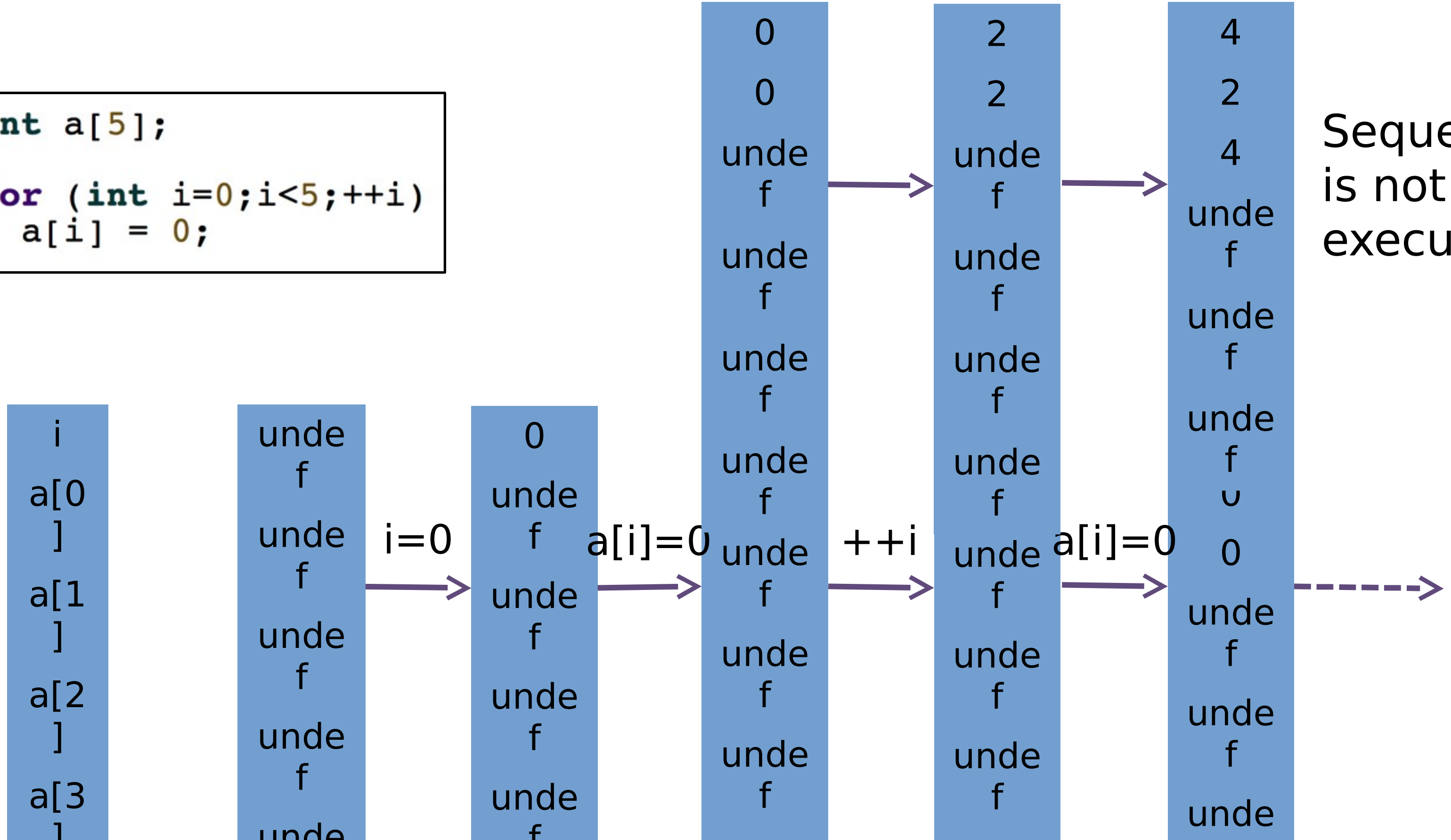
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.



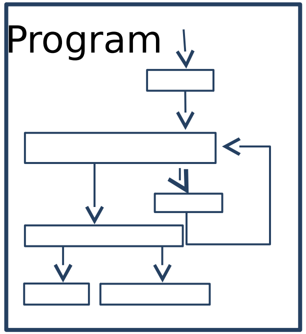
Sequence of States vs. Executions

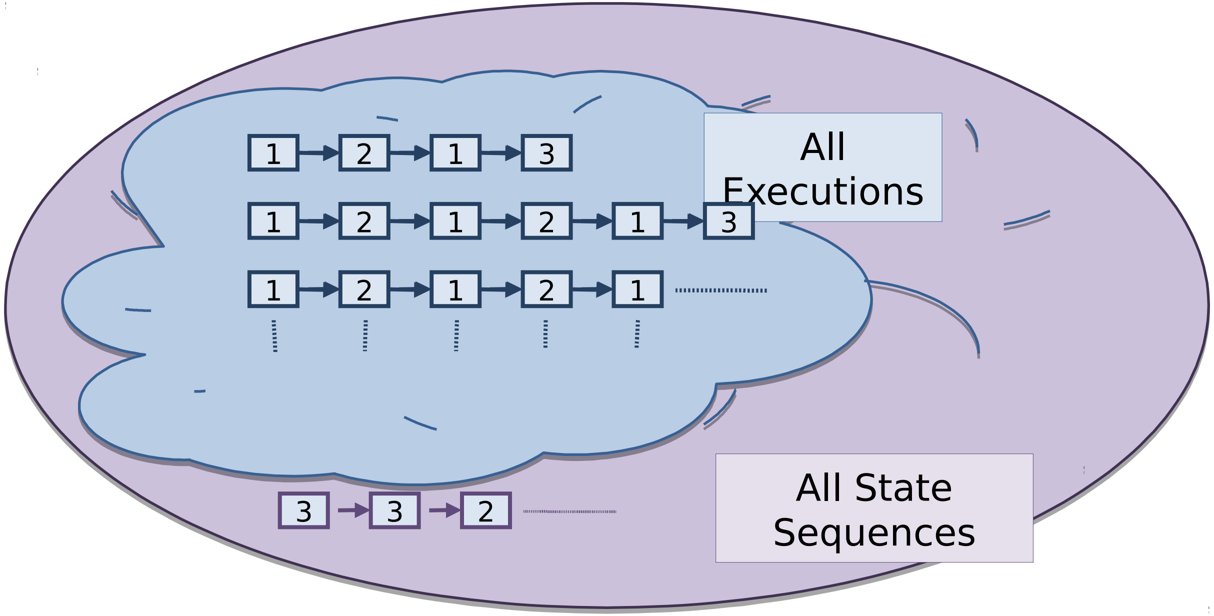
```
int a[5];

for (int i=0;i<5;++i)
  a[i] = 0;
```



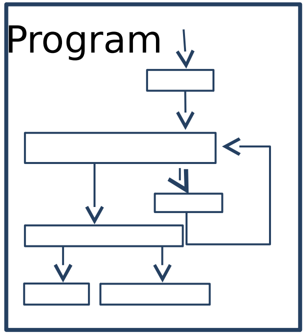
Sequence that is not an execution

All Sequences of States vs. All Executions

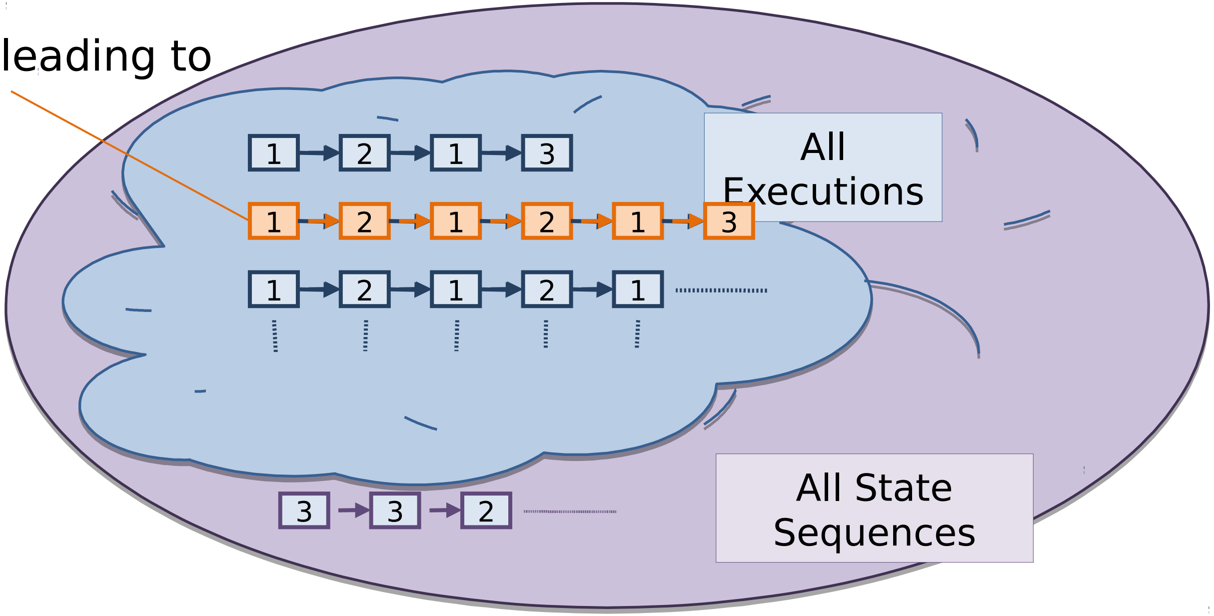




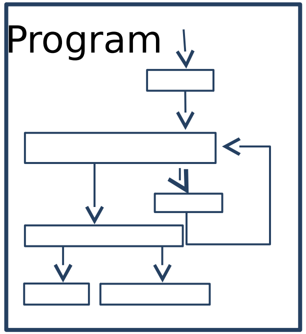
All Sequences of States vs. All Executions

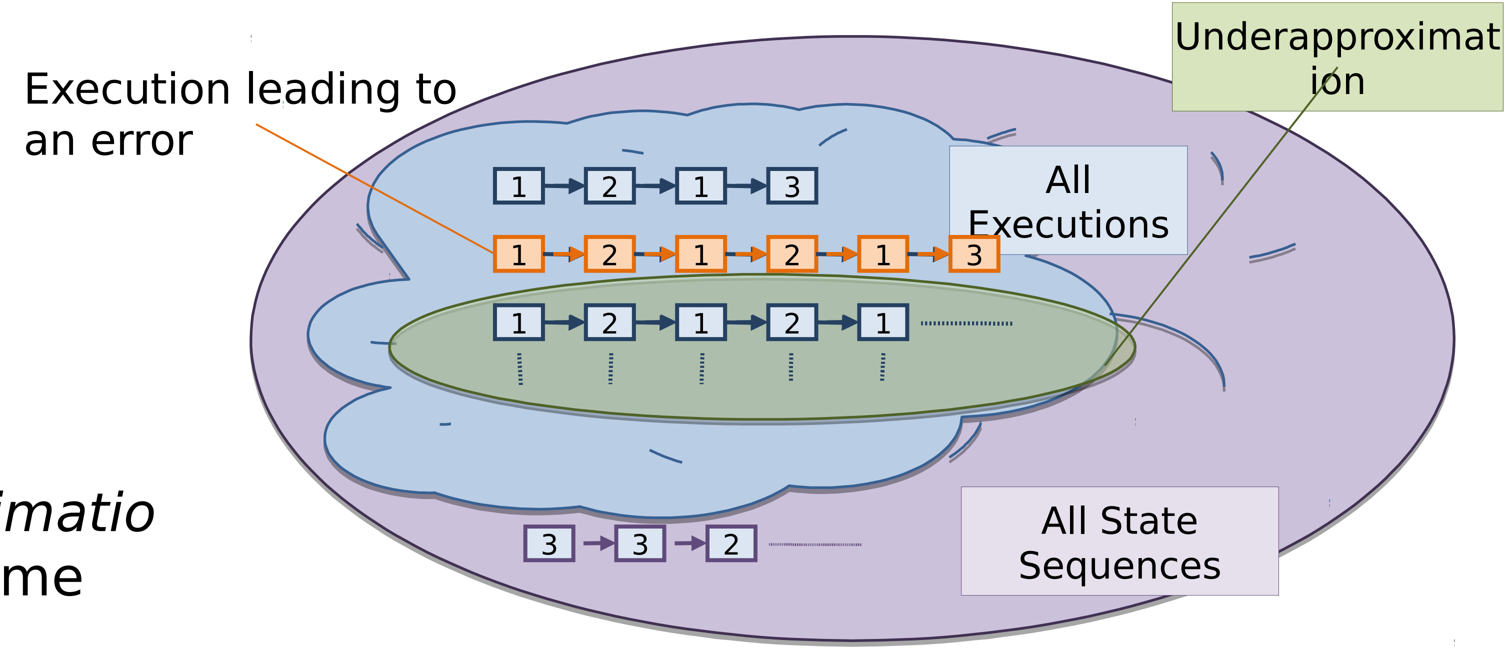


Execution leading to an error



Underapproximation

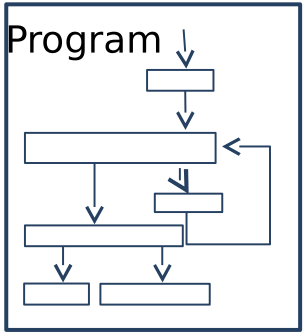


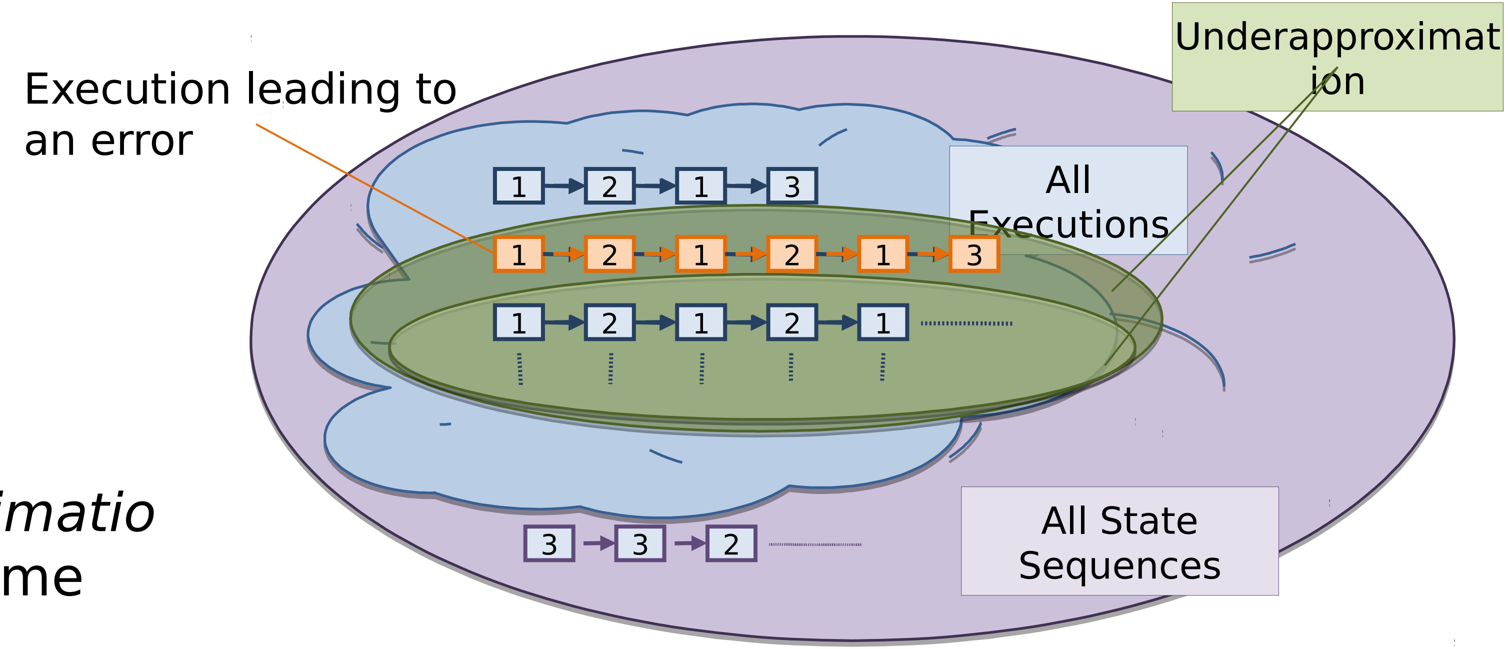


An *underapproximation* contains some but not all executions.

Underapproximate analysis may conclude there is no error when an error exists: a *false negative.*

Underapproximation

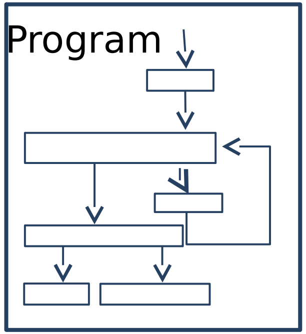


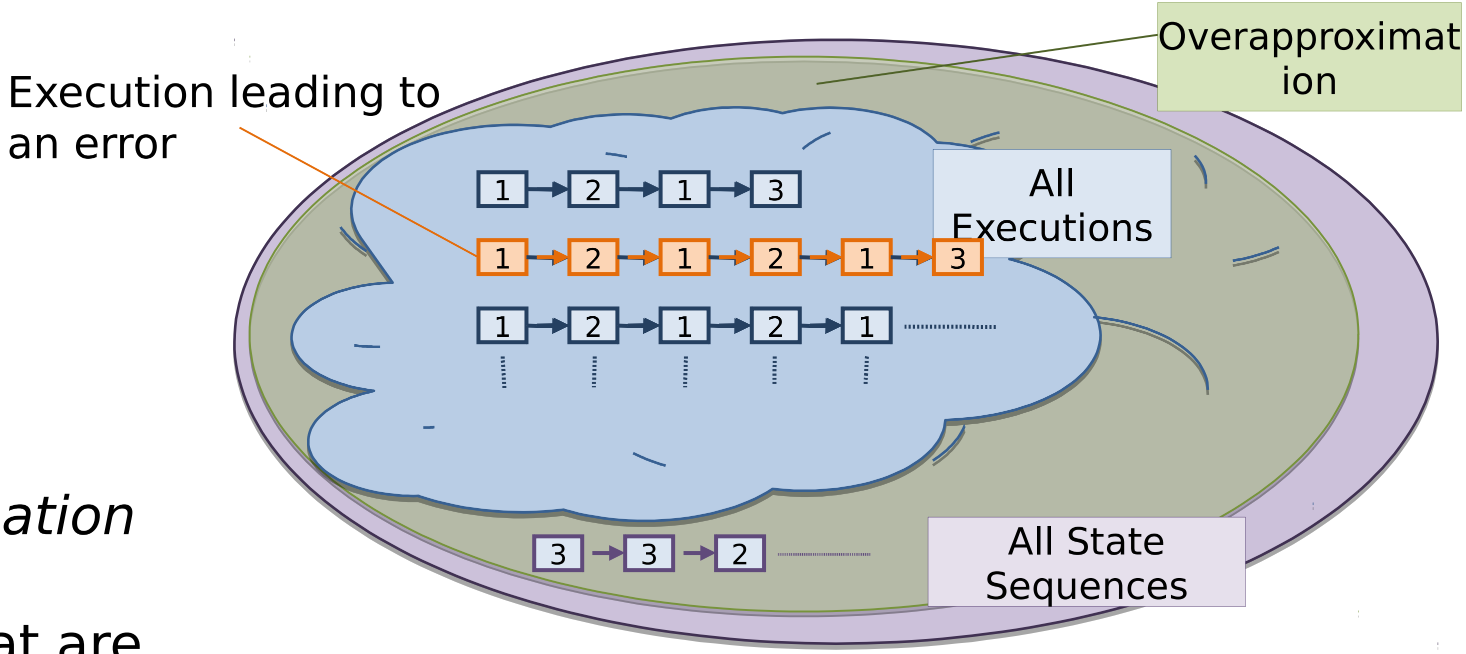


An *underapproximation* contains some but not all executions.

Underapproximate analysis may conclude there is no error when an error exists: a *false negative.* A better underapproximation considers *more* executions.

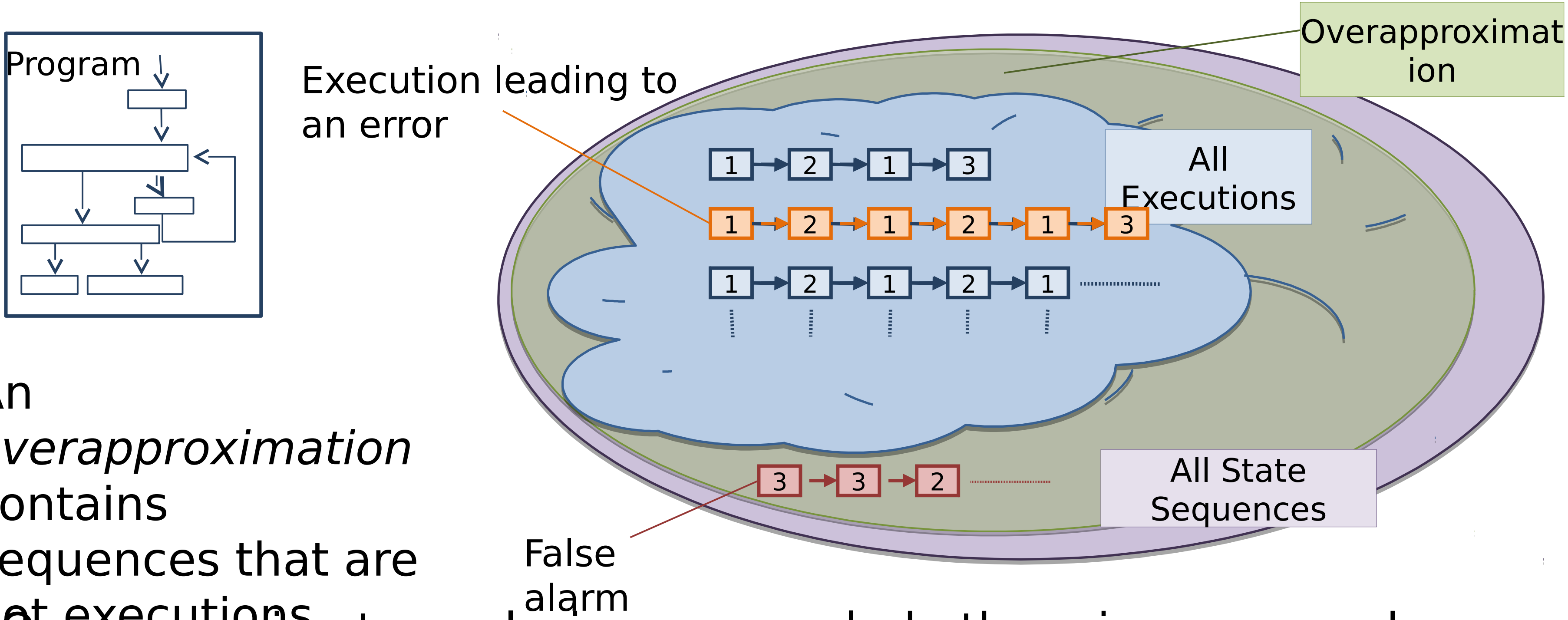
Overapproximation





An *overapproximation* contains sequences that are not executions.

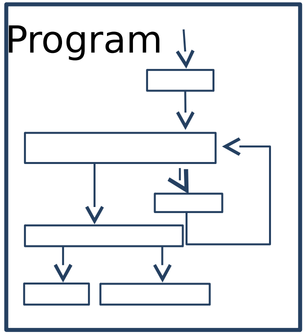
Overapproximation



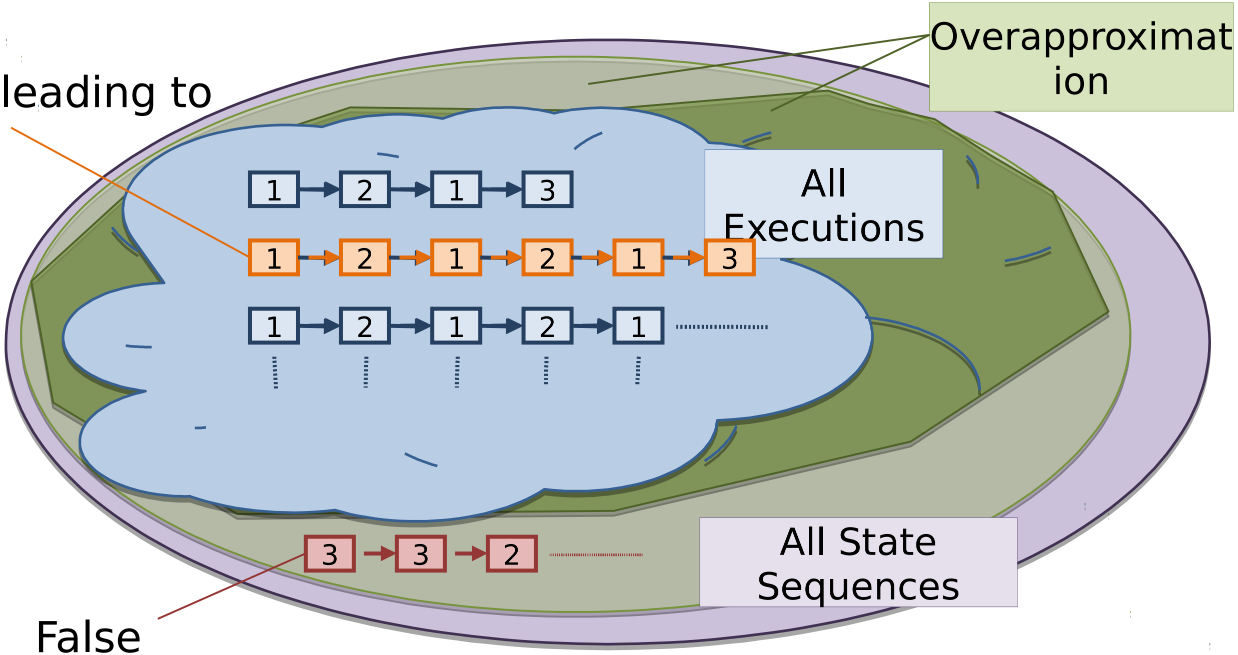
An *overapproximation* contains sequences that are not executions.

Overapproximate analysis may conclude there is an error when no error exists: a *false positive* or *false alarm.*

Overapproximation



Execution leading to an error



False alarm

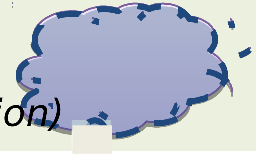
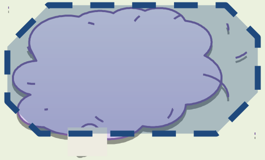
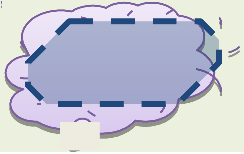
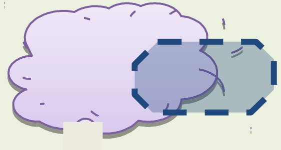
An *overapproximation* contains sequences that are not executions.

Overapproximate analysis may conclude there is an error when no error exists: a *false positive* or *false alarm.* A more precise overapproximation considers fewer sequences that are not executions.

Soundness and Completeness

Property	Definition
Soundness	If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”
Completeness	If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”

Note: these terms have different meaning in other contexts

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**  *(Ex: Manual Program Verification)*	Reports all errors May report false alarms  *(Ex: Abstract Interpretation)*
Unsound	May not report all errors Reports no false alarms  *(Ex: Symbolic Execution)*	May not report all errors May report false alarms **Analysis terminates(?)**  *(Ex: Syntactic Analysis)*

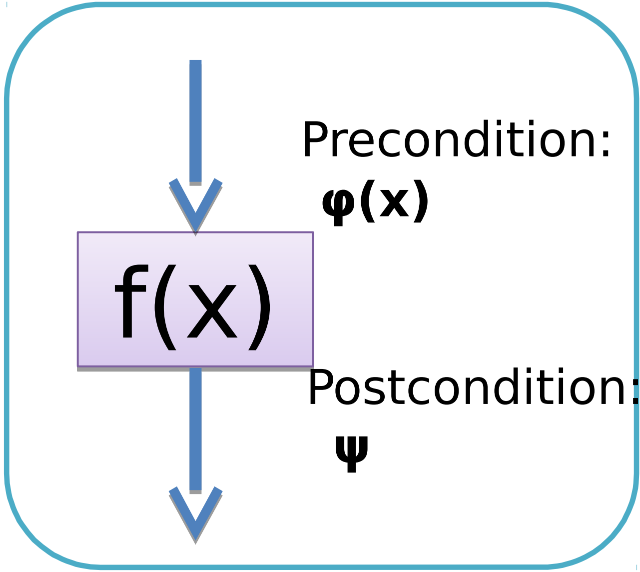
Program Verification

Program Verification

- How to prove a program free of buffer overflows?
  - Precondition
  - Postcondition
  - Loop invariants

Precondition

- Precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`
  - If any precondition is not met, f() may not behave correctly
  - Callee may freely assume obligation has been met
- The concept similarly holds for any statement or block of statements



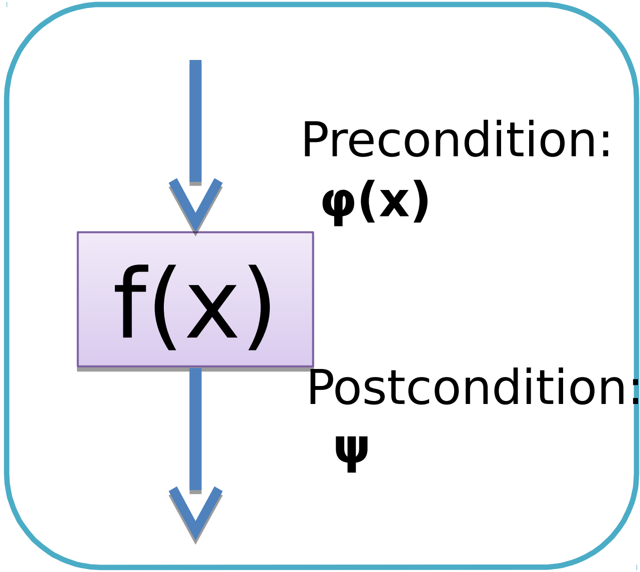
Precondition Example

- Precondition:
  - fp points to a valid location in memory
  - fp points to a file
  - the file that fp points to contains at least 4 characters
  - …

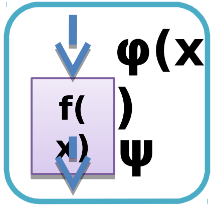
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘\n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Postcondition

- *Postcondition* for `f()`
  - An assertion that holds when `f()` returns
  - `f()` has obligation of ensuring condition is true when it returns
  - Caller may assume postcondition has been established by `f()`



Postcondition Example

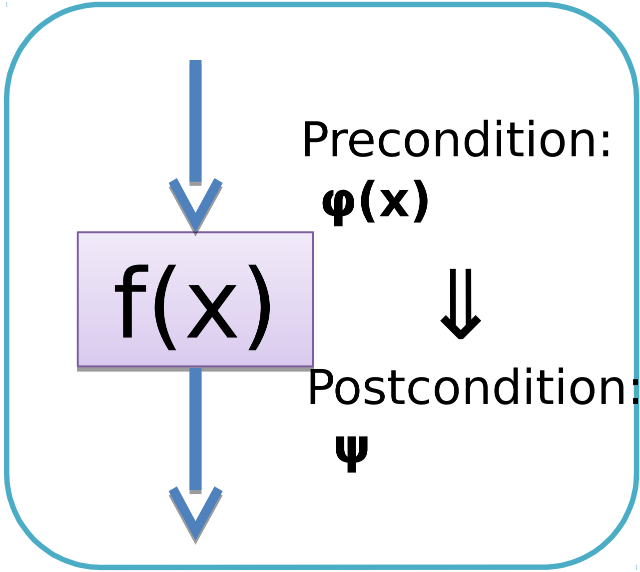


- Postcondition:
  - *buf* contains no uppercase letters
  - (return 0) ⇒(cmd[0..3] == “GET “)

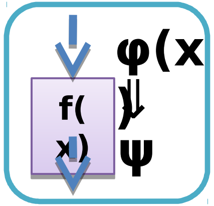
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Proving Precondition ⇒ Postcondition

- Given preconditions and postconditions
  - Specifying what obligations caller has and what caller is entitled to rely upon
- Verify: No matter how function is called,
  - if precondition is met at function's entrance,
  - then postcondition is guaranteed to hold upon function's return

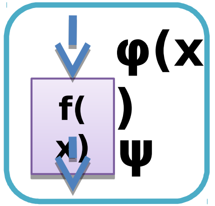


Proving Precondition ⇒ Postconditio

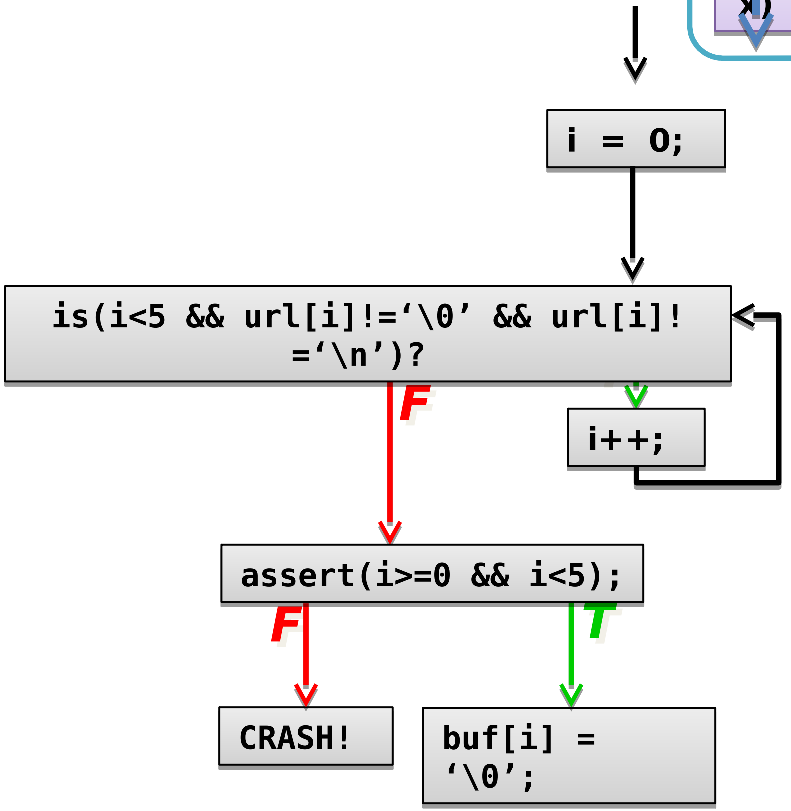


- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Use logical reasoning
- Requirement:
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one

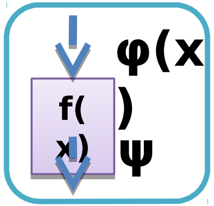
We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.



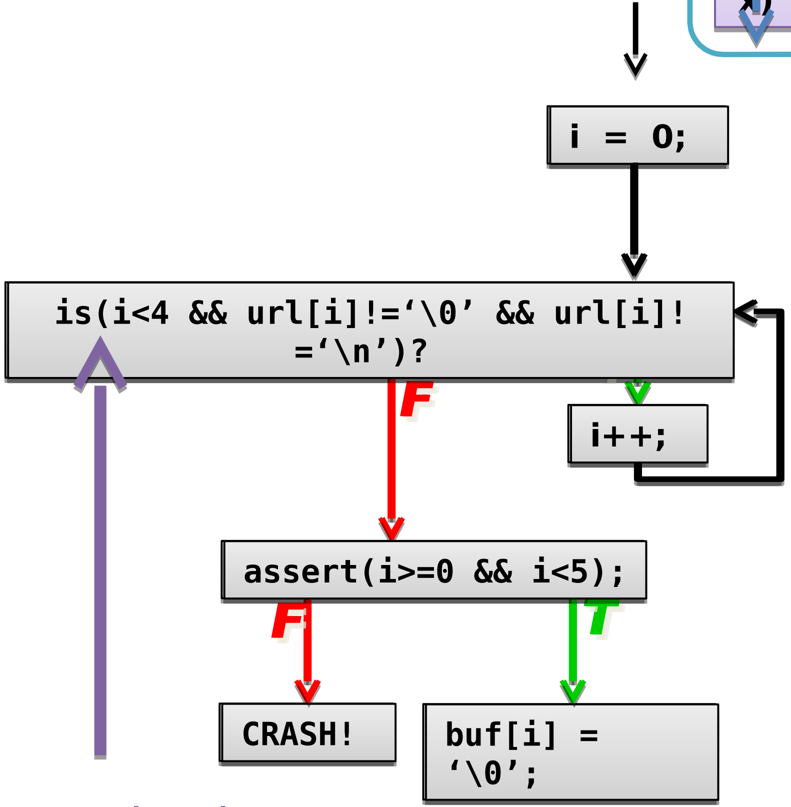

```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘n’)
{
    buf[i] = tolower(url[i]);
    i++;
  }
  assert(i>=0 && i <5);
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```



We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.



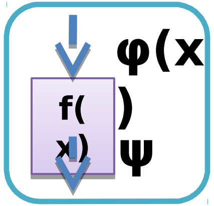

```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<4 && url[i]!=‘\0’ && url[i]!=‘n’)
{
    buf[i] = tolower(url[i]);
    i++;
  }
  assert(i>=0 && i <5);
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```



Bug Fixed!

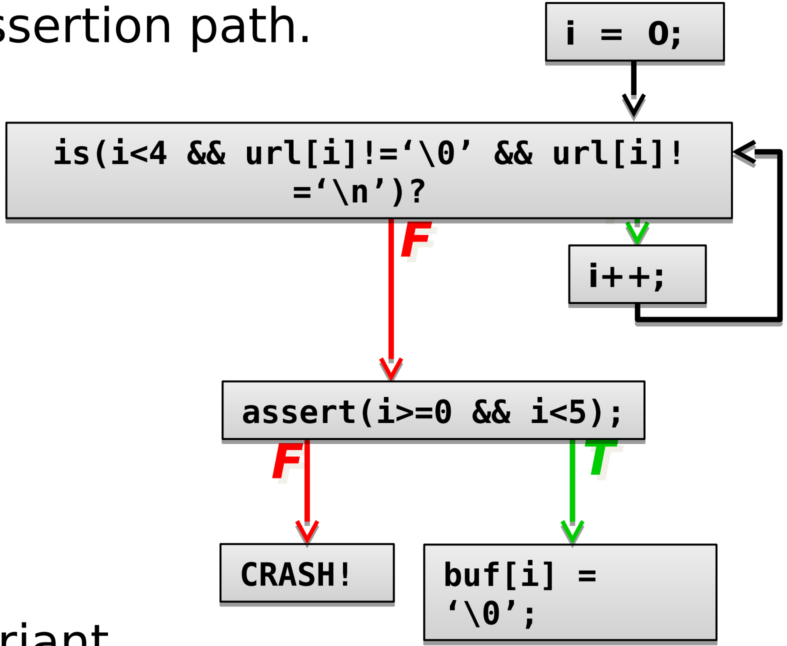
Dawn Song

We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists…



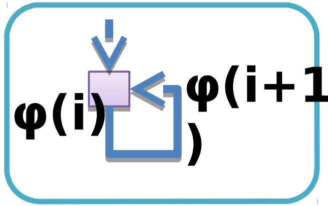

```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<4 && url[i]!=‘\0’ && url[i]!=‘n’)
{
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

…So assuming fp points to a file that begins with “GET “, we want to show that *parse* never goes down the false assertion path.

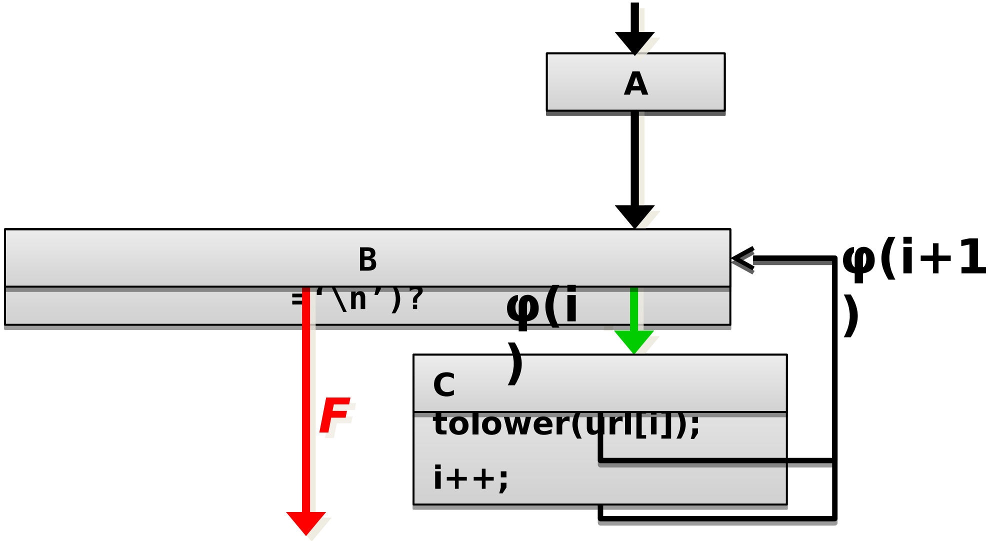


ut first, we will need the concept of loop invariant.

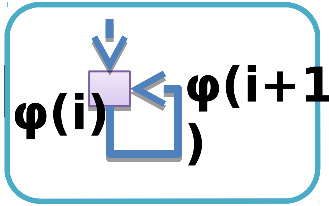
Loop Invariant and Induction



- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - Both a pre- and post-condition for the loop body



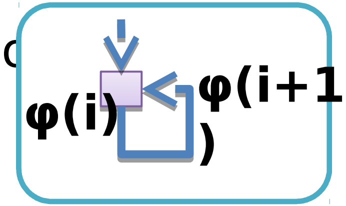
Loop Invariant and Induction

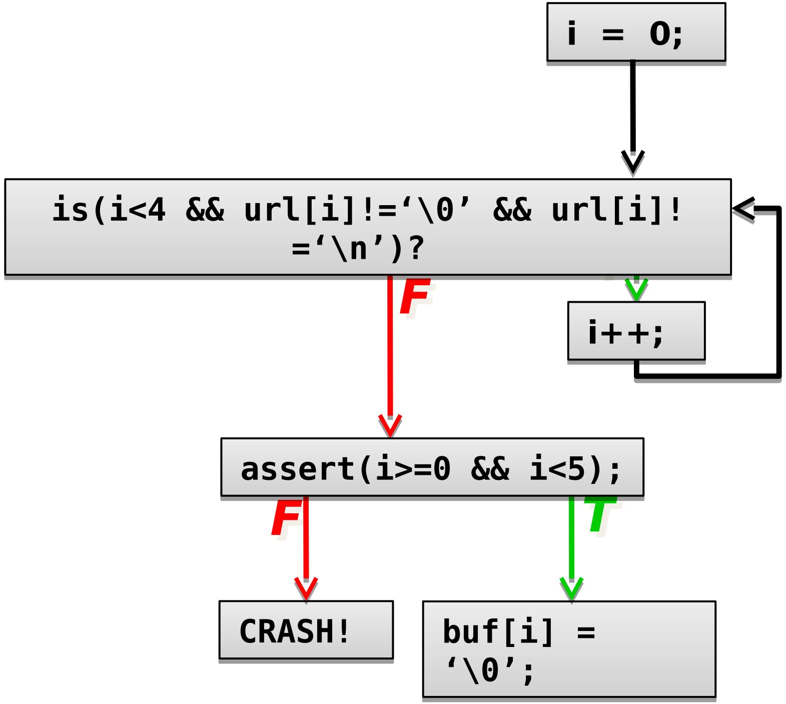


- To verify:
  - Base Case: Prove true for first iteration: **φ(0)**
  - Inductive step: Assume **φ(i)** at the beginning of the loop. Prove **φ(i+1)** at the start of the next iteration.

Try with our familiar example, proving that (0≤i<5) after the loop terminates:

LOOP INVARIANT: /* φ(i) = (0≤i<5) */





Base Case:

/* φ(0) = (0≤0<5) */

Inductive Step:

/* assume(0≤i<5)at the beginning of the loop

/* for (0≤i<4), clearly (0≤i+1<5) */

/* (i=5) is not a possible case since that would fail the looping predicate */

/* ⇒ (0≤i+1<5) at the end of the loop */

/* ⇒ *parse* never fails the assertion */

Function Post-/Pre-Conditions

- For every function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function `f()` by looking at only its code and the annotations on every function `f()` calls
    - Can ignore code of all other functions and functions called transitively
  - Makes reasoning about `f()` an almost purely local activity

Dafny

- A programming language with built-in specification constructs.
- A static program verifier to verify the functional correctness of programs.
- Powered by Boogie and Z3.
- Available here: http://rise4fun.com/dafny/

Documentation

- Pre-/post-conditions serve as useful documentation
  - To invoke Bob's code, Alice only has to look at pre- and post-conditions – she doesn't need to look at or understand his code
- Useful way to coordinate activity between multiple programmers:
  - Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  - Alice and Bob can negotiate the interface (and responsibilities) between their code at design time

Preventing Security Vulnerabilities

- Identify implicit requirements code must meet
  - Must not make out-of-bounds memory accesses, deference null pointers, etc.
- Prove that code meets these requirements
  - Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds

Preventing Security Vulnerabilities

- How easy it is to prove a certain property of code depends on how code is written
  – Structure your code to make it easy to prove