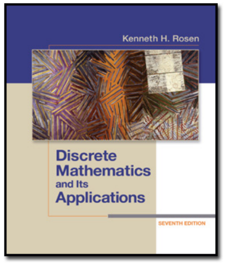
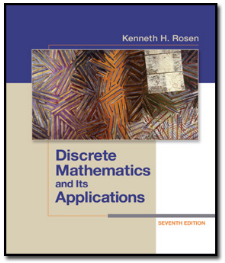


Program Verification (Rosen, Sections 5.5)

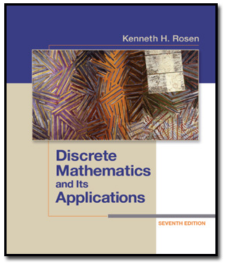
TOPICS

- Program Correctness
- Preconditions & Postconditions
- Program Verification
    - Assignments
    - Composition
    - Conditionals
    - Loops



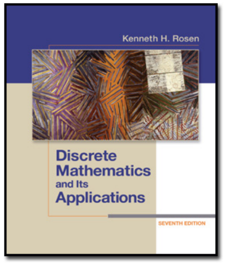
Proofs about Programs

- Why study logic?
- Why do proofs?
- Because we want to prove properties of *programs*
  - In particular, we want to prove properties of variables at specific points in a program



Isn’t testing enough?

- Assuming the program compiles, we perform some amount of testing.
- Testing shows that for specific examples the program seems to be running as intended.
- Testing can only show existence of some bugs but cannot, in general, exhaustively identify all of them.
- Verification can be used to prove the correctness of the program with any input.

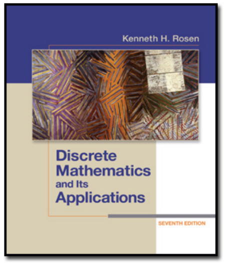


Program Verification

- We consider a program to be *correct* if it produces the expected output **for all possible (combinations of) inputs**.
- Domain of input values can be very large, how many possible values of an integer?

  $-2^{31} – 2^{31}-1$

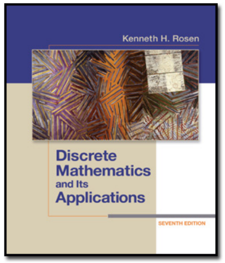
- Domain of doubles even larger!
- Instead we can formally specify program behavior, then use techniques for inferring correctness.



Program Correctness Proofs

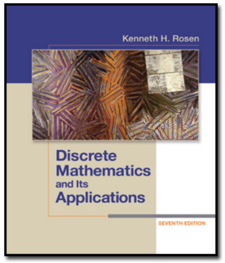
- Two parts:
  - Correct answer when the program terminates (called *partial correctness)*
  - The program does terminate

- We will only do part 1
  - Prove that a method is correct if it terminates
- Part 2 has been shown to be impossible in general! (Halting problem.)



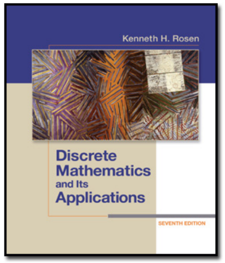
Predicate Logic and Programs

- Variables in programs are like variables in predicate logic:
  - They have a domain of discourse (data type)
  - They have values (drawn from the data type)
- Variables in programs are different from variables in predicate logic:
  - Their values change over time



Assertions

- Two parts:
  - ***Initial Assertion****:* a statement of what must be true about the input values or values of variables at the beginning of the program segment
    - E.g Method that determines the sqrt of a number, requires the input (parameters) to be >= 0
  - ***Final Assertion***: a statement of what must be true about the output values or values of variables at the end of the program segment
    - E.g. What is the final result after a call to the method?

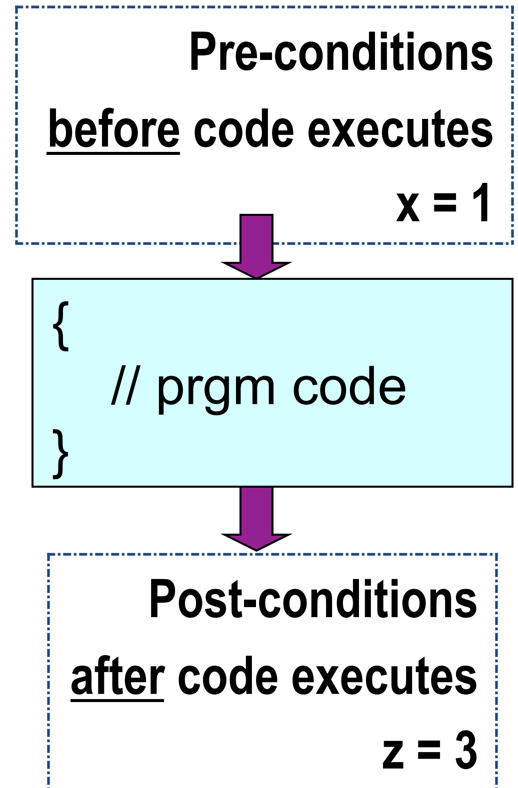


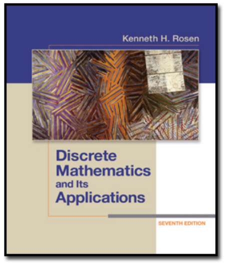
Preconditions and PostConditions

- ***Initial Assertion:*** called the **<u>pre</u>condition**

- ***Final Assertion***: called the **<u>post</u>condition**

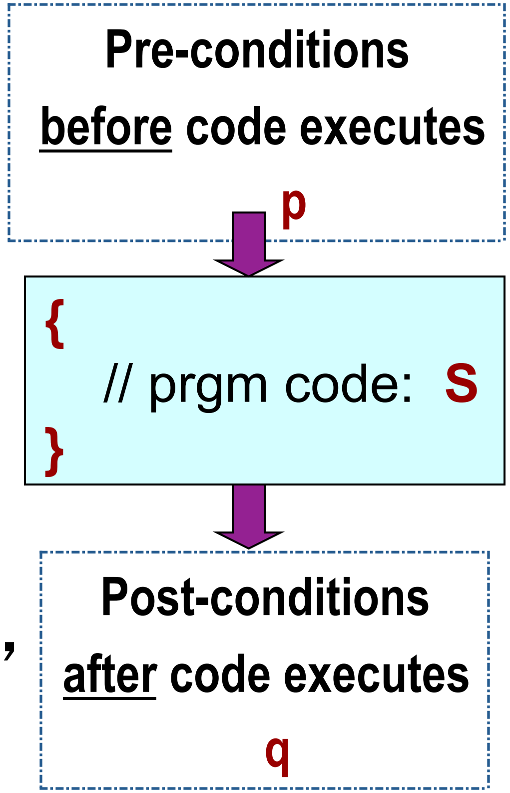
- **Note**: these assertions can be represented as propositions or predicates, OR as asserts in your program!



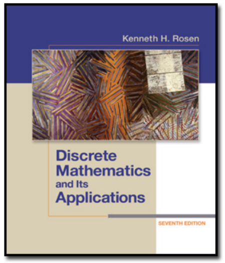


Hoare Triple

- "A program, or program segment, **S**,
  is said to be **partially correct**
  with respect to the
  initial assertion (precondition) **p**
  and the final assertion (postcondition) **q**
  if, whenever **p** is true
  for the input values of **S**
  and S terminates,
  then **q** is true for the output values of **S**."



- Notation: **p{S}q**



Program Verification
Example #1: Assignment Statements

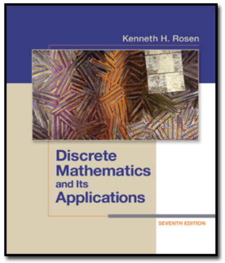
- Assume that our proof system already includes rules of arithmetic…
- Consider the following code:

```
y = 2;
z = x + y;
```

- Precondition: $p(x), x = 1$
- Postcondition: $q(z), z = 3$

What is true BEFORE code executes

What is true AFTER code executes



Program Verification
Example #1: Assignment Statements

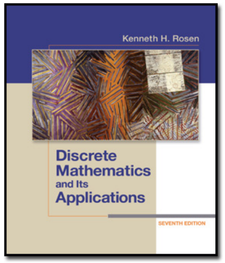
- Prove that the program segment:

  y = 2;

  z = x + y;
- Is correct with respect to

  precondition: x = 1

  postcondition: z = 3
- Suppose x = 1 is true as program begins
  - Then y is assigned the value of 2
  - Then z is assigned the value of 3 (x + y = 1 + 2)
  - Thus, the program segment is correct with regards to the precondition x = 1 and postcondition z = 3



Program Verification Example #2: Assignment Statements

- Prove that the program segment:

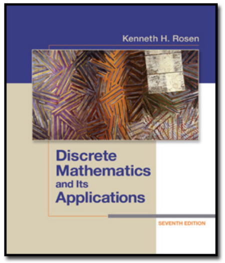
```
x = 2;
z = x * y;
```

- Is correct with respect to

  precondition: y >= 1

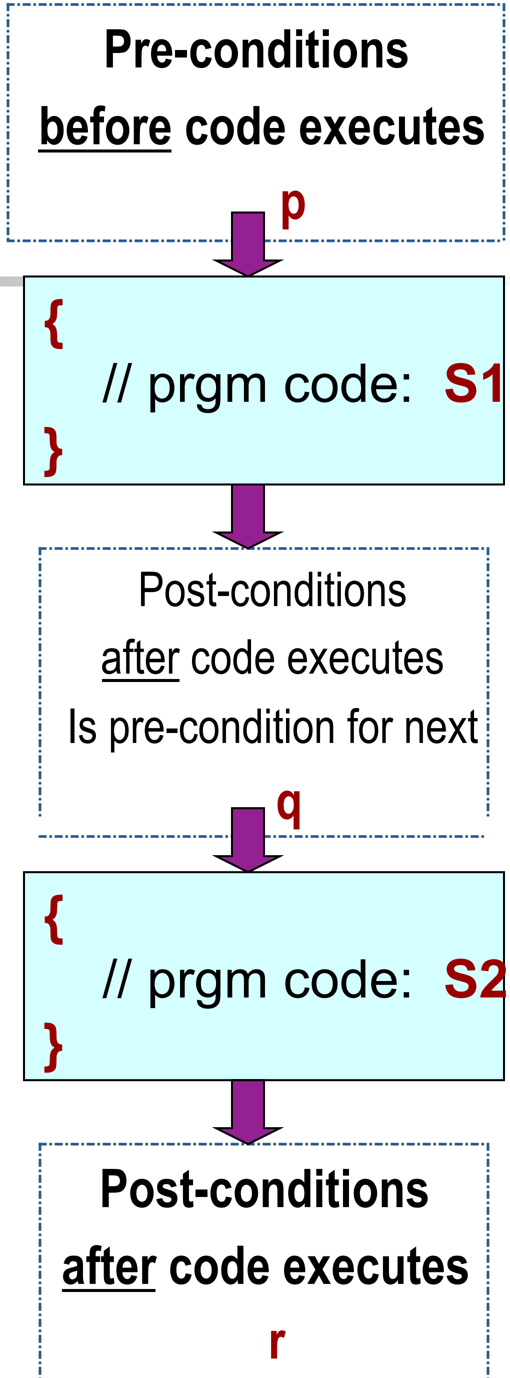
  postcondition: z >= 2

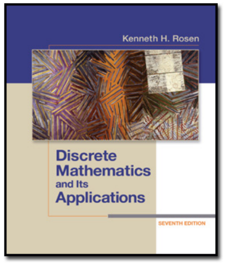
- Suppose y >= 1 is true as program begins
  - Then x is assigned the value of 2
  - Then z is assigned the value of x * y which is 2*(y>=1) which makes z >= 2
  - Thus, the program segment is correct for precondition y >= 1 and postcondition z >= 2



Rule 1: Composition Rule

- Once we prove correctness of program segments, we can combine the proofs together to prove correctness of an entire program.
- This is like the hypothetical syllogism inference rule, or direct proof in Proof Techniques





Program Verification
Example #1: Composition Rule

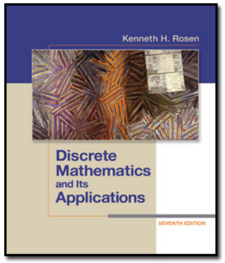
- Prove that the program segment (swap):

```
t = x;
x = y;
y = t;
```

- Is correct with respect to

  precondition: $x = 7, y = 5$

  postcondition: $x = 5, y = 7$



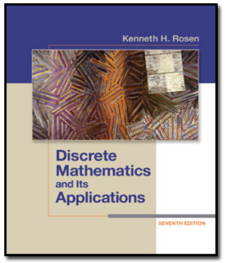
Program Verification
Example #1 (cont.): Composition Rule

Suppose x = 7 and y = 5 is true as program begins

– // Precondition: x = 7, y = 5
  - t = x
– // t = 7, x = 7, y = 5
  - x = y
– // t = 7, x = 5, y = 5
– y = t
– // Postcondition: t = 7, x = 5, y = 7

Thus, the program segment is correct with regards to the precondition that x = 7 & y =5 and

postcondition x = 5 and y = 7



Rule 2: Conditional Statements

- Given

  if (*condition*)

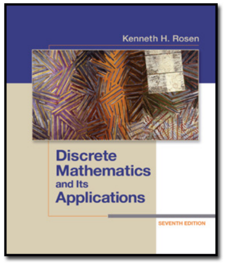
  statement;

  With precondition: ***p*** and postcondition: ***q***

- Must show that
  - Case 1: when ***p (precondition)*** is true and ***condition*** is true then ***q (postcondition)*** is true, when ***S (statement)*** terminates

    OR

  - Case 2: when ***p*** is true and ***condition*** is false, then ***q*** is true (***S*** does not execute)



Conditional Rule: Example #1

Verify that the program segment:

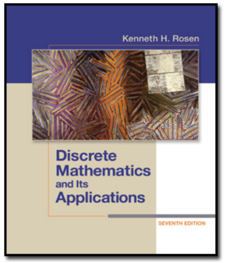
if (x > y) y = x

Is correct with respect to precondition T and postcondition that y >= x   Precondition T (true) is the weakest possible precondition, and nothing can be concluded from it.

Consider the two cases…

1. Condition (x >y) is true, then y = x
2. Condition (x >y) is false, then that means x <= y

Thus, if the precondition is true, then y == x or x <=y which means that the postcondition that y >=x is true



Conditional Rule: Example #2

Verify that the program segment:

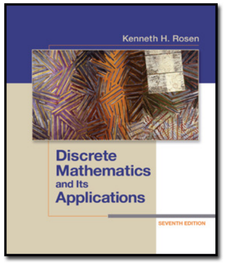
if (x % 2 == 1) x = x + 1

Is correct with respect to precondition T (state of program is correct as enter this program segment) and postcondition that x is even

Consider the two cases…

1. Condition (x % 2 equals 1) is true, then x is odd. If x is odd, then adding 1 means x is even
2. Condition (x % 2 equals 1) is false, then x is even.

Thus, if the precondition is true, then x is odd or x is even which means that the postcondition that x is even is true



Conditional Rule: Example #3 (in code form)

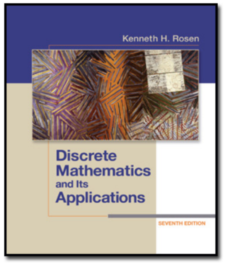
```
int a,b,p;
...
// pre: a>0 AND b>0 AND p + a*b == val
if (a%2==1)  p+=b;
a/=2;
b*=2;
// post: p + a*b == val
```

if a is even, post holds, because at pre: a*b == at post: (a/2) * (b*2)

if a is odd, when we divide odd a by 2 and multiply b by 2,

at pre: a*b == at post: a*b – b, but then b is added to p

try it for p=0,a=3, b=2 at pre, try it for p=0,a=2, b=5 at pre

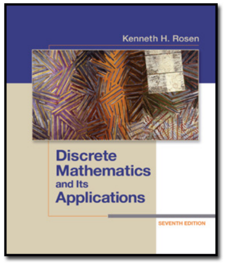


Rule 2a: Conditional with Else

```
if (condition)
   S1;
else
   S2;
```

- Must show that
  - Case 1: when ***p (precondition)*** is true and ***condition*** is true then ***q (postcondition)*** is true, when ***S1 (statement)*** terminates

    OR
  - Case 2: when ***p*** is true and ***condition*** is false, then ***q*** is true, when ***S2*** (***statement***) terminates

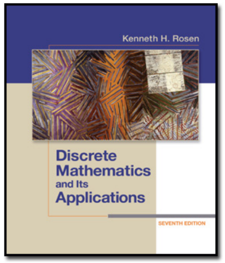


Conditional Rule: Example #4

Verify the program segment:

```
// pre: T
  if (x < 0) abs = -x;
else abs = x;
 // post: abs = |x|
```

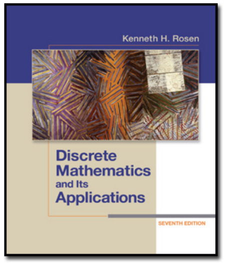
1. Condition (x < 0) is true, then x is negative. Assigning abs the negative of a negative number, means abs is the absolute value of x
2. Condition (x < 0) is false, then x >= 0 which means x is positive. Assigning abs a positive number, means abs is the absolute value of x



How to we prove loops correct?

General idea: a *loop invariant* *allows us to make a static observation (invariant) about a dynamic phenomenon (loop)*

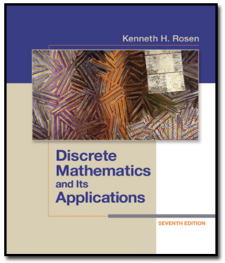
- Find a property that is true before the loop
- Show that it must still be true after every iteration of the loop, so it is true after the loop
- Also, the loop has terminated, so the loop condition is false



Loop invariants

The loop condition and loop invariant allows us to reason about the behavior of the loop:

```
<loop invariant>
while(condition){
   <condition AND loop invariant>
    S;
   <loop invariant>
}
< not condition AND loop invariant>
```



In other words...

```
<loop invariant>
while(test){
  <test AND loop
   invariant>
     S;
   <loop invariant>
}
< not test AND
  loop invariant>
```

If we can prove that

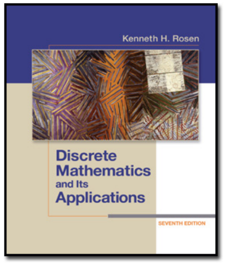
- the loop invariant holds before the loop

and

- the loop body keeps the loop invariant true
  i.e. <test AND loop invariant> S; <loop invariant>

then we can infer that

- not test AND loop invariant
  holds after the loop terminates



Example #1: loop index value after loop

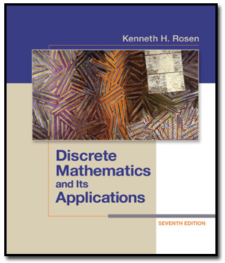
```
<precondition: n>0>
int i = 0;
 while (i < n){
   i = i+1;
}
<post condition: i==n >
```

We want to prove:
i==n right after the loop

What is a good loop invariant?

i==0?
i==n?

i<=n

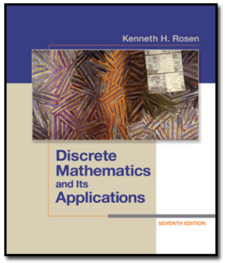


loop index value after loop

```
// precondition: n>0
int i = 0;
// i<=n  loop invariant  WHY TRUE?
 while (i < n){
   // i < n  test passed
   //   AND
   //   i<=n  loop invariant
     i++;
   // i <= n  loop invariant
}
// i>=n  AND i <= n  → i==n
```

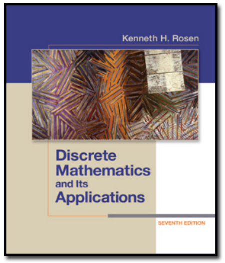
So we can conclude:

i==n right after the loop



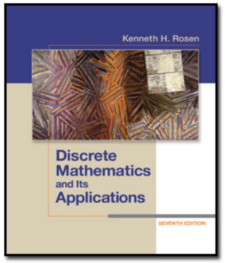
Example #2: summing

```
int total (int[] elements){
  int sum = 0,i = 0, n = elements.length;
  // invariant?
   while (i < n){
     // i<n and invariant?
      sum  += elements [i];
      i++;
     // invariant?
  }
  // i==n  (previous example)  AND invariant
  // → sum == sum of int[] elements
 return sum;
}
```



Summing

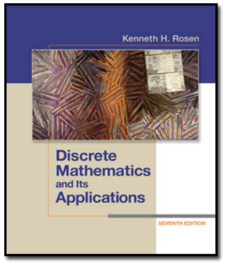
```
int total (int[] elements){
  // pre elements.length > 0
  int sum = 0,i = 0, n = elements.length;
  // sum == sum of elements from 0 to i-1
   while (i < n){
     // sum == sum of elements 0..i-1
      sum  += elements [i];
      i++;
     // sum == sum of elements 0..i-1
  }
  // i==n  (previous example)  AND
  // sum has sum elements 0..i-1  →  sum == sum of elements 0..n-1
  //                              →  sum == sum of int[] elements
```



Example #3: factorial

Given following program segment, what is the loop invariant for factorial?

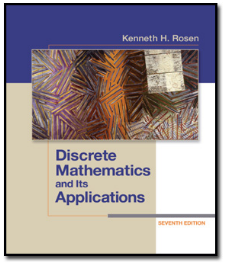
```
// precondition: n >= 1
  i = 1;
  factorial = 1;

  while (i < n) {

            i++;

            factorial *= i;

  }
```



Example #3: factorial

Given following program segment, what is the loop invariant for factorial?

```
// precondition: n >= 1
  i = 1;
  factorial = 1;
  // i<= n AND factorial = i!
  while (i < n) {
          // i<n AND i<= n AND factorial == i!
          i++;
          factorial *= i;
          //  i<= n AND factorial == i!
  }
// i==n (example 1) and factorial == i!  → factorial == n!
```



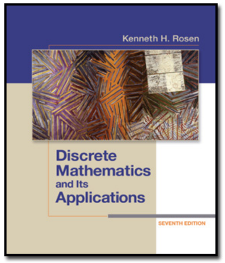
Example #4: Egyptian multiplication

	A	B	
	19	5	
/2	9	10	*2
/2	4	20	*2
/2	2	40	*2
/2	1	80	*2

throw away all rows with even A:

	A	B
	19	5
	9	10
	1	80
	___________	
add B's		95

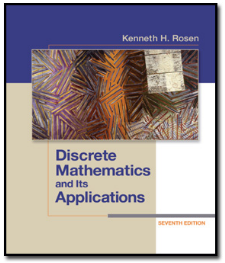
--> the product !!



Try it on 7 * 8

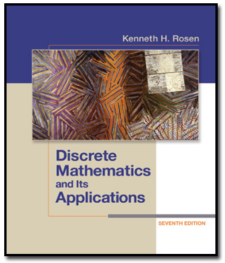
left	right	p	a	b
7	8	0	7	8
		+=b: 8	3	16
		+=b: 24	1	32
		+=b: **56**	0	64

Now try it on 8*7



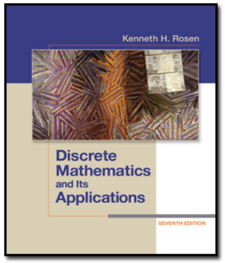
The code:

```
// pre:   left >=0 AND right >=0
int a=left, b=right, p=0; // and and b copies, p accumulating product

while (a!=0){

  if (odd(a))  p+=b;
  a/=2;
  b*=2;

}
// post: p == left*right
```



Can we show it works?
Yes, loop invariants!!

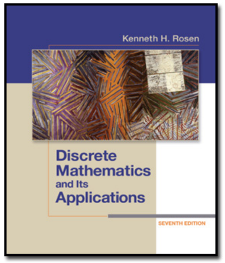
```
// pre:   left >=0 AND right >=0
int a=left, b=right, p=0; // and and b copies, p accumulating product
// p+(a*b) == left * right        loop invariant
while (a!=0){
  // a!=0 and p+a*b == left*right   loop condition and loop invariant
  if (odd(a))  p+=b;
  a/=2;
  b*=2;
  // p+(a*b) == left*right  (see slide 19 conditional rule, example #3)
}
// a==0 and p+a*b == left*right  →  p == left*right
```



int representation 19*5

```
  00101
  10011
______
    101    5
   1010   10
  00000
 000000
1010000   80
_______
1011111   95 = 64 + 31
```

Try it on 7*8 and 8*7



Summary: Loop Invariant Reasoning

```
// precondition
// use precondition to show loop invariant true
while (b){
  // b  AND loop invariant
      S;
  // use S to show loop invariant true
}
// not b  AND loop invariant  → conclusion
```

not b AND loop invariant: stronger than loop invariant alone.