Foundations for SE Analysis

Formal models

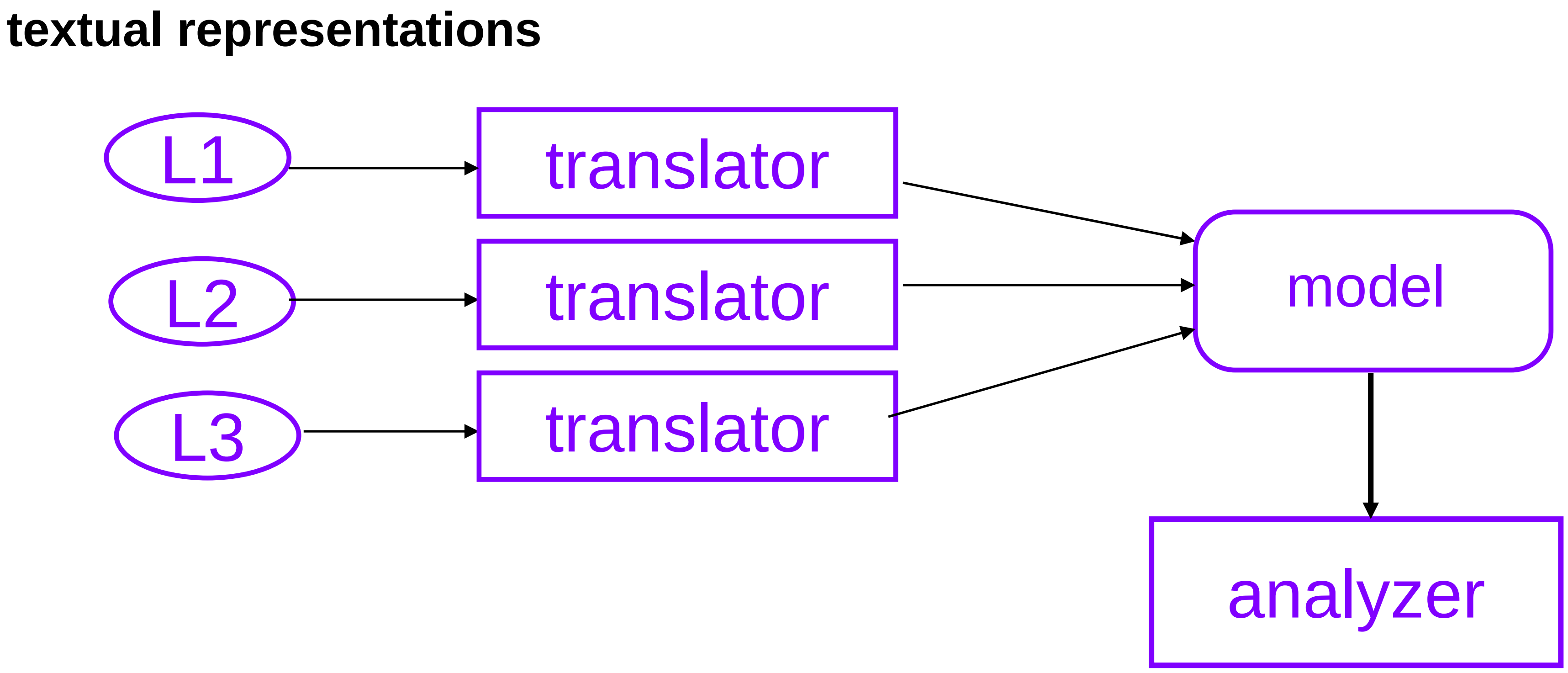
- Analysis is usually done on a model of an artifact
  - textual representation of the artifact is translated into a model that is more amenable to analysis then the original representation
  - the translation may require syntactic and semantic analysis so that the model is as accurate as possible

    e.g., x:= y + foo.bar
  - model must be appropriate for the intended analysis
- graphs are the most common form of models used
  - e.g., abstract syntax graphs, control flow graphs, call graphs, reachability graphs, Petri nets, program dependence graphs

Ideally want general models

- different languages
  - e.g., Ada, C++, Java
- different levels of abstraction/detail
  - e.g., detailed design, arch. design
- different kinds of artifacts
  - e.g., code, designs, requirements

Creating a Common Underlying Model



Graphs

- A graph, $G = (N, E)$, is an ordered pair consisting of a node set, N, and an edge set, $E = \{(n_i, n_j)\}$
  - If the pairs in E are ordered, then G is called a directed graph and is depicted with arrowheads on its edges
  - If not, the graph is called an undirected graph
- Graphs are suggestive devices that help in the visualization of relations. The set of edges in the graph are visual representations of the ordered pairs that compose relations
- Graphs provide a mathematical basis for reasoning about s/w

Paths

- a **path**, P, through a directed graph $G = (N, E)$ is a sequence of edges, $((n_{i,1}, n_{j,1}), (n_{i,2}, n_{j,2}), \ldots, (n_{i,t}, n_{j,t}))$

  such that $n_{j,k-1} = n_{i,k}$ for all $2 \leq k \leq t$
- $n_{i,1}$ is called the start node and $n_{j,t}$ is called the end node
- the length of a path is the number of edges in the path
- Paths are also frequently represented by a sequence of nodes $(n_{i,1}, n_{i,2}, n_{i,3}, \ldots, n_{i,t})$

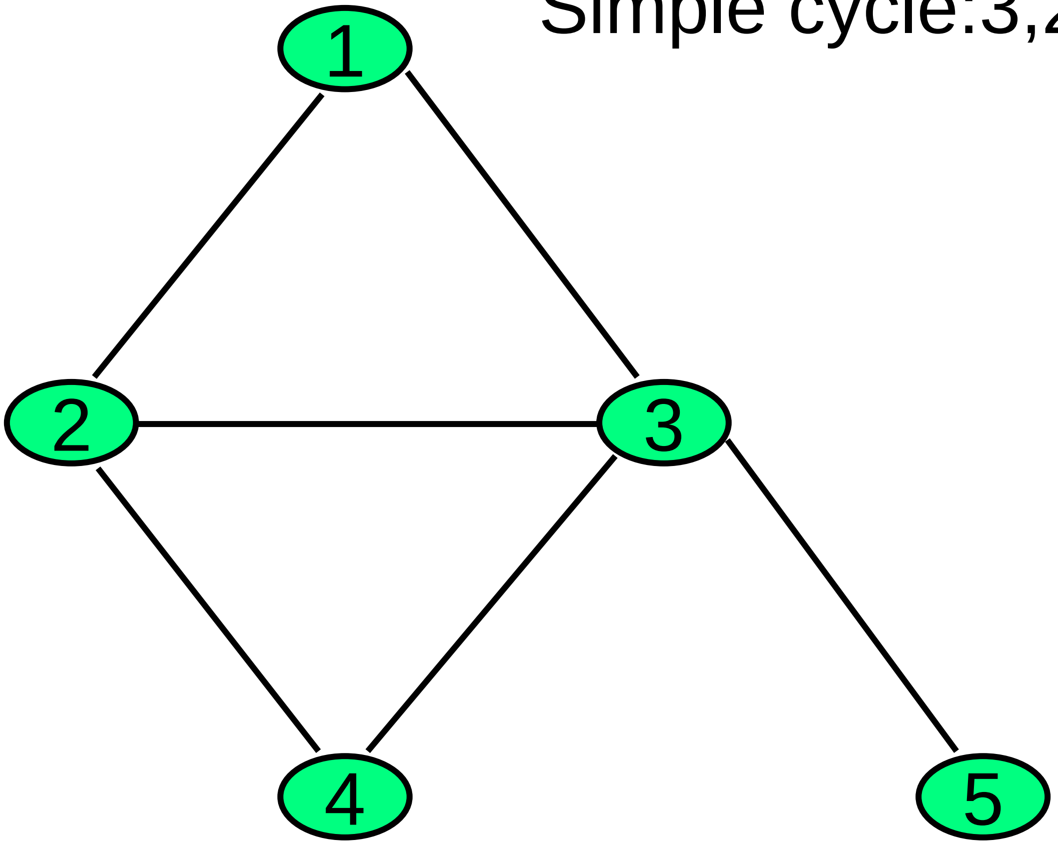
Cycles

- a *cycle* in a graph G is a path whose start node and end node are the same
- a *simple cycle* in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)
- if a graph G has no path through it that is a cycle, then the graph is called *acyclic*

Examples

Cycle:1,3,2,4,3,1

Simple cycle:3,2,4,3

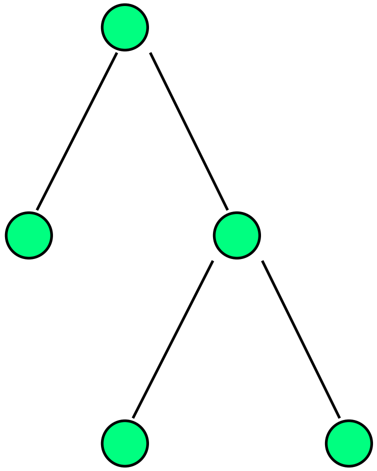


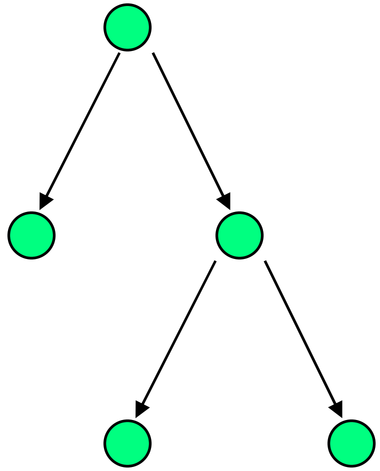
Trees

- an acyclic, undirected graph is called a tree
- if the undirected version of a directed graph is acyclic, then the graph is called a directed tree
- if the undirected version of a directed graph has cycles, but the directed graph itself has no cycles, then the graph is called a Directed Acyclic Graph (DAG)

Examples

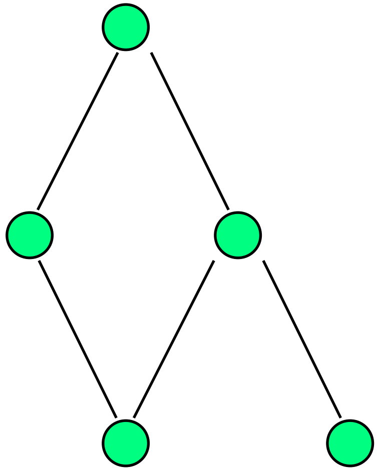
tree

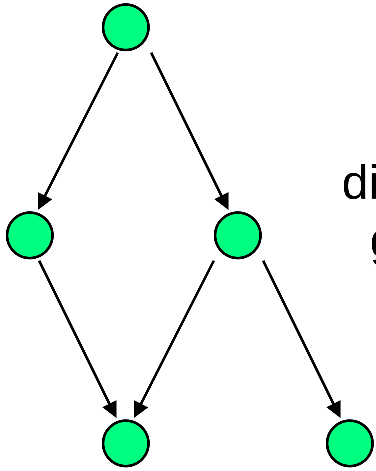




directed tree

cyclic undirected graph





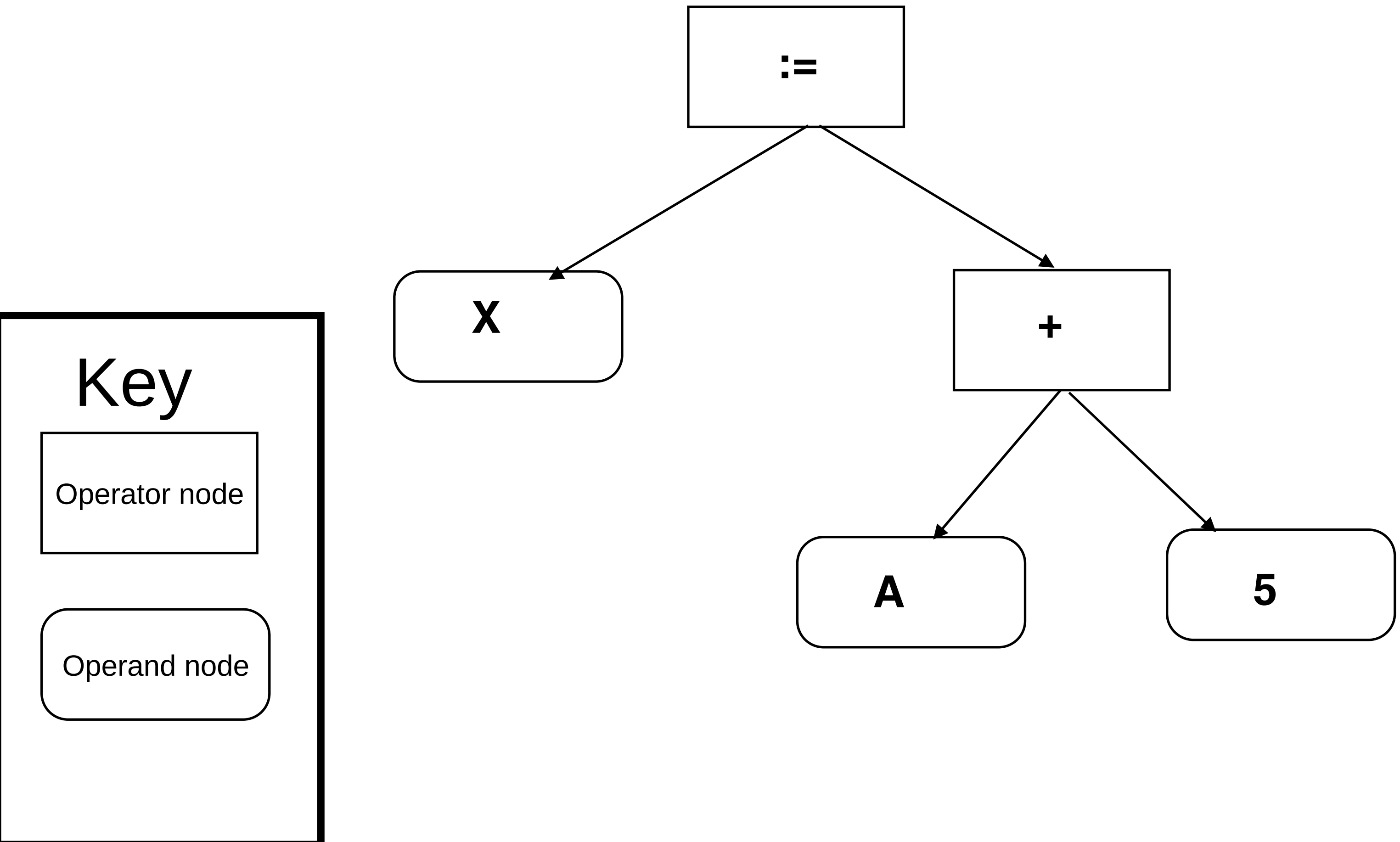
directed acyclic graph (DAG)

Abstract Syntax Tree (AST)

- a common form for representing expressions
  - executable statements are expressions
  - programs are expressions, where the operator is execute and the operands are the statements
- 2 kinds of nodes: operator and operands
  - operator applied to N operands
- An abstract syntax graph G = (N1, N2, E) where N1 are nodes that represent operators in the language, N2 are nodes that represent identifiers or literals , and E represents is "applied to"

Example Abstract Syntax Tree

X:= A + 5;



Abstract Syntax Trees have many advantages

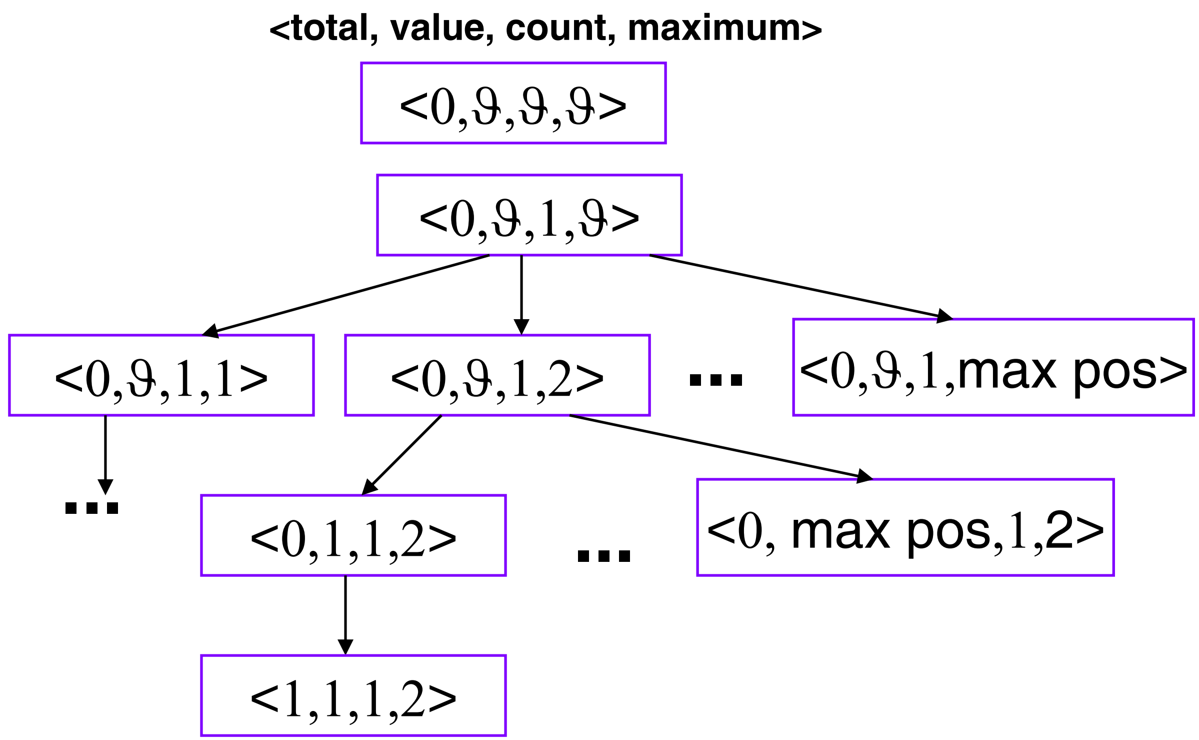
- provide a visual display of the body of an object
  - body of an assignment, addition, while, etc.
- supports incremental modification
  - incremental syntactic or semantic analysis
- Basis for structural editing
  - user is provided with a template and fills in the slots
  - can assure syntactic consistent
  - need to control granularity of consistency checking
    - e.g., keystroke, semi-colon, user-request
- Used to create other graph models

Computation tree

- Models all the possible executions of a system
- At each node, shows the state (value) of each variable
- Effectively infinite number of paths
- Some paths may be effectively infinite

Example Computation Tree

```
total, value, count, maximum : pos int;

total := 0;

count := 1;

read maximum;

while (count <= maximum) do

        read value;

        total := total + value;

        count: = count + 1;

endwhile;

print total;
```



Computation Trees have few advantages

- Represent the space that we want to reason about
- For anything interesting they are too large to create or reason about
- Other models of executable behavior are providing abstractions of the computation tree model
  - Abstract values
  - Abstract flow of control
  - Specialize abstraction depending on focus of analysis

Control Flow Graph (CFG)

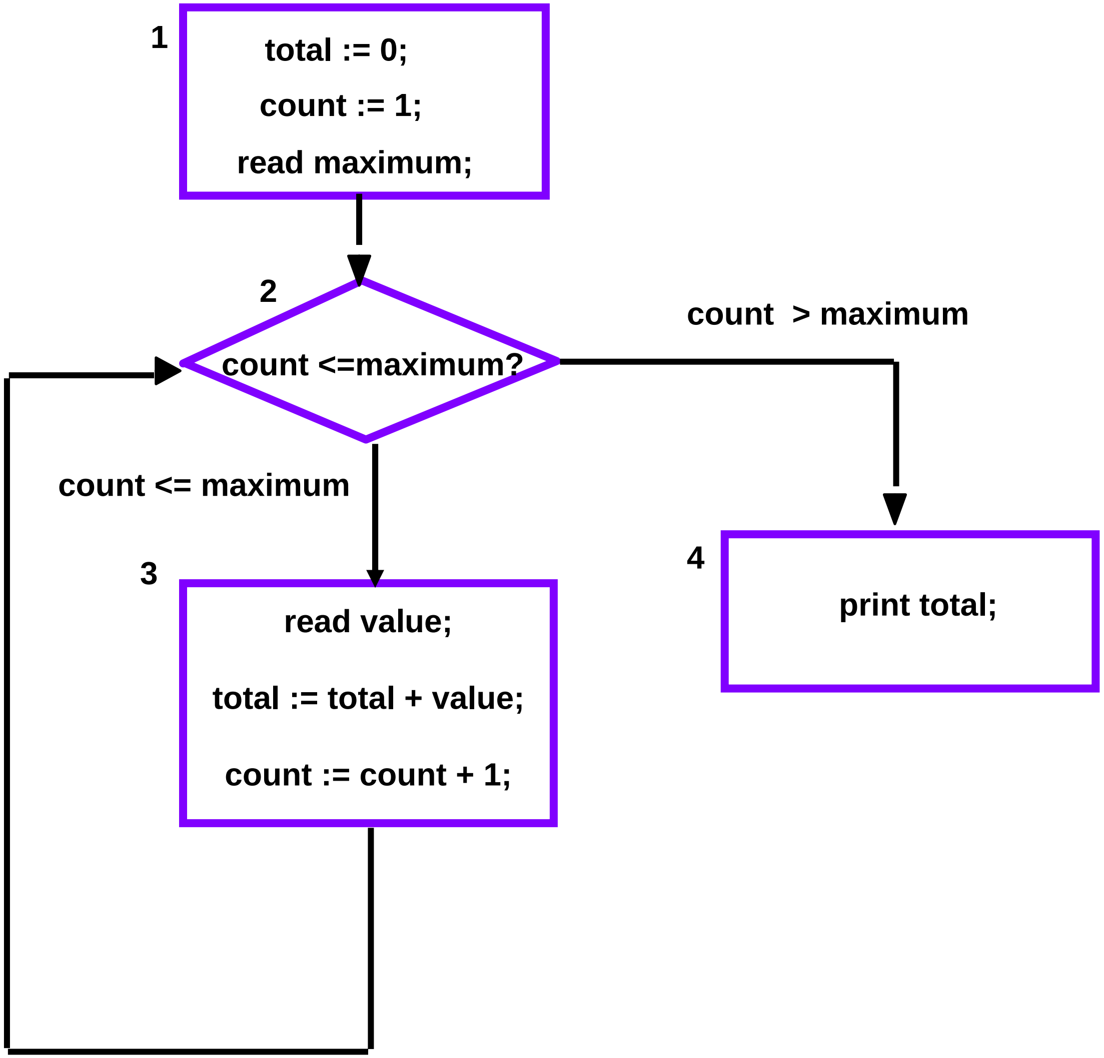
- represents the flow of executable behavior
- G = (N, E, S, T) where
  - the nodes N represent executable instructions (statement or statement fragments);
  - the edges E represent the *potential* transfer of control;
  - S is a designated start node;
  - T is a designated final node
  - E = { (n_i, n_j) | syntactically, the execution of n_j follows the execution of n_i}

Control Flow Graph (CFG)

- Nodes may correspond to single statements, parts of statements, or several statements
- Execution of a node means that the instructions associated with a node are executed in order from the first instruction to the last
- Nodes are 1-in, 1-out

Example

```
total, value, count, maximum : int;

total := 0;

count := 1;

read maximum;

while (count <= maximum) do

        read value;

        total := total + value;

        count: = count + 1;

endwhile;

print total;
```



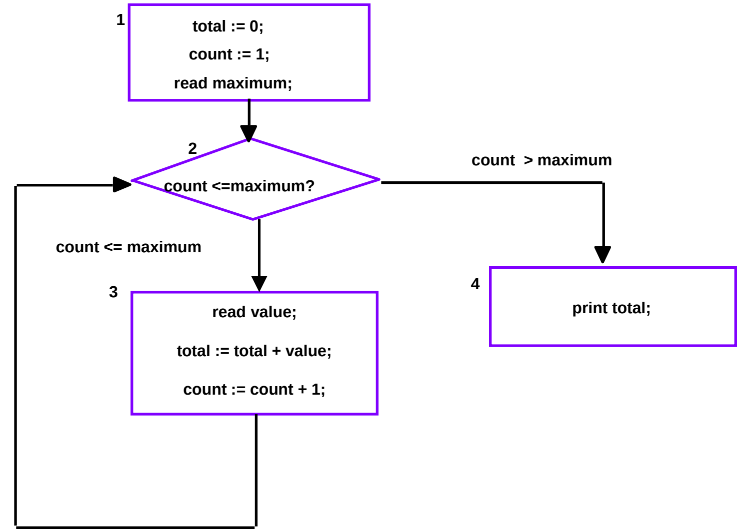
Control Flow Graphs

- a subpath through a control flow graph is a sequence of nodes $(n_1, n_2, \ldots n_t)$ where for each n_k, $1 \leq k < t$, (n_k, n_{k+1}) is an edge in the graph

  e.g., 2, 3, 2, 3, 2, 4

- a complete path starts at the start node and ends at the final node

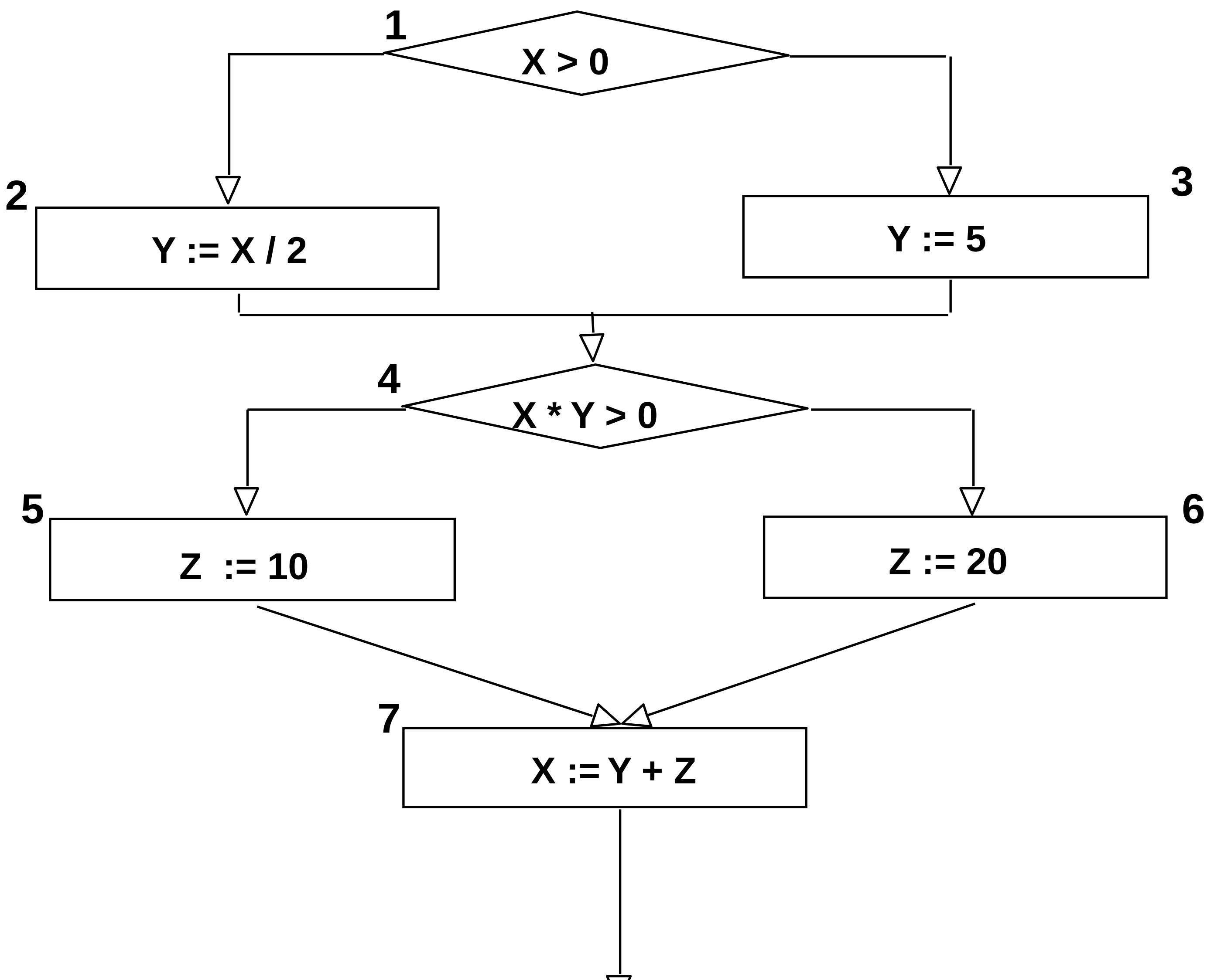
  1, 2, 3, 2, 4



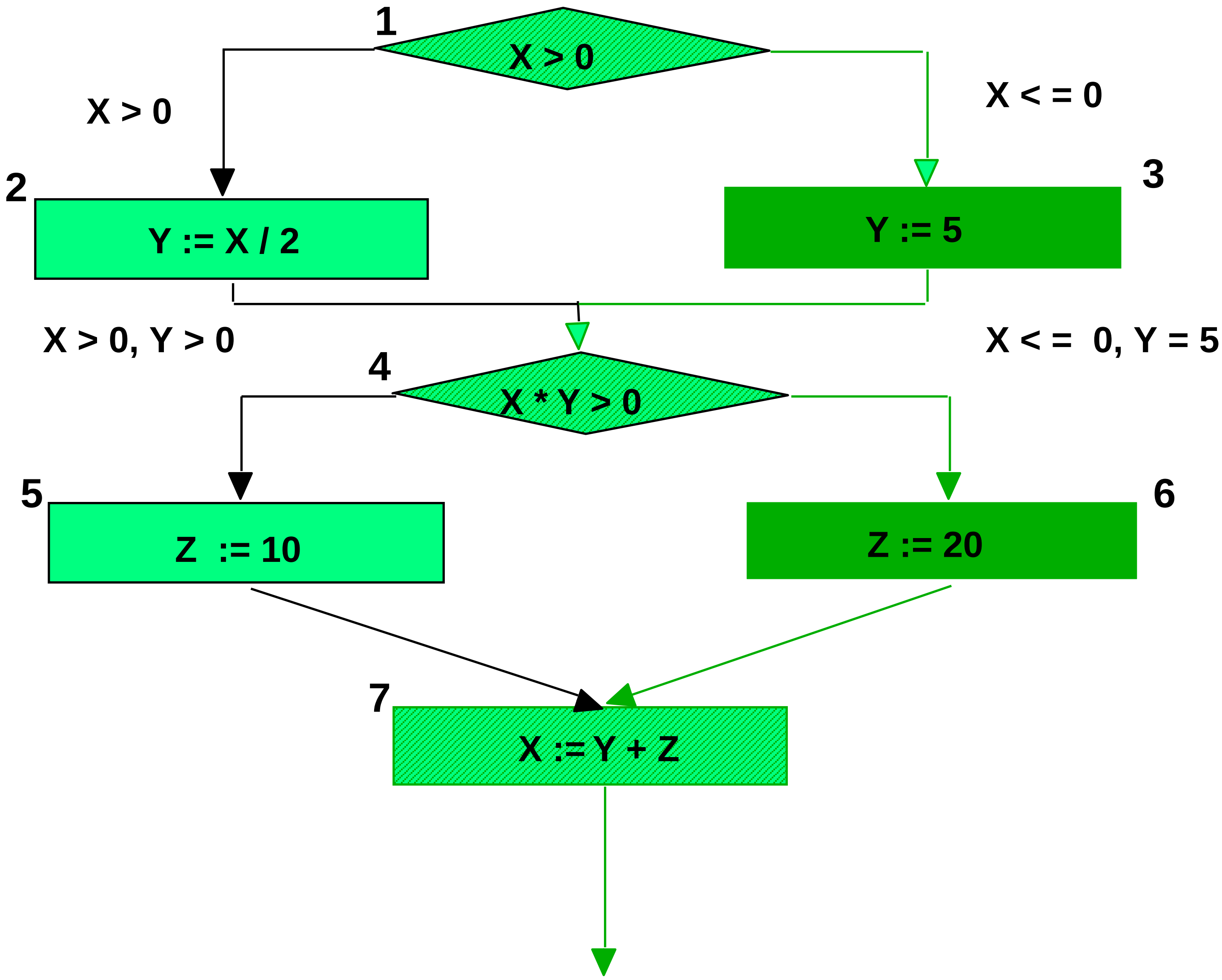
Control Flow Graphs

- Every executable sequence in the represented component corresponds to a path in G
- not all paths correspond to executable sequences
  - requires additional semantic information
  - "infeasible paths" are not an indication of a fault
- CFG usually overestimates the executable behavior

Example with an infeasible path



Example with an infeasible path

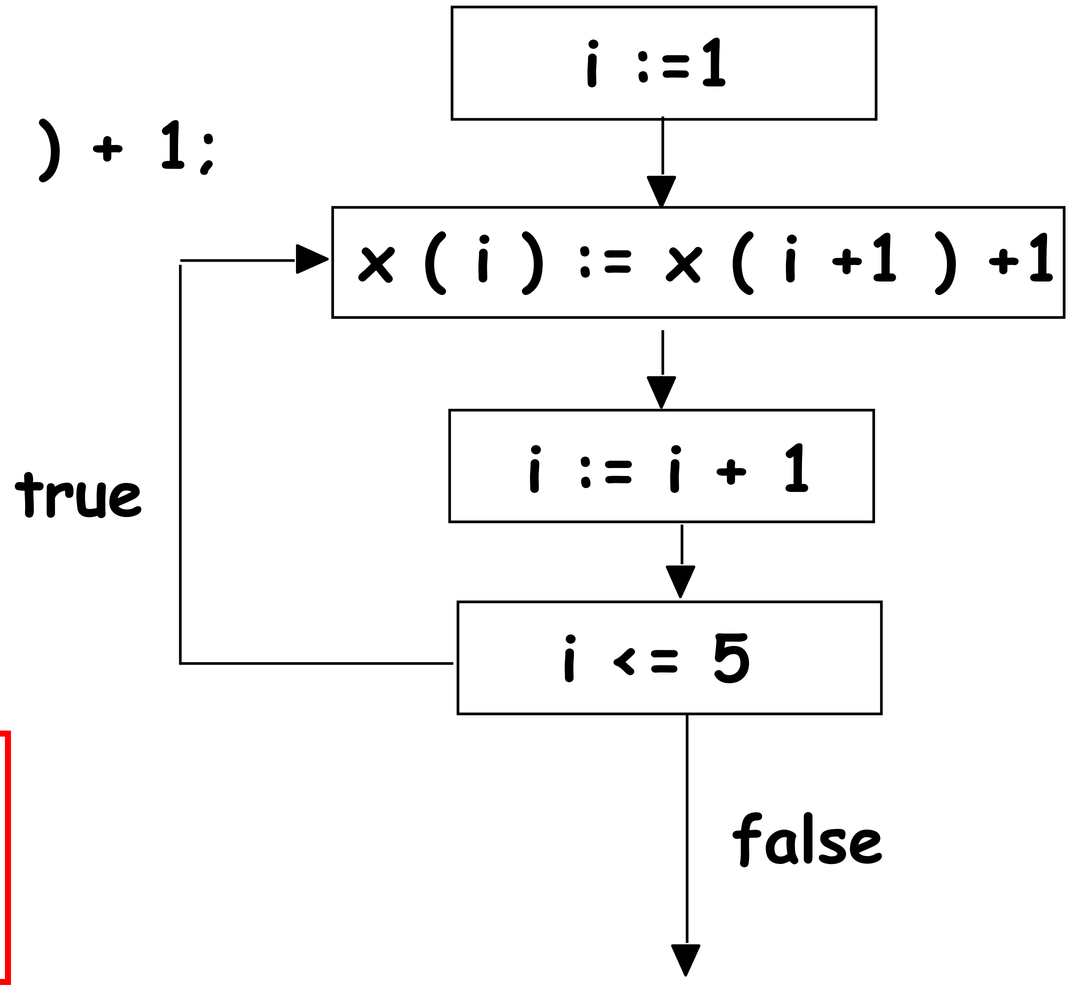


Example Paths

- Feasible path: 1, 2, 4, 5, 7
- Infeasible path: 1, 3, 4, 5,7
- Determining if a path is feasible or not requires additional semantic information
  - In general, unsolveable
  - In practice, intractable

Another example of an infeasible path

For i :=1 to 5 do
x (i) := x (i +1) + 1;
end for:



Note, implicit instructions are explicitly represented

Infeasible paths vs. unreachable code and dead code

unreachable code

```
X := X + 1;
Goto loop;
Y := Y + 5;
```

Never executed

dead code

```
X := X + 1;
X := 7;
X := X + Y;
```

'Executed', but irrelevant

Modeling behavior

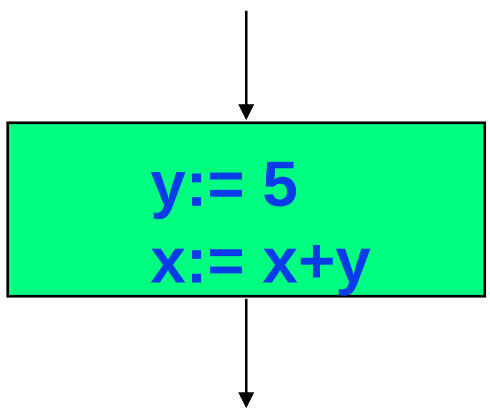
- want to accurately capture the semantics
  - i := i + 1

    not explicitly stated in the For loop
- the way in which information is represented in a CFG depends on the analysis that is planned

Reducing the CFG

- basic blocks are nodes that contain sequential execution

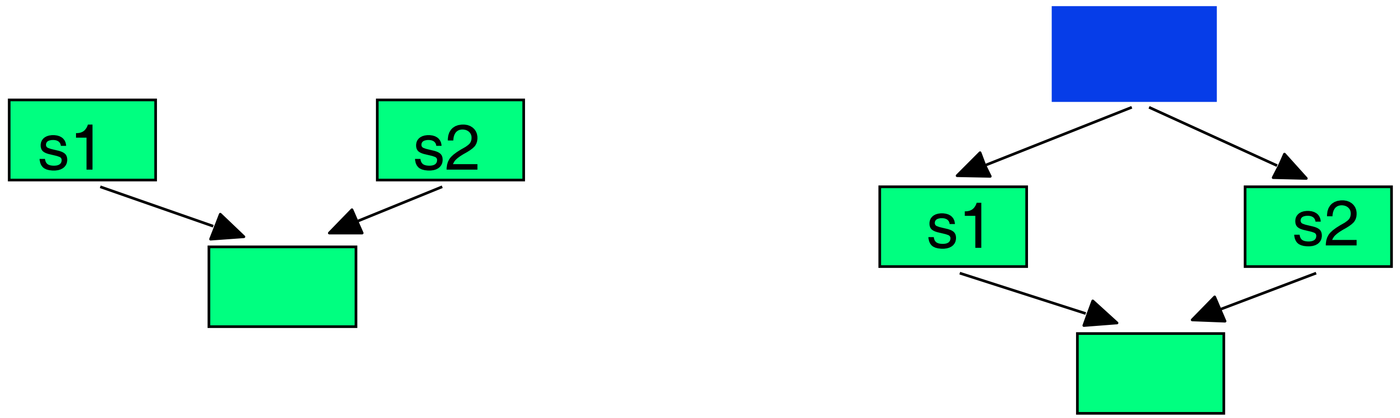
  Can reduce the number of nodes in the CFG, but may add more complications to the analysis



  - y defined in the node before it is used
  - x defined in the node after it is used

CFGs

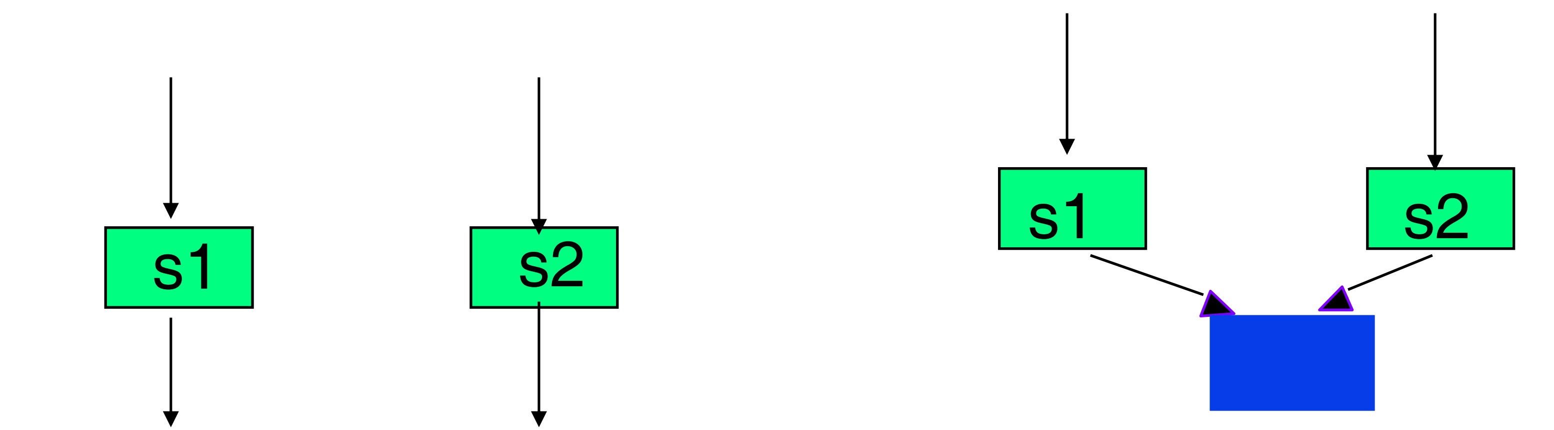
- Usually, have a single start and a single exit node
- Multiple start nodes $\Rightarrow$ Single start node



  - May have to encode information about each start in an auxiliary variable

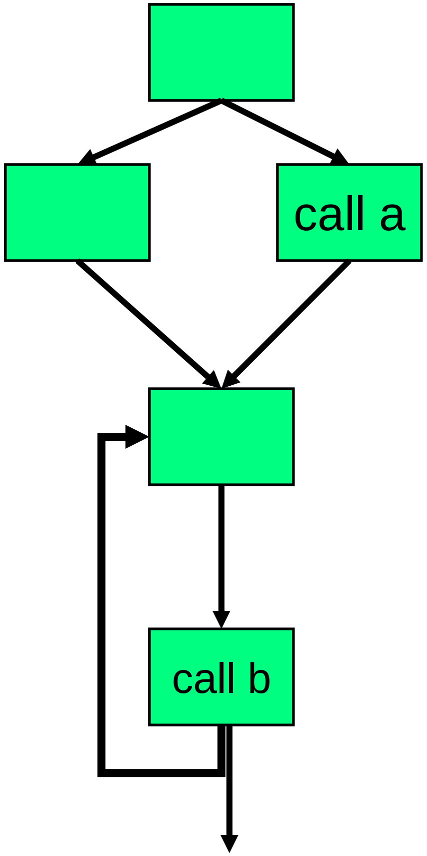
CFGs

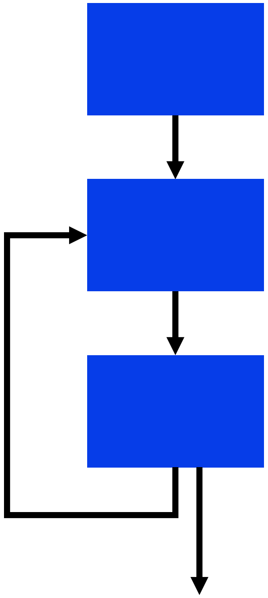
- Multiple exit nodes ⇒ Single terminal node



- Single-entrance, single exit CFGs facilitate inter-component analysis
  - plugable

Plugable components





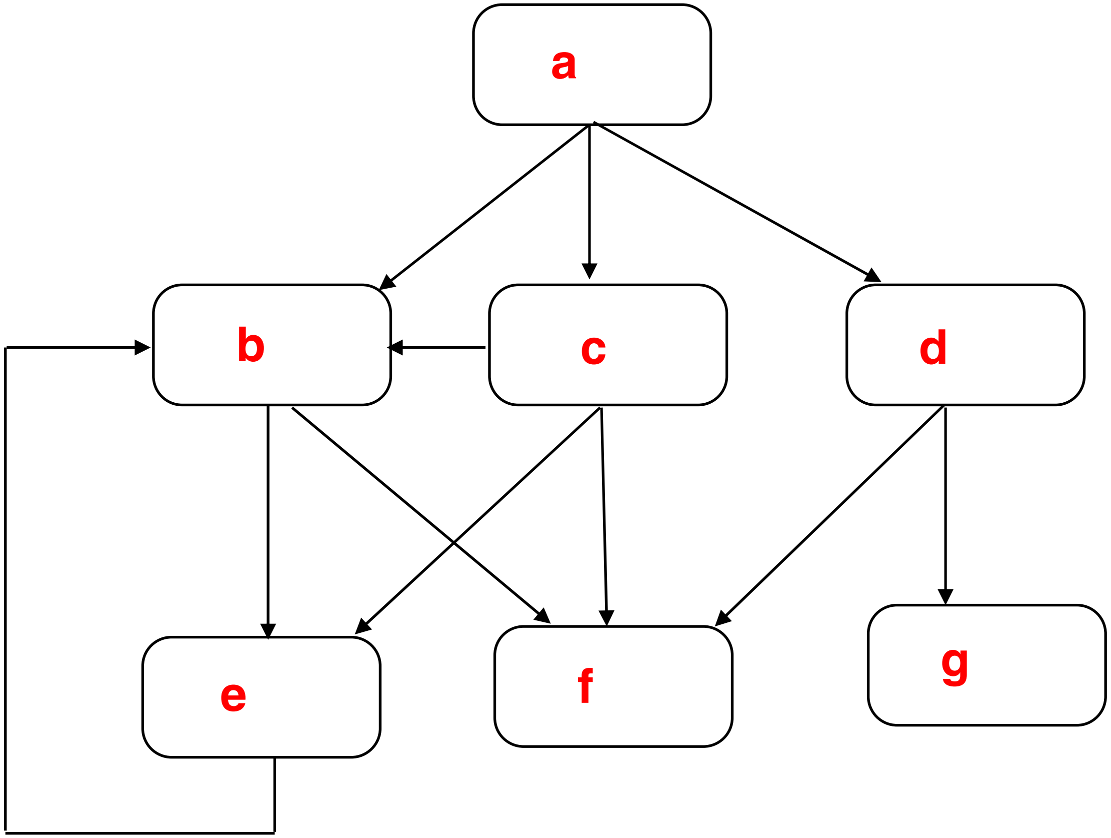
Benefits of CFG

- Probably the most commonly used representation
  - Numerous variants
- Basis for inter-component analysis
  - Collections of CFGs
- Basis for various transformations
  - Compiler optimizations
  - S/W analysis
- Basis for automated analysis
  - Graphical representations of interesting programs are too complex for direct human understanding

Call Graph

- represents "may invoke" relationship between components
- $G = (N, E, M)$ where
  the nodes N represent invokable entities;
  the edges E represent the potential for one entity
  to invoke another entity;
  M is the start node
- $E = \{ (n_i, n_j) \mid$ syntactically n_j is directly invoked by $n_i \}$
- Does not represent the order entities are invoked
- Does not represent the number of times an entity is invoked
- A cycle in G indicates that the nodes along the cycle syntactically participate in a recursive calling chain
- Provides a framework for inter-component analysis

Call Graph Example

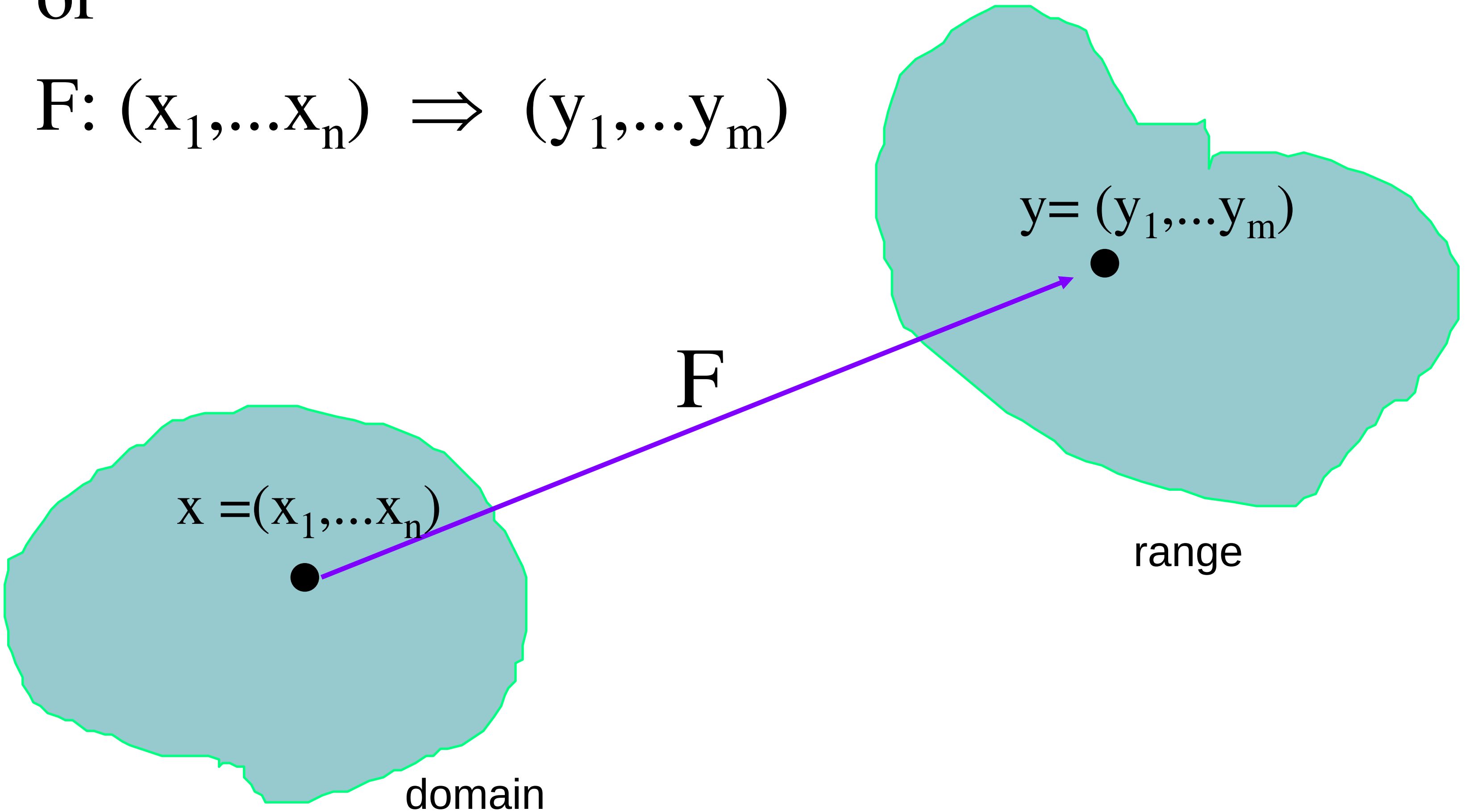


Functional Representation of an Executable Component

$$F: X \Rightarrow Y$$

or

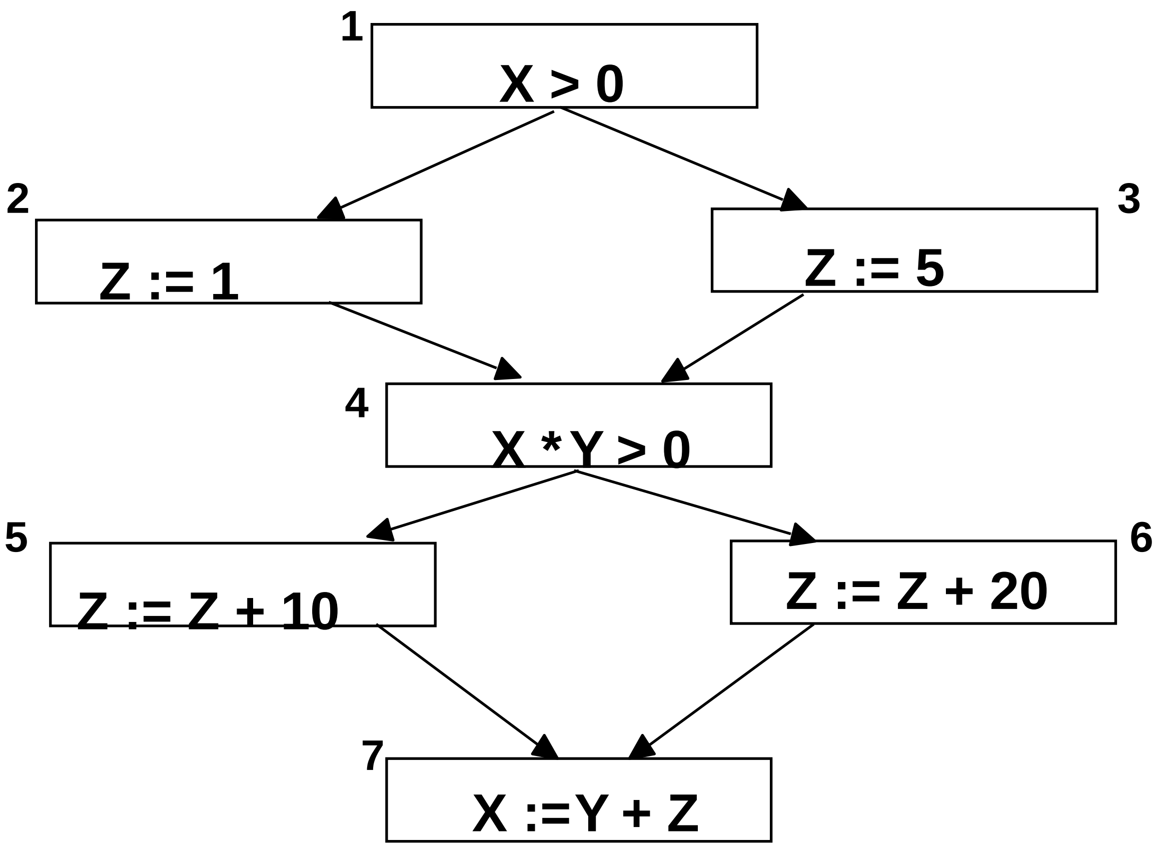
$$F: (x_1,\ldots x_n) \Rightarrow (y_1,\ldots y_m)$$



Combining the functional and graph view of a component

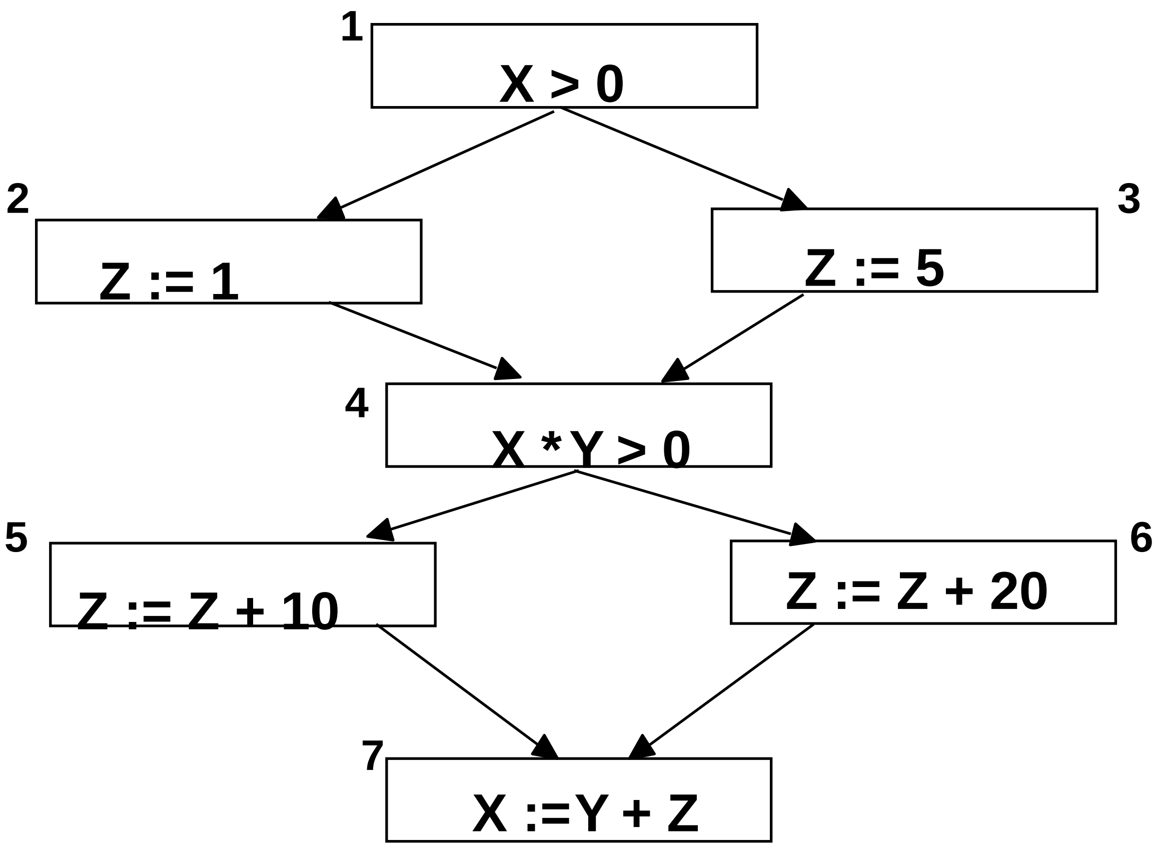
- $F : X \rightarrow Y$
- F is composed of partial functions, where each partial function corresponds to a path in a program
- $F = \{ f_1, f_2, \ldots f_r \}$, where $f_i : X_i \rightarrow Y_i$
- $X = X_1 \cup \ldots \cup X_r$ (r could be ∞)
- $X_i \cap X_j = \varnothing,\ i \neq j$ (deterministic)

Paths



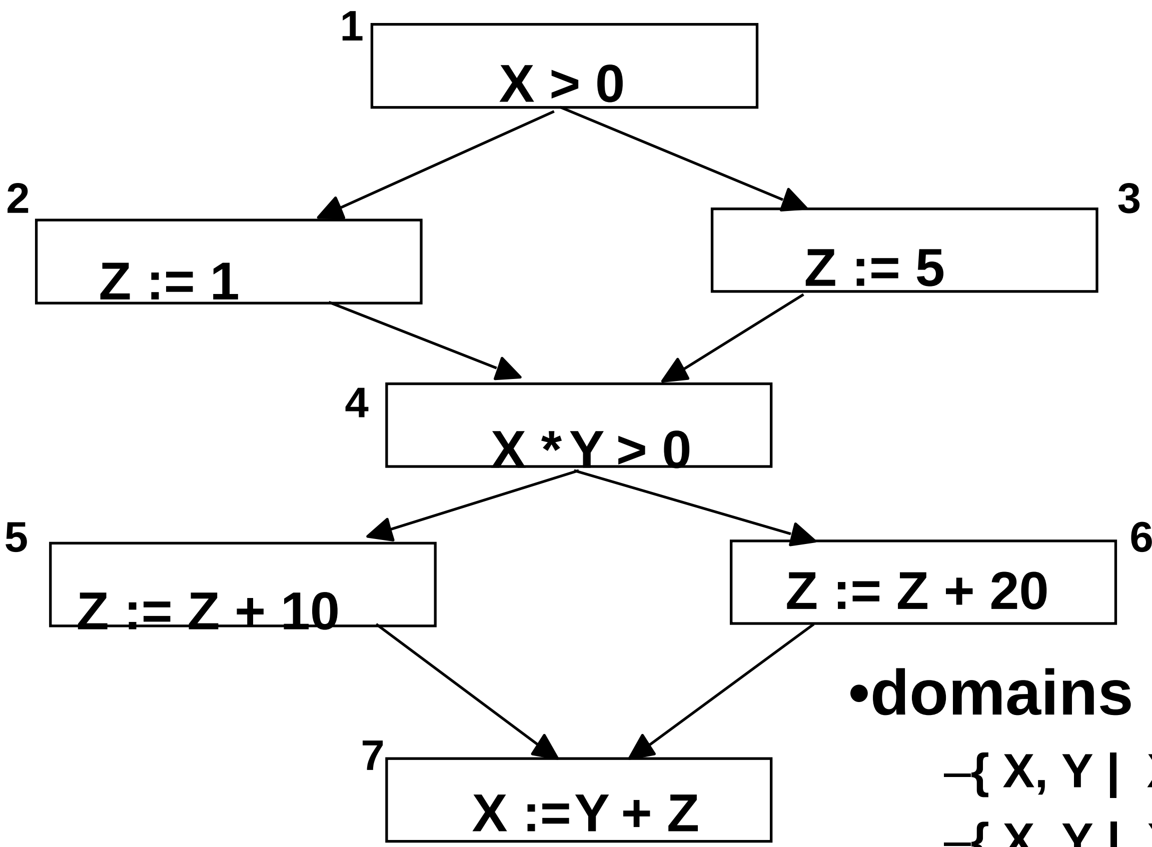
- Paths:
  - 1, 2, 4, 5, 7
  - 1, 2, 4, 6, 7
  - 1, 3, 4, 5, 7
  - 1, 3, 4, 6, 7

Paths can be identified by predicate outcomes



- outcomes
  - t, t
  - t, f
  - f, t
  - f, f

Paths can be identified by domains



- domains
  - { X, Y | X > 0 and X * Y > 0}
  - { X, Y | X > 0 and X * Y < = 0 }
  - { X, Y | X < = 0 and X * Y > 0}
  - { X, Y | X < = 0 and X * Y < = 0}

Common Definitions

- Failure-- result that deviates from the expected or specified intent
- Fault/defect-- a flaw that could cause a failure
- Error -- erroneous belief that might have led to a flaw that could result in a failure
- Static Analysis -- the static examination of a product or a representation of the product for the purpose of inferring properties or characteristics
- Dynamic Analysis -- the execution of a product or representation of a product for the purpose of inferring properties or characteristics
- Testing -- the (systematic) selection and subsequent "execution" of sample inputs from a product's input space in order to infer information about the product's behavior.
  - usually trying to uncover failures
  - the most common form of dynamic analysis
- Debugging -- the search for the cause of a failure and subsequent repair

Validation and Verification: V&V

- Validation -- techniques for assessing the quality of a software product
- Verification -- the use of analytic inference to (formally) prove that a product is consistent with a specification of its intent
  - the specification could be a selected property of interest or it could be a specification of all expected behaviors and qualities

    e.g., all deposit transactions for an individual will be completed before any withdrawal transaction will be initiated
  - a form of validation
  - usually achieved via some form of static analysis

Correctness

- a product is correct if it satisfies all the requirement specifications
  - correctness is a mathematical property
  - requires a specification of intent
  - specifications are rarely complete
  - difficult to prove poorly-quantified qualities such as user-friendly
- a product is behaviorally or functionally correct if it satisfies all the specified behavioral requirements

Reliability

- measures the dependability of a product
  - the probability that a product will perform as expected
  - sometimes stated as a property of time
    e.g., mean time to failure
- Reliability vs. Correctness
  - reliability is relative, while correctness is absolute

  (but only wrt a specification)
  - given a "correct" specification, a correct product is reliable, but not necessarily vice versa

Robustness

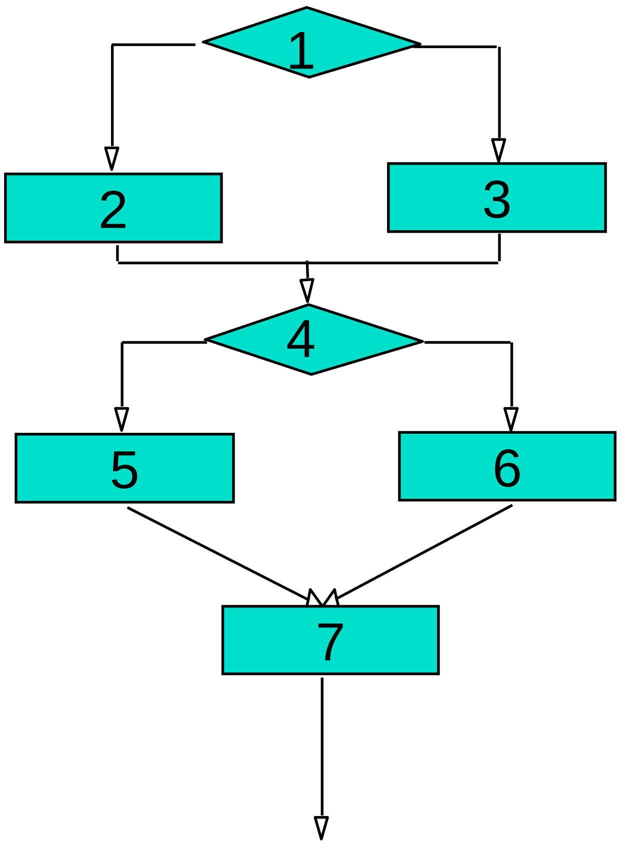
- behaves "reasonably" even in circumstances that were not expected
  - making a system robust more than doubles development costs
  - a system that is correct may not be robust, and vice versa

Static analysis techniques usually try to be conservative

- never declare a property to be valid if it is not
- Usually achieve this by using representations that over-estimate actual behavior
- The representation depends on the analysis
  - AST is a conservative representation for
    - Determining all the operators in a program
    - Determining all the locations where X is defined
  - CFG is a conservative representation for
    - Determining how many loops are in the program
    - Determining how deeply nested each loop is

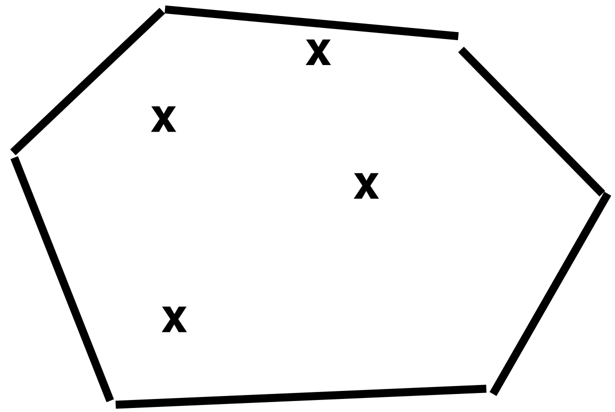
Conservative analysis when considering paths

- For all execution sequences, is P true?
  - if P is true for all paths, then P is true
  - if P is true for some paths, then P may be true or false
    - Paths where P is not true may not be feasible
- For some execution sequence, is P true?
  - If P is true for some path, P may be true or false
    - the path where P is true may or may not be feasible



Conservative static analysis would only say P is true if it is known to be true for all paths

Dynamic analysis techniques draw inferences from a sample of the problem domain



How do we choose that sample?

Static analysis

- Tries to find errors in the system
  - Conservative=>too many false positives?
    - Over reporting
  - Too precise=> too expensive?
  - Not conservative => effective enough?
    - Under reporting