Foundations: Syntax, Semantics, and Graphs

Testing, Quality Assurance, and Maintenance
Winter 2017

Prof. Arie Gurfinkel

based on slides by Ruzica Pizkac, Claire Le Goues, Lin Tan, Marsha Chechik, and others

UNIVERSITY OF WATERLOO

Foundations

Syntax

- Syntax and BNF Grammar
- Abstract Syntax Trees (AST)

Semantics

- Natural Operational Semantics (a.k.a. big step)
- Judgements and derivations

Graphs

- Graph, cyclic, acyclic
- Nodes, edges, paths
- Trees, sub-graphs, sub-paths, …
- Control Flow Graph (CFG)

SYNTAX

WHILE: A Simple Imperative Language

We will use a simple imperative language called WHILE

- *the language is also sometimes called IMP*

An example WHILE program:

```
{ p := 0; x := 1; n := 2 };
while x ≤ n do {
      x := x + 1;
      p := p + m
} ;
print_state
```

‘;’ is a connective, not terminator as in C!

WHILE: Syntactic Entities

$n \in Z$	– integers	Terminal
true,false $\in$ B	– Booleans	
$x,y \in L$	– locations (program variables)	
$e \in Aexp$	– arithmetic expressions	Non-terminal
$b \in Bexp$	– Boolean expressions	
$c \in Stmt$	– statements	

Terminals are atomic entities that are completely defined by their tokens

- integers, Booleans, and locations are terminals

Non-Terminals are composed of of one or more terminals

- determined by rules of the **grammar**
- Aexp, Bexp, and Stmt are non-terminals

WHILE: Syntax of Arithmetic Expressions

Arithmetic expressions (Aexp)

e ::=	n	for $n \in Z$
	| -n	for $n \in Z$
	| x	for $x \in L$
	| e_1 aop e_2	
	| '(' e ')'	

BNF grammar rules

aop := '*' | '/' | '-' | '+'

Notes:

- Variables are not declared before use
- All variables have integer type
- Expressions have no side-effects

BNF: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

WHILE: Syntax of Boolean Expressions

Boolean expressions (Bexp)

b ::= ‘true’
| ‘false’
| ¬ b
| e_1 rop e_2 for e_1, $e_2 \in$ Aexp
| e_1 bop b_2 for e_1, $e_2 \in$ Bexp
| ‘(‘ b ‘)’

rop ::= ‘<’ | ‘<=‘ | ‘=‘ | ’>=‘ | ‘>’

bop ::= ‘and’ | ‘or’

Syntax of Statements

Statements

```
s ::=       skip
          | x := e
          | if b then s [ else s ]
          | while b do s
          | '{' slist '}'
          | print_state
          | assert b | assume b | havoc v1, …, vN
slist ::= s ( ';' s )*
prog ::= slist
```

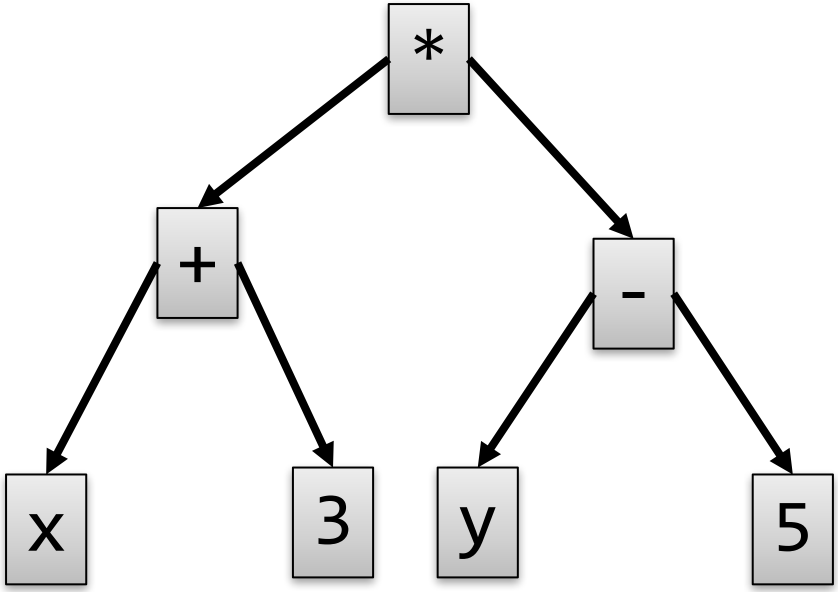
Notes:

- Semi-colon ‘;’ is a statement composition, not statement terminator!!!
- Statements contain all the side-effects in the language
- Many usual features of a PL are missing: references, function calls, …
  - the language is very very simple yet hard to analyze

Abstract Syntax Tree (AST)

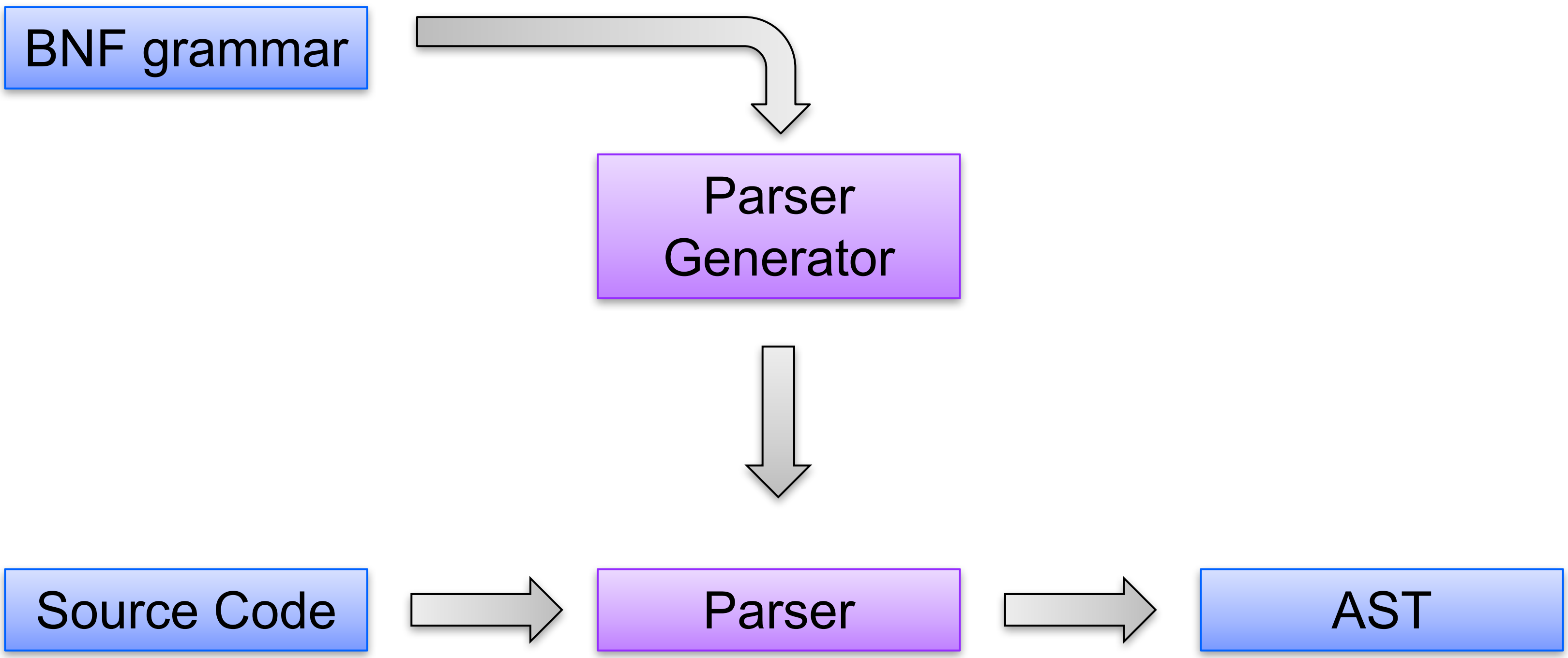
AST is an abstract tree representation of the source code

- each node represents a syntactic construct occurring in the code
  - statement, variable, operator, statement list
- called "***abstract***" because some details of concrete syntax are omitted
  - AST normalizes (provides common representation) of irrelevant differences in syntax (e.g., white space, comments, order of operations)
- example AST: `(x + 3) * (y - 5)`



UNIVERSITY OF WATERLOO

Language Parsing in a Nutshell



Parser generator

- input: BNF grammar; output: parser (program)

Parser

- input: program source code; output: AST or error

WHILE AST in Python

One class per syntactic entity

One field per child

Class hierarchy corresponds to the semantic one

Emacs-x86_64-10_9 Prelude - /tmp/ast.py

```python
class Ast(object):
class Ast(object):
    """Base class of AST hierarchy"""
    pass

class Stmt (Ast):
    """A single statement"""
    pass

class AsgnStmt (Stmt):
    """An assignment statement"""
    def __init__(self, lhs, rhs):
        self.lhs = lhs
        self.rhs = rhs

class IfStmt (Stmt):
    """If-then-else statement"""
    def __init__ (self, cond, then_stmt, else_stmt=None):
        self.cond = cond
        self.then_stmt = then_stmt
        self.else_stmt = else_stmt
```

Behavior Pattern: Visitor

Applicability

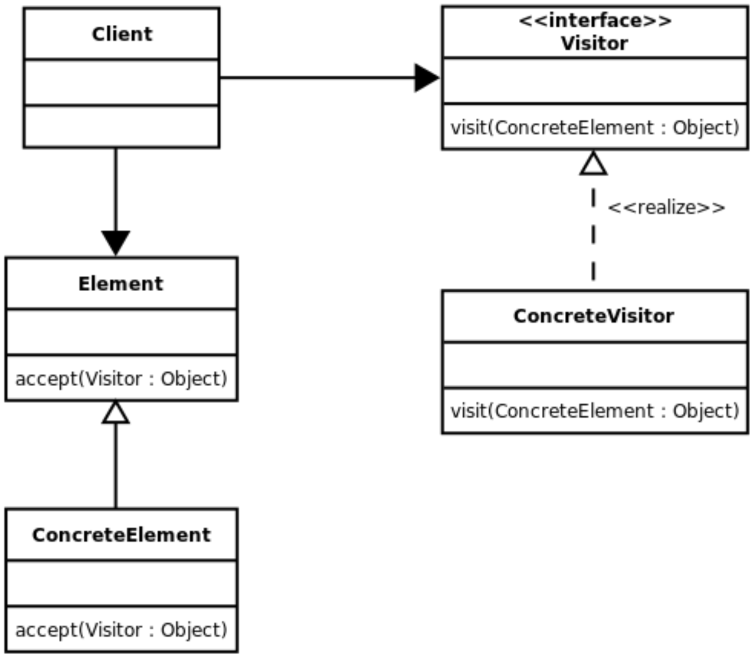
- Object hierarchy with many classes
- Operations depend on classes
- Set of classes is stable
- Want to define new operations

Consequences

- Simplifies adding new operations
- Groups related behavior in one class
- Extending class hierarchy is difficult
- Visitor can maintain state
- **Element** must expose interface

In Python

- Method name is used instead of polymorphism, e.g., `visit_Stmt()`
- Visitor's `visit()` method dispatches calls based on reflection. No need for `accept()`



Example Visitor in Python

Emacs-x86_64-10_9 Prelude - /tmp/ast.py

```
class AstVisitor(object):
```

```python
class AstVisitor(object):
    """Base class for AST visitor"""
    def __init__(self):
        pass

    def visit (self, node, *args, **kwargs):
        """Visit a node."""
        method = 'visit_' + node.__class__.__name__
        visitor = getattr (self, method)
        return visitor (node, *args, **kwargs)

    def visit_BoolConst (self, node, *args, **kwargs):
        visitor = getattr (self, 'visit_' + Const.__name__)
        return visitor (node, *args, **kwargs)
```

```python
class PrintVisitor (AstVisitor):
    """A printing visitor"""
    def visit_IntVar (self, node, *args, **kwargs):
        self._write (node.name)

    def visit_Const (self, node, *args, **kwargs):
        self._write (node.val)

    def visit_Exp (self, node, *args, **kwargs):
        if node.is_unary ():
            self._write (node.op)
            self.visit (node.arg (0))
        else:
            self._open_brkt (**kwargs)
            self.visit (node.arg (0))
            for a in node.args [1:]:
                self._write (' ')
                self._write (node.op)
                self._write (' ')
                self.visit (a)
            self._close_brkt (**kwargs)
```

Exercise: Implement a state counting visitor

Write a visitor that counts the number of statements in a program

- (a) Implementation 1:
  - the visitor should be stateless and return the number of statements

- (b) Implementation 2:
  - the uses an internal state (field) to keep track of the number of statements

Stateless Visitor

```
class StmtCounterStateless (wlang.ast.AstVisitor):
    def __init__ (self):
        super (StmtCounterStateless, self).__init__ ()

    def visit_StmtList (self, node, *args, **kwargs):
        if node.stmts is None:
            return 0
        res = 0
        for s in node.stmts:
            res = res + self.visit (s)
        return res

    def visit_IfStmt (self, node, *args, **kwargs):
        res = 1 + self.visit (node.then_stmt)
        if node.has_else ():
            res = res + self.visit (node.else_stmt)
        return res

    def visit_WhileStmt (self, node, *args, **kwargs):
        return 1 + self.visit (node.body)

    def visit_Stmt (self, node, *args, **kwargs):
        return 1
```

UNIVERSITY OF WATERLOO

Statefull Visitor

```
class StmtCounterStatefull (wlang.ast.AstVisitor):
    def __init__ (self):
        super (StmtCounterStatefull, self).__init__ ()
        self._count = 0

    def get_num_stmts (self):
        return self._count

    def count (self, node, *args, **kwargs):
        self._count = 0
        self.visit (node, *args, **kwargs)

    def visit_StmtList (self, node, *args, **kwargs):
        if node.stmts is None:
            return
        for s in node.stmts:
            self.visit (s)

    def visit_Stmt (self, node, *args, **kwargs):
        self._count = self._count + 1
    def visit_IfStmt (self, node, *args, **kwargs):
        self.visit_Stmt (node)
        self.visit (node.then_stmt)
        if node.has_else ():
            self.visit (node.else_stmt)

    def visit_WhileStmt (self, node, *args, **kwargs):
        self.visit_Stmt (node)
        self.visit (node.body)
```

From Programming to Modeling

Extend a programming language with 3 modeling features

Assertions

- assert(e) – aborts an execution when e is false, no-op otherwise

```
void assert (bool b) { if (!b)  error(); }
```

Non-determinism

- nondet_int() – returns a non-deterministic integer value

```
int nondet_int () { int x; return x; }
```

Assumptions

- assume(e) – "ignores" execution when e is false, no-op otherwise

```
void assume (bool e) { while (!e) ;  }
```

Safety Specifications as Assertions

A program is correct if all executions that satisfy all assumptions also satisfy all assertions

A program is incorrect if there exists an execution that violates an assertion, but satisfies all of the assumptions

Assumptions are used to expressed pre-conditions on which the program behavior relies

Assertions are used to expressed desired post-conditions that the program must maintain

Non-determinism vs. Randomness

A *deterministic* function always returns the same result on the same input

- e.g., F(5) = 10

A *non-deterministic* function may return different values on the same input

- e.g., G(5) in [0, 10] "G(5) returns a non-deterministic value between 0 and 10"

A *random* function may choose a different value with a probability distribution

- e.g., H(5) = (3 with prob. 0.3, 4 with prob. 0.2, and 5 with prob. 0.5)

Non-deterministic choice cannot be implemented!

- used to model the worst possible adversary/environment

Modeling with Non-determinism

```
int x, y;

void main (void)
{
  x = nondet_int ();
  assume (x > 10);
  assume (x <= 100);

  y = x + 1;

  assert (y > x);
  assert (y < 200);

}
```

```
int x, y;

void main (void)
{
  havoc (x);
  assume (x > 10);
  assume (x <= 100);

  y = x + 1;

  assert (y > x);
  assert (y < 200);

}
```

WHILE language uses **havoc** statement instead of **nondet_int()** function!

Order of Assumptions

```
int x, y;

void main (void)
{
  x = nondet_int ();

  y = x + 1;

  assume (x > 10);
  assume (x <= 100);

  assert (y > x);
  assert (y < 200);
}
```

```
int x, y;

void main (void)
{
  x = nondet_int ();

  y = x + 1;

  assert (y > x);
  assert (y < 200);

  assume (x > 10);
  assume (x <= 100);
}
```

UNIVERSITY OF WATERLOO

Dangers of unrestricted assumptions

Assumptions can lead to vacuous correctness claims!!!

Is this program correct?

```
if (x > 0) {
  assume (x < 0);
  assert (0); }
```

Assume must either be checked with assert or used as an idiom:

```
x = nondet_int ();
y = nondet_int ();
assume (x < y);
```

SEMANTICS

Announcements

Tentative midterm schedule

Monday, February 13, 5:00-6:30pm, RCH 105

Let me know ASAP if you have a conflict with this time!

Guest Lecture by Prof. Vijay Ganesh on Symbolic Execution

Friday, January 20th during class

Meaning of WHILE Programs

Questions to answer:

- What is the “meaning” of a given WHILE expression/statement?
- How would we evaluate WHILE expressions and statements?
- How are the evaluator and the meaning related?
- How can we reason about the effect of a command?

Semantics of Programming Languages

Denotational Semantics

- Meaning of a program is defined as the mathematical object it computes (e.g., partial functions).
- example: Abstract Interpretation

Axiomatic Semantics

- Meaning of a program is defined in terms of its effect on the truth of logical assertions.
- example: Hoare Logic

(Structural) Operational Semantics

- Meaning of a program is defined by formalizing the individual computation steps of the program.
- example: Natural Operational Semantics

Semantics of WHILE

The meaning of WHILE expressions depends on the values of variables, i.e. the current state.

A state *s* is a function from L to

- assigns a value for every location/variable
- *s(x)* is the value of variable *x* in state *s*

The set of all states is $Q = L \rightarrow Z$

We use *q* to range over Q

Judgments

We write <e, q> $\Downarrow$ n to mean that expression *e* evaluates to *n* in state *q*.

- The formula <e, q> $\Downarrow$ n is called a judgment
  (a judgement is a relation between an expression *e*, a state *q* and a number *n*)
- We can view $\Downarrow$ as a function of two arguments *e* and *q*

This formulation is called natural operational semantics

- also known as *big-step* operational semantics
- the judgment relates the expression and its “meaning”

How to define <e1 + e2, q> $\Downarrow$ … ?

Inference Rules

We express the evaluation rules as inference rules for our judgments.

The rules are also called evaluation rules.

An inference rule

$$\frac{F_1 \; \ldots \; F_n}{G} \quad \text{where } H$$

defines a relation between judgments $F_1,\ldots,F_n$ and G.

- The judgments $F_1,\ldots,F_n$ are the premises of the rule;
- The judgments G is the conclusion of the rule;
- The formula H is called the side condition of the rule.

If n=0 the rule is called an axiom. In this case, the line separating premises and conclusion may be omitted.

Inference Rules for Aexp

In general, we have one rule per language construct:

$$\frac{}{\langle n, q\rangle \Downarrow n} \qquad\qquad \frac{}{\langle x, q\rangle \Downarrow q(x)}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 + e_2, q\rangle \Downarrow (n_1 + n_2)} \qquad\qquad \frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 - e_2, q\rangle \Downarrow (n_1 - n_2)}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 * e_2, q\rangle \Downarrow (n_1 * n_2)}$$

This is called structural operational semantics.

- rules are defined based on the structure of the expressions.

Inference Rules for *Bexp*

$$\frac{}{\langle \mathit{true}, q\rangle \Downarrow \text{true}}$$

$$\frac{}{\langle \text{false}, q\rangle \Downarrow \text{false}}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 = e_2, q\rangle \Downarrow (n_1 = n_2)}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \leq e_2, q\rangle \Downarrow (n_1 \leq n_2)}$$

$$\frac{\langle b_1, q\rangle \Downarrow t_1 \qquad \langle e_2, q\rangle \Downarrow t_2}{\langle b_1 \wedge b_2, q\rangle \Downarrow (t_1 \wedge t_2)}$$

Derivation

Derivation is a well-formed application of inference rules

- Derivation infers new facts from existing ones

$$\frac{\langle 5, q\rangle \Downarrow 5 \qquad \dfrac{\langle 7, q\rangle \Downarrow 7 \qquad \langle 2, q\rangle \Downarrow 2}{\langle 7*2, q\rangle \Downarrow 14}}{\langle 5 + (7*2), q\rangle \Downarrow 19}$$

Evaluation of Statements

Evaluation of a statement produces a side-effect

- The *result* of evaluation of a statement is a new state

We write <s, q> ⇓ q' to mean that evaluation of statement *s* in state *q* results in a new state *q'*

$$\frac{}{\langle \text{skip}, q\rangle \Downarrow q} \qquad \frac{}{\langle \text{print\_state}, q\rangle \Downarrow q}$$

$$\frac{\langle s_1, q\rangle \Downarrow q'' \qquad \langle s_2, q''\rangle \Downarrow q'}{\langle s_1 ; s_2, q\rangle \Downarrow q'} \qquad \frac{\langle b, q\rangle \Downarrow true \qquad \langle s_1, q\rangle \Downarrow q'}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2, q\rangle \Downarrow q'}$$

$$\frac{\langle e, q\rangle \Downarrow n}{\langle x := e, q\rangle \Downarrow q[x:=n]} \qquad \frac{\langle b, q\rangle \Downarrow false \qquad \langle s_2, q\rangle \Downarrow q'}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2, q\rangle \Downarrow q'}$$

Derivation and Execution

Derivation of statement facts corresponds to execution / interpretation

For example

- Show that $\langle p:=0; x:=1; n:=2, [\ ]\rangle \Downarrow [p:=0,x:=1,n:=2]$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 0, []\rangle \Downarrow 0}}{\langle p:=0, []\rangle \Downarrow [p:=0]} \qquad \dfrac{\dfrac{}{\langle 1, [p:=0]\rangle \Downarrow 1}}{\langle x:=1, [p:=0]\rangle \Downarrow [p:=0,x:=1]}}{\langle p:=0,x:=1, [\ ]\rangle \Downarrow [p:=0,x:=1]} \qquad \langle n:=2, [p:=0,x:=1]\rangle \Downarrow [p:=0,x:=1,n:=2]}{\langle p:=0; x:=1; n:=2, [\ ]\rangle \Downarrow [p:=0,x:=1,n:=2]}
$$

Semantics of Loops

$$\frac{\langle b, q\rangle \Downarrow \mathit{false}}{\langle \text{while b do s}, q\rangle \Downarrow \text{q}} \qquad \frac{\langle b, q\rangle \Downarrow \mathit{true} \quad \langle s\,;\text{ while b do s}, q\rangle \Downarrow q'}{\langle \text{while b do } s, q\rangle \Downarrow \text{q}'}$$

What about infinite execution?

- Can introduce a special state $\top$, called *top*, that represents divergence
- Infinite loop enters divergent state

$$\frac{}{\langle \text{while true do } s, q\rangle \Downarrow \top}$$

- Any statement in divergent state is treated like 'skip'

$$\frac{}{\langle \text{s}, \top\rangle \Downarrow \top}$$

Need *small step* semantics to deal with reactive execution

- execution that does not terminate, but produces useful result

GRAPHS

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 programs

Paths

a path, P, through a directed graph $G = (N, E)$ is a sequence of edges, $((u_1, v_1), (u_2, v_2), \ldots (u_t, v_t))$ such that

- $v_{k-1} = u_k$ for all $1 < k \leq t$
- u_1 is called the start node and v_t is called the end node

The length of a path is the number of edges (or nodes-1 ☺) in the path

Paths are also frequently represented by a sequence of nodes

- $(u_1,u_2,u_3,\ldots,u_t)$

Cycles

A cycle in a graph G is a path whose start node and end node are the same

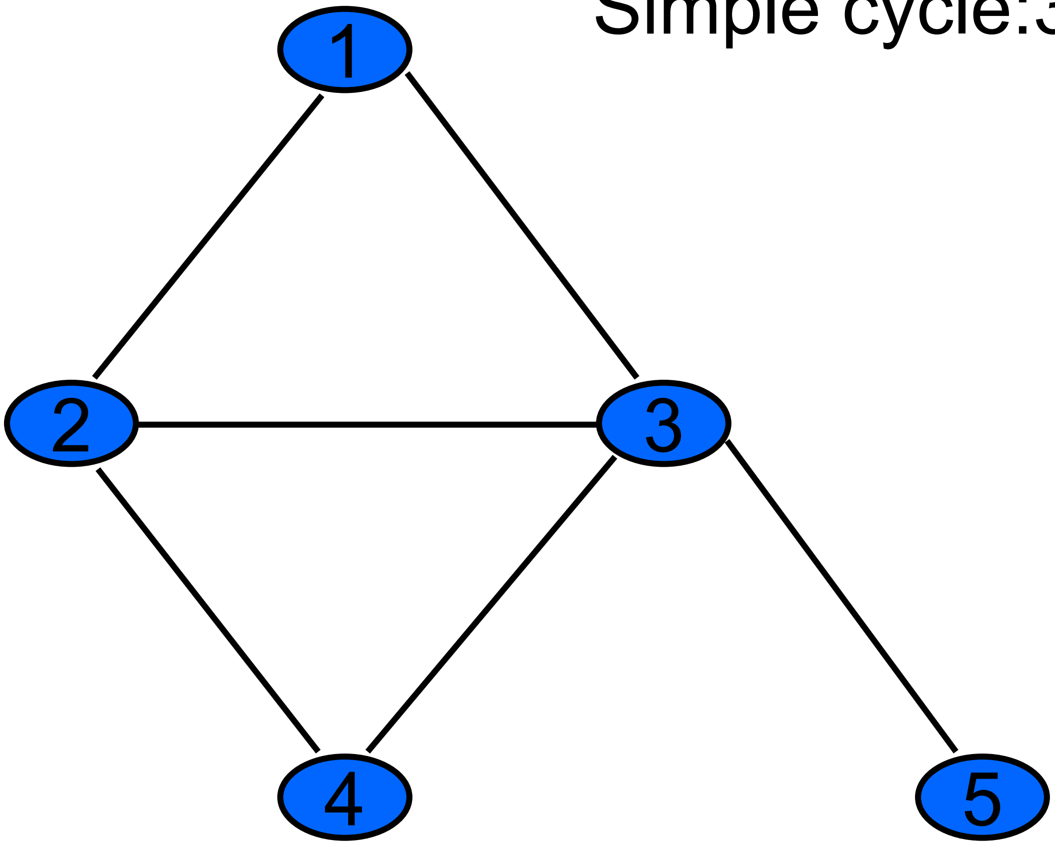
A simple cycle in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes)

If a graph G has no path through it that is a cycle, then the graph is called acyclic

Example of Cycles

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

Simple cycle:3,2,4,3



Trees

An acyclic, undirected graph is called a tree

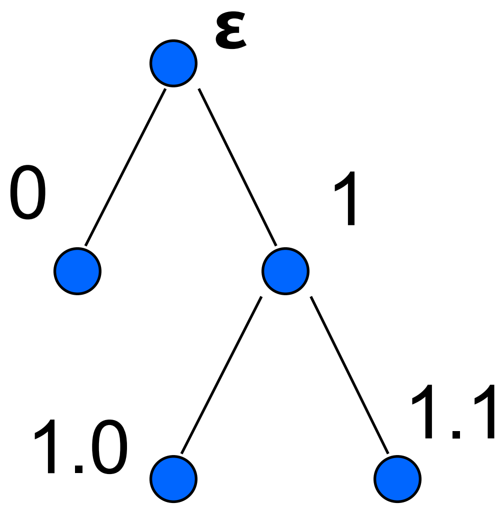
If the undirected version of a directed graph is acyclic, then the graph is called a directed tree

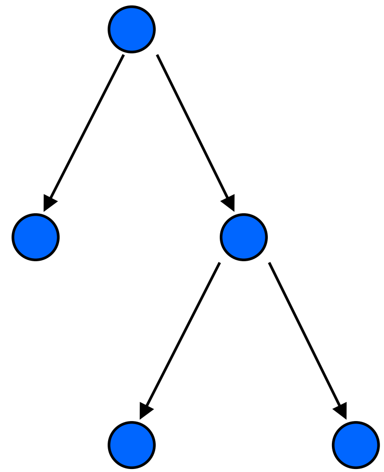
If the undirected version of a directed graph has cycles, but the directed graph itself has no cycles, then the graph is called a Directed Acyclic Graph (DAG)

Every tree is isomorphic to a prefix-closed subset of N^* for some natural number N

Examples

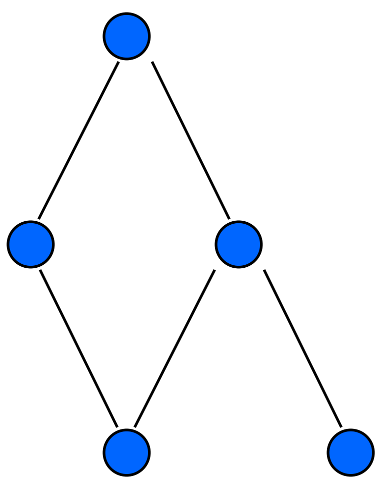
tree

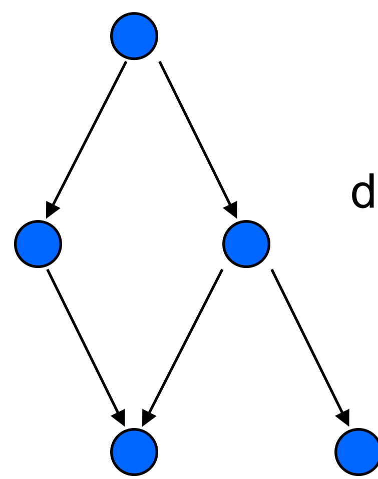




directed tree

cyclic undirected graph





directed acyclic graph (DAG)

GRAPHS AS MODELS OF COMPUTATION

Computation tree

A tree model of all the possible executions of a system

At each node represents a state of the system

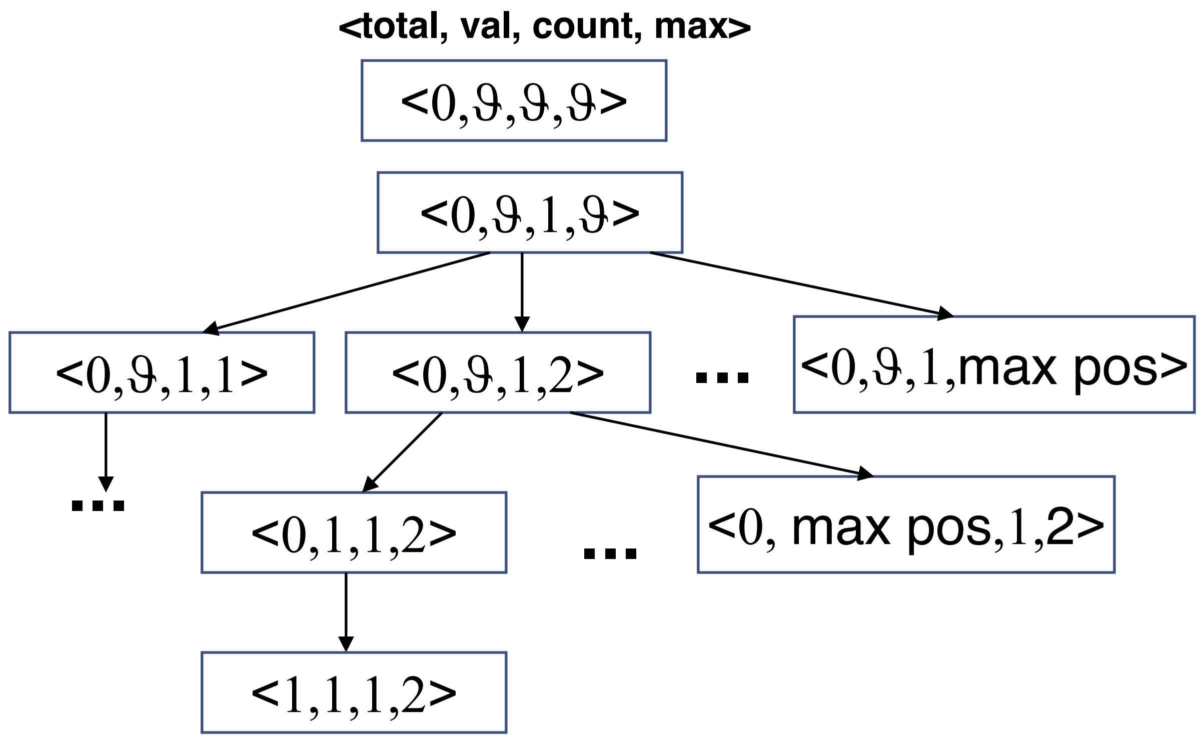
- valuation of all variables

Can have infinite number of paths

Can have infinite paths

Example Computation Tree

```
total := 0;
count := 1;
max := input();
while (count <= max)
 do {
  val := input();
  total := total+val;
  count := count+1};
print (total)
```



Is this tree infinite?

Disadvantages of Computation Trees

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 execution in the program

G = (N, E, S, T) where

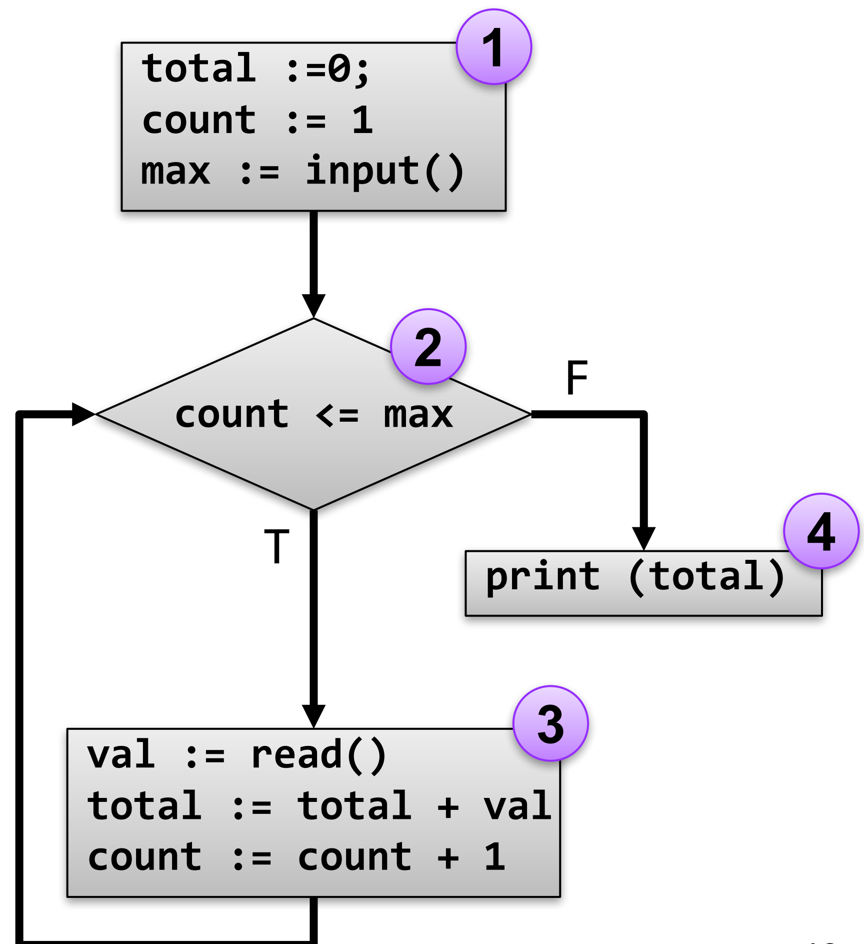
- the nodes N represent executable instructions (statement, statement fragments, or basic blocks);
- the edges E represent the potential transfer of control;
- S is a designated start node;
- T is a designated final node
- E = { (n_i, n_j) | syntactically, the execution of n_j follows the execution of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

Nodes may correspond to single statements, parts of statements, or several statements (i.e., basic blocks)

Execution of a node means that the instructions associated with a node are executed in order from the first instruction to the last

Example of a Control Flow Graph

```
total := 0;
count := 1;
max := input();
while (count <= max)
 do {
  val := input();
  total := total+val;
  count := count+1};
print (total)
```

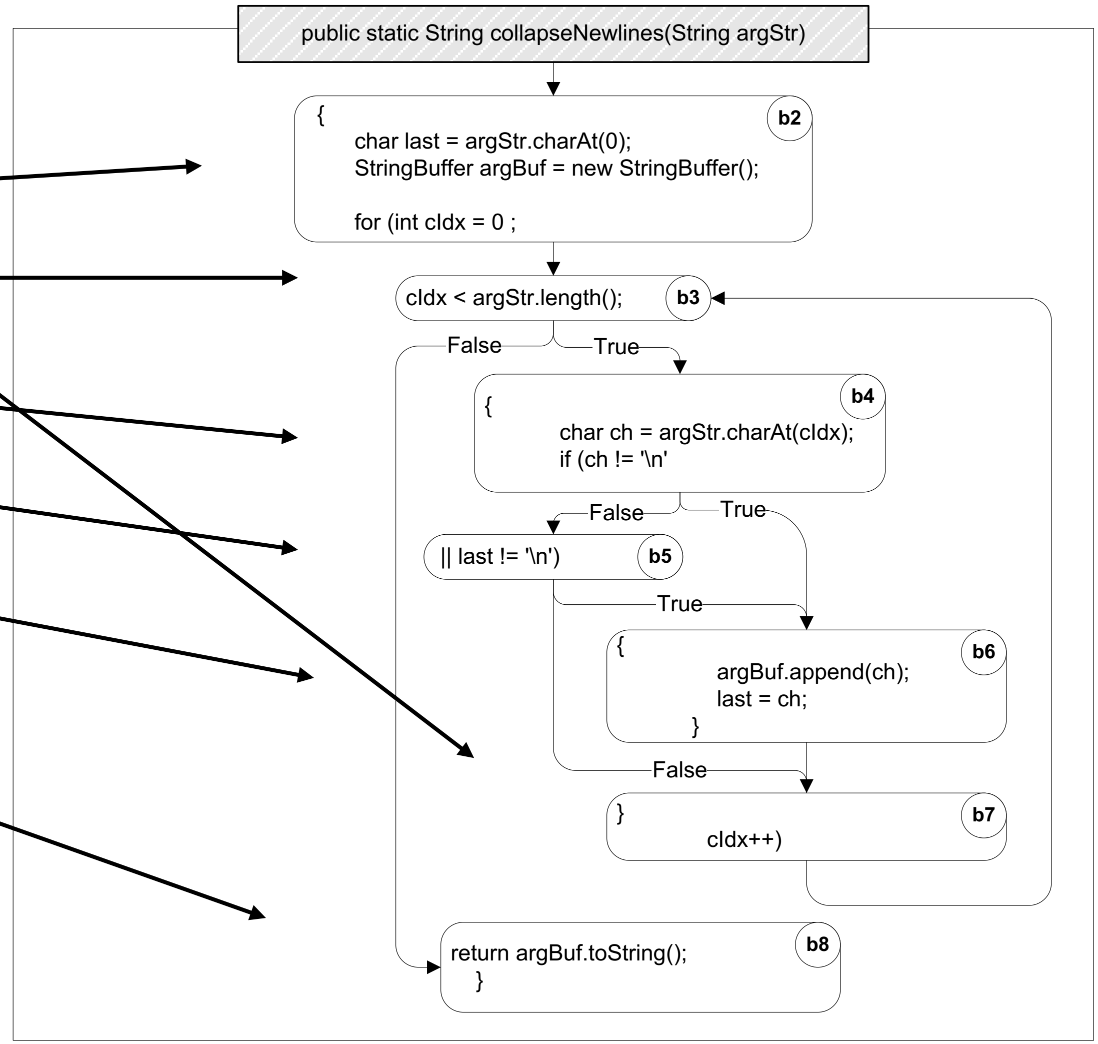


Deriving a Control Flow Graph

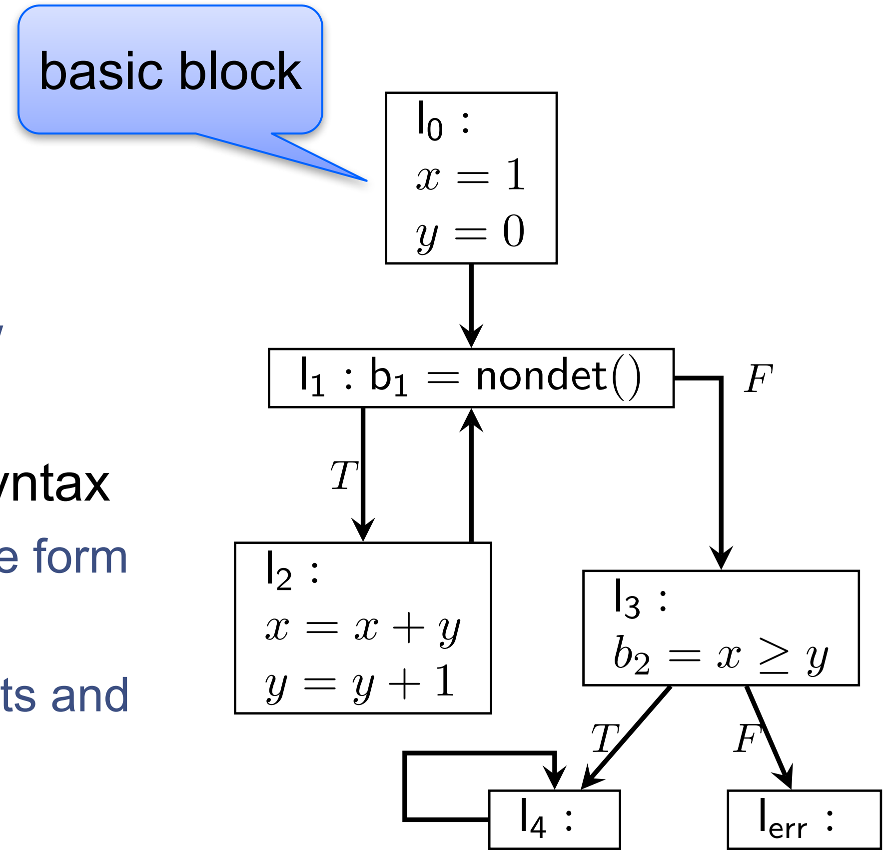
```
public static String collapseNewlines(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}
```



Control Flow Graph



A CFG is a graph of basic blocks

- edges represent different control flow

A CFG corresponds to a program syntax

- where statements are restricted to the form

  L_i:S ; goto L_j

  and S is control-free (i.e., assignments and procedure calls)

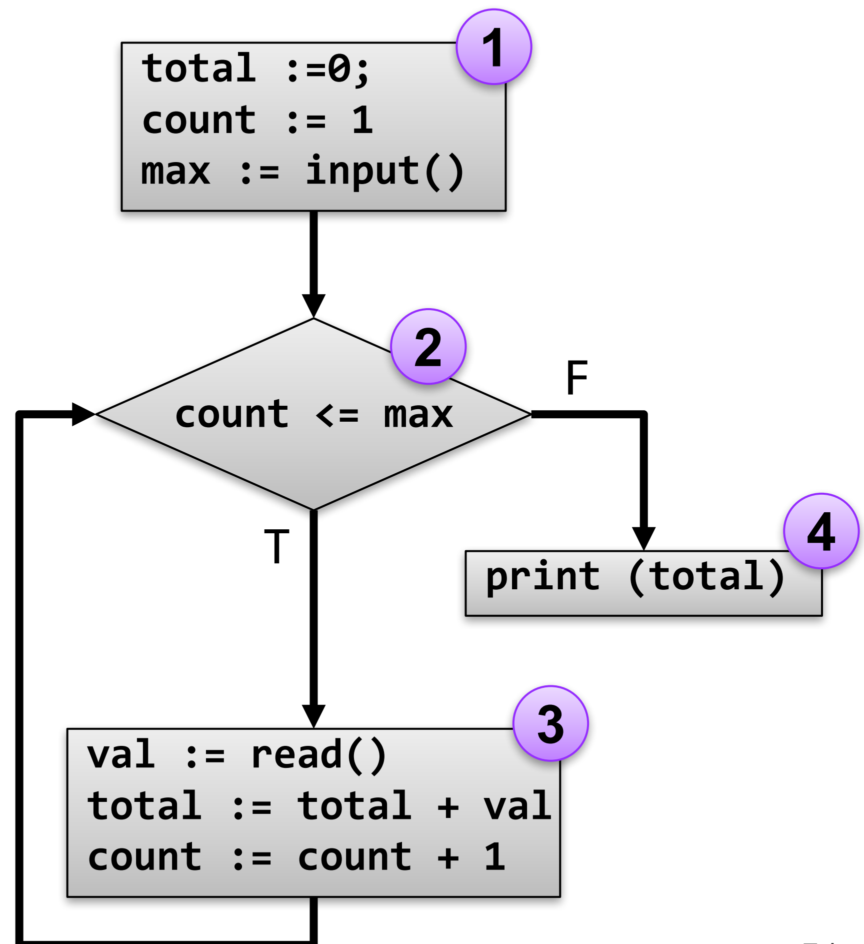
CFG: Sub-path and Complete Path

a sub-path through a CFG is a sequence of nodes (n_i, n_{i+1},...n_t), i ≥1 where for each n_k, i ≤ 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

- e.g., 1, 2, 3, 2, 4



Infeasible Paths

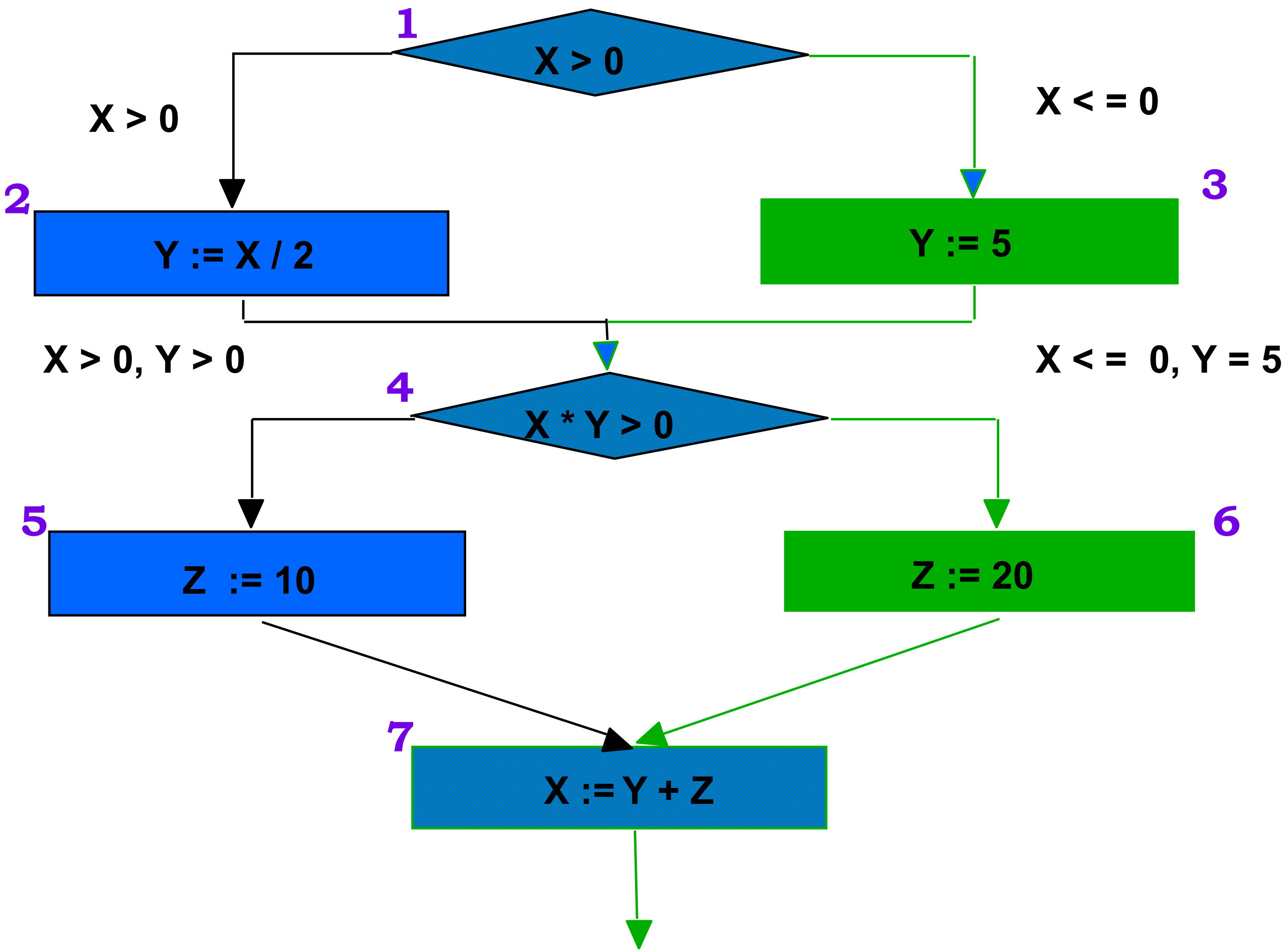
Every executable sequence in the represented component corresponds to a path in G

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 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, undecidable
- In practice, intractable
  - Some exceptions studied in this course

Infeasible paths vs. unreachable vs dead code

unreachable code

- X := X + 1;
- Goto loop;
- $\boxed{\text{Y := Y + 5;}}$

Never executed

dead code

- X := X + 1;
- X := 7;
- $\boxed{\text{X := X + Y;}}$

'Executed', but irrelevant

Benefits of CFG

Probably the most commonly used representation

- Numerous variants

Basis for inter-component analysis

- Collections of CFGs

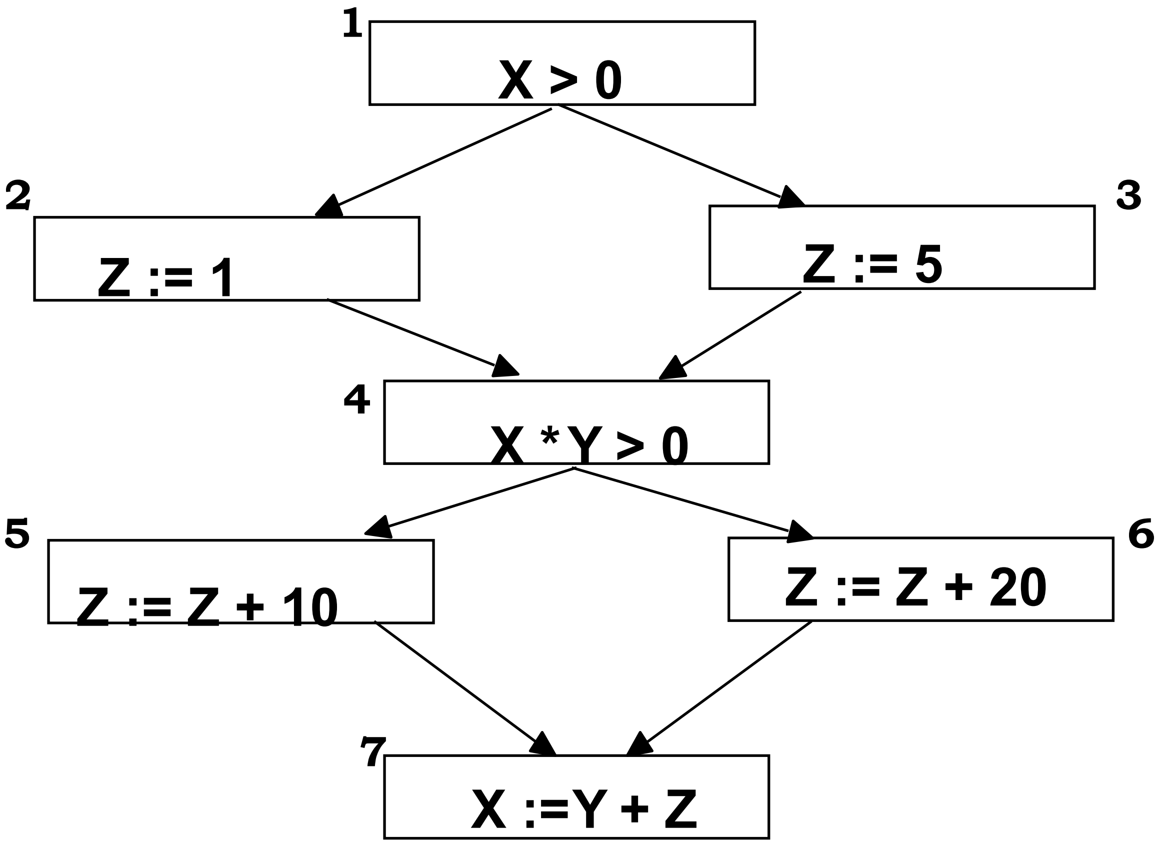
Basis for various transformations

- Compiler optimizations
- S/W analysis

Basis for automated analysis

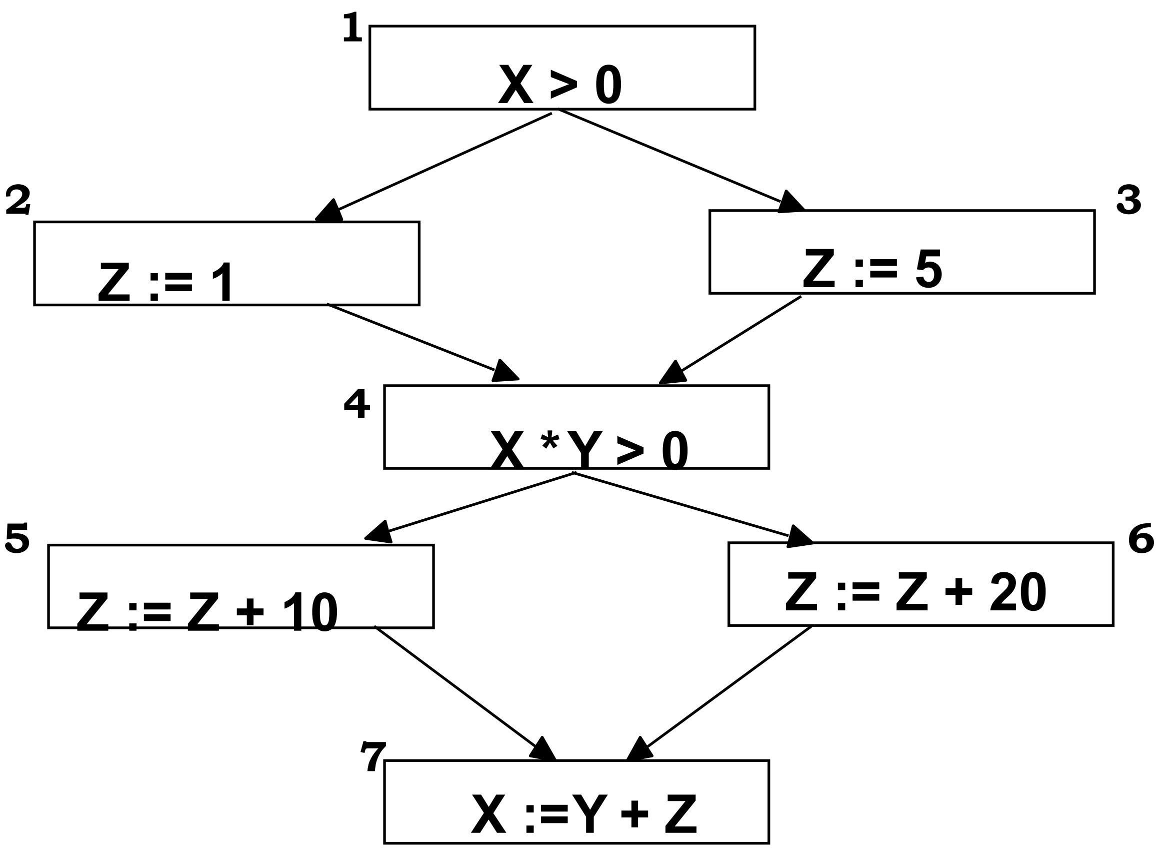
- Graphical representations of interesting programs are too complex for direct human understanding

Paths



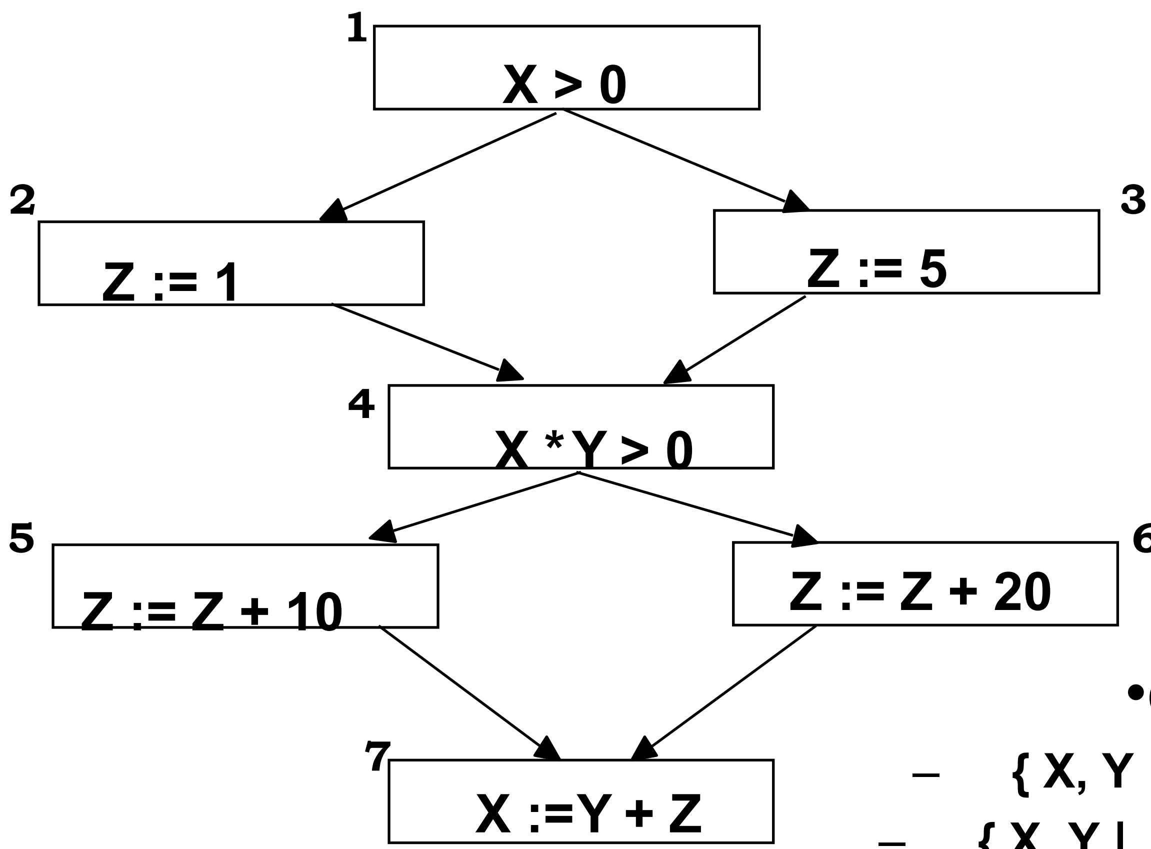
- Paths:
  - 1, 2, 4, 5, 7
  - 1, 2, 4, 6, 7
  - 1, 3, 4, 5, 7
  - 1, 3, 4, 6, 7

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}

CFG Abstraction Level?

Loop conditions? (yes)

Individual statements? (no)

Exception handling? (no)

What's best depends on type of analysis to be conducted

CFG Exercise (1)

Draw a control flow graph with 7 nodes.

```
int binary_search(int a[], int low, int high,
int target) { /* binary search for target in
the sorted a[low, high] */
 1     while (low <= high) {
 2         int middle = low + (high - low)/2;
 3         if (target < a[middle])
 4             high = middle - 1;
 5         else if (target > a[middle])
 6             low = middle + 1;
           else
 7             return middle;
       }
 8   return -1; /* return -1 if target is not
found in a[low, high] */
}
```

CFG Exercise (2)

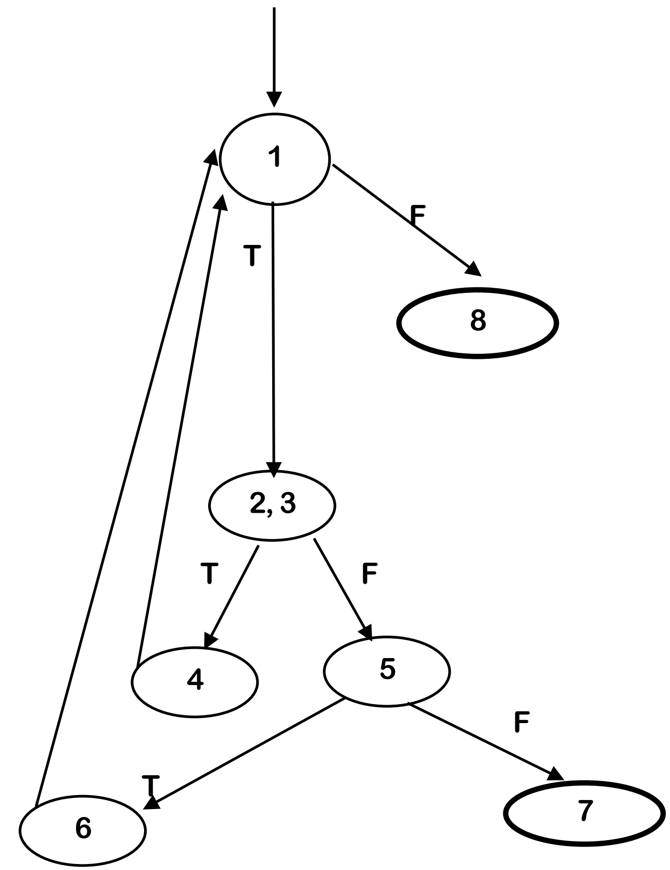
Draw a control flow graph with 8 nodes.

```
int binary_search(int a[], int low, int high,
int target) { /* binary search for target in
the sorted a[low, high] */
  1     while (low <= high) {
  2         int middle = low + (high - low)/2;
  3         if (target < a[middle])
  4             high = middle - 1;
  5         else if (target > a[middle])
  6             low = middle + 1;
            else
  7             return middle;
        }
  8    return -1; /* return -1 if target is not
found in a[low, high] */
}
```

CFG Exercise (1) Solution

Draw a control flow graph with 7 nodes.

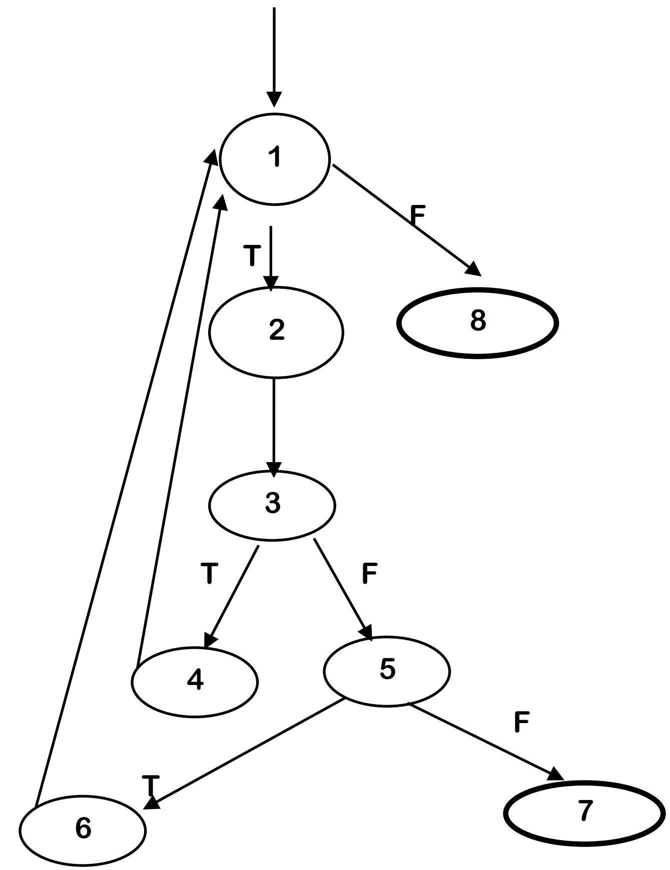
```
int binary_search(int a[], int low, int high,
int target) { /* binary search for target in
the sorted a[low, high] */
  1     while (low <= high) {
  2         int middle = low + (high - low)/2;
  3         if (target < a[middle])
  4             high = middle - 1;
  5         else if (target > a[middle])
  6             low = middle + 1;
            else
  7             return middle;
        }
  8    return -1; /* return -1 if target is not
found in a[low, high] */
}
```



CFG Exercise (2) Solution

Draw a control flow graph with 8 nodes.

```
int binary_search(int a[], int low, int high,
int target) { /* binary search for target in
the sorted a[low, high] */
  1     while (low <= high) {
  2         int middle = low + (high - low)/2;
  3         if (target < a[middle])
  4             high = middle - 1;
  5         else if (target > a[middle])
  6             low = middle + 1;
            else
  7             return middle;
        }
  8    return -1; /* return -1 if target is not
found in a[low, high] */
}
```



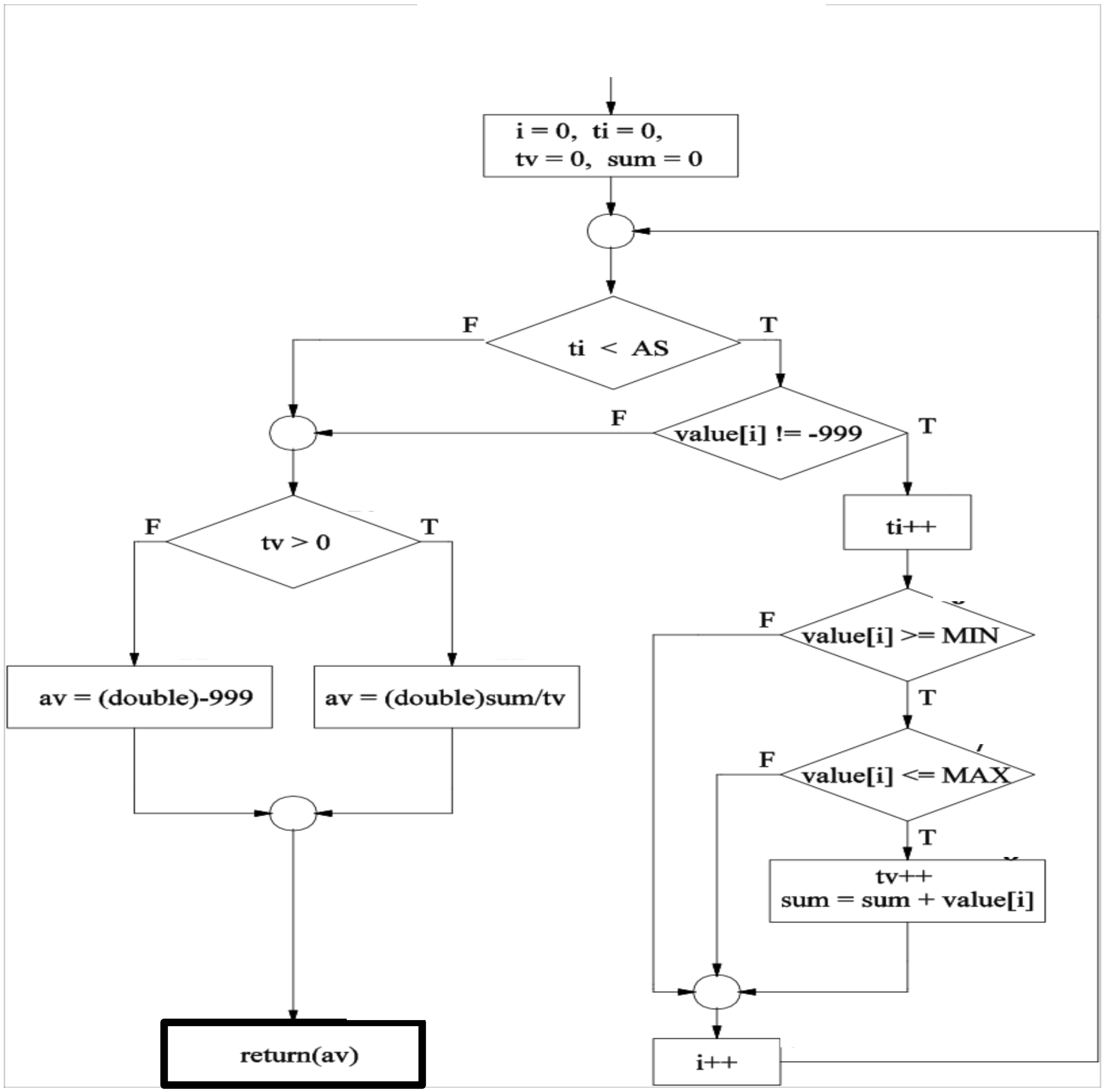
UNIVERSITY OF WATERLOO

CFG Exercise (3)

/ Function: ReturnAverage Computes the average of all those numbers in the input array in the positive range [MIN, MAX]. The max size of the array is AS. But, the array size could be smaller than AS in which case the end of input is designated by -999. */*

```
public static double ReturnAverage(int value[], int AS, int MIN, int MAX) {
        int i, ti, tv, sum;
        double av;
        i = 0; ti = 0; tv = 0; sum = 0;
        while (ti < AS && value[i] != -999) {
                ti++;
                if (value[i] >= MIN && value[i] <= MAX) {
                 tv++;
                 sum = sum + value[i];
                 }
                 i++;
         }
         if (tv > 0) av = (double)sum/tv;
                        else av = (double) -999;
         return (av);
}
```

CFG of ReturnAverage



Single Static Assignment

SSA == every variable has a unique assignment (a *definition*)

A procedure is in SSA form if every variable has exactly one definition

SSA form is used by many compilers

- explicit def-use chains
- simplifies optimizations and improves analyses

PHI-function are necessary to maintain unique definitions in branching control flow

$x = \text{PHI}\ (\ v_0\text{:}bb_0,\ \ldots,\ v_n\text{:}bb_n)\ )$ (phi-assignment)

"x gets v_i if previously executed block was bb_i"

Single Static Assignment: An Example

val:bb

```
int x, y, n;

x = 0;
while (x < N) {
  if (y > 0)
    x = x + y;
  else
    x = x - y;
  y = -1 * y;
}
```

```
0: goto 1
1: x_0 = PHI(0:0, x_3:5);
   y_0 = PHI(y:0, y_1:5);
   if (x_0 < N) goto 2 else goto 6


2: if (y_0 > 0) goto 3 else goto 4


3: x_1 = x_0 + y_0; goto 5


4: x_2 = x_0 - y_0; goto 5


5: x_3 = PHI(x_1:3, x_2:4);
   y_1 = -1 * y_0;
   goto 1
6:
```