An Outline of Discrete Mathematics for Computer Science

Mohamed G. Gouda
(gouda@cs.utexas.edu)

Department of Computer Science
The University of Texas at Austin

August 2015

Table of Contents

1 Formulas

A Boolean value is either “false”, denoted F, or “true”, denoted T

A Boolean parameter x has a name, x, and a value, which is a Boolean value. Thus, the value of a Boolean parameter x is either F or T

A Boolean operator applies to one or two Boolean values and returns a Boolean value. There are three basic Boolean operators denoted:

not
and
or

Operator “not” applies to one Boolean value and returns a Boolean value. This operator is defined as follows:

not x	=	F	if x is T
		T	otherwise

This definition is called the truth table of operator “not”

Operator “and” applies to two Boolean values and returns a Boolean value. This operator is defined as follows:

x and y	=	F	if x is F or y is F
		T	otherwise

This definition is called the truth table of operator “and”. Note that the and in the left-hand side of this definition is a Boolean operator and the or in the right – hand side of this definition is an English or.

Operator “or” applies to two Boolean values and returns a Boolean value. This operator is defined as follows:

x or y	=	F	if x is F and y is F
		T	otherwise

This definition is called the truth table of operator “or”. Note that the or in the left-hand side of this definition is a Boolean operator and the and in the right – hand side of this definition is an English and.

Boolean Formulas

A Boolean formula f over Boolean parameters x, y, and z, denoted f(x, y, z), is an expression that involves:

Boolean values: F, T
Boolean parameters: x, y, z
Boolean operators: not, and, or
Parentheses: (…)

The parentheses determine the order in which the operators are applied when we compute the Boolean value of the formula

Example:
Let f(x, y, z) be the formula
(not x) or (not (y and (not z)) or F)

The Boolean value of this formula can be computed as follows

f1	= (not x)
f2	= (not z)
f3	= (y and f2)
f4	= (not f3)
f5	= (f4 or F)
f(x, y, z)	= (f1 or f5)

Binding Rules

The parentheses (…) have the strongest binding power

Example:
To compute (x and y) or (not z), first apply the “and” and “not” operators, then apply the “or” operator.

The “not” operators have the second strongest binding power

Example:
To compute (not x or y) and (not y or z), first apply the “not” operators, then apply the “or” operators, then apply the “and” operator

The “and” operators have the third strongest binding power

Example:
To compute (x and y or z and x or y), first apply the “and” operators in any order, then apply the “or” operators in any order.

The “or” operators have the fourth strongest binding power

Example:
To compute (x or y or not z or x), first apply the “not” operator, then apply the “or” operators in any order.

Equivalent Formulas

Two formulas f(x) and g(x) are said to be equivalent, denoted f(x) = g(x), iff
the two Boolean values of f(F) and g(F) are the same and
the two Boolean values of f(T) and g(T) are the same

Equivalence Laws

Elementary Laws:

f(x) and F	= F
f(x) and T	= f(x)
f(x) or F	= f(x)
f(x) or T	= T

Idempotence Laws:

not (not (f(x)))	= f(x)
f(x) and f(x)	= f(x)
f(x) and not (f(x))	= F
f(x) or f(x)	= f(x)
f(x) or not (f(x))	= T

Symmetry Laws:

f(x) and g(x)	= g(x) and f(x)
f(x) or g(x)	= g(x) or f(x)

Absorption Laws:

f(x) and (f(x) or g(x))	= f(x)
f(x) or (f(x) and g(x))	= f(x)

De Morgan's Laws:

not (f(x) and g(x))	= not (f(x)) or not (g(x))
not (f(x) or g(x))	= not (f(x)) and not (g(x))

Distribution Laws:

f(x) and (g(x) or h(x))	= (f(x) and g(x)) or (f(x) and h(x))
f(x) or (g(x) and h(x))	= (f(x) or g(x)) and (f(x) or h(x))

Associativity Laws:

f(x) and (g(x) and h(x))	= (f(x) and g(x)) and h(x)
f(x) or (g(x) or h(x))	= (f(x) or g(x)) or h(x)

Proving Equivalence Laws

Equivalence laws can be proven using truth tables.

Example:
To prove the elementary law

f(x) and F = F

note that either f(x) = F or f(x) = T.

If f(x) = F, then the law becomes F and F = F

If f(x) = T, then the law becomes T and F = F

In both cases, the law follows from the truth table of the “and” operator

Example:
To prove the absorption law

f(x) and (f(x) or g(x)) = f(x)

note that either f(x) = F or f(x) = T and either g(x) = F or g(x) = T.

If f(x) = F and g(x) = F, then the law becomes F and (F or F) = F

If f(x) = F and g(x) = T, then the law becomes F and (F or T) = F

If f(x) = T and g(x) = F, then the law becomes T and (T or F) = T

If f(x) = T and g(x) = T, then the law becomes T and (T or T) = T

In all cases, the law follows from the truth tables of the “and” and “or” operators

Proving Formula Equivalence

There are two methods to prove that two formulas are equivalent:

Method 1: Using Truth Tables
Method 2: Using Equivalence Laws

We demonstrate these two methods by using them to prove that the following two formulas f(x, y) and g(x, y) are equivalent:

f(x, y) = not ((x and not y) or (x and y))
g(x, y) = not x

Method 1: Using Truth Tables:
There are four possible values for the pair (x, y): (F, F), (F, T), (T, F), and (T, T).

If (x, y) = (F, F), then f(x, y) = not ((F and not F) or (F and F)) = T = not F = g(x, y)

If (x, y) = (F, T), then f(x, y) = not ((F and not T) or (F and T)) = T = not F = g(x, y)

If (x, y) = (T, F), then f(x, y) = not ((T and not F) or (F and F)) = F = not T = g(x, y)

If (x, y) = (T, T), then f(x, y) = not ((T and not T) or (T and T)) = F = not T = g(x, y)

Method 2: Using Equivalence Laws:

```
f(x, y)   =     {definition of f(x, y)}
                not ((x and not y) or (x and y))

          =     {distribution law}
                not (x and (not y or y))

          =     {idempotence law}
                not (x and T)

          =     {elementary law}
                not x

          =     {definition of g(x, y)}
                g(x, y)
```

Satisfiability

A formula f(x) is satisfiable iff there is a Boolean value u such that f(u) = T

A formula f(x, y) is satisifiable iff here are two Boolean values u and v such that f(u, v) = T

And so on …

Example:
The formula f(x, y, z) = (x or y or not z) is satisfiable because f(T, y, z) = T

It is also satisfiable because f(x, T, z) = T

It is also satisfiable because f(x, y, F) = T

Example:
The formula f(x, y) = (x and y) and (not x or not y) is not satisfiable because it is equivalent to the formula F as indicated by the following proof:

```
        f(x, y)
=       {definition of f(x, y)}
        (x and y) and (not x or not y)

=       {de-Morgan’s}
        (x and y) and not (x and y)

=       {idempotence}
        F
```

Normal Forms

A formula f is called a factor iff f is written as one of the following four forms: F, T, x, or not x

A formula f is said to be in a Conjunctive Normal Form (or CNF for short) iff f is written as a conjunction of one or more disjunctions of one or more factors. In other words, f is written as

f = ((factor or factor or …) and (factor or factor or …) and …)

Examples:
The formula (T and (not x or y) and (x or y)) is in CNF

The formula (not x and y) is in CNF

The formula (not (x and y)) is not in CNF

A formula f is said to be in a Disjunctive Normal Form (or DNF for short) iff f is written as a disjunction of one or more conjunctions of one or more factors. In other words, f is written as

f = ((factor and factor and …) or (factor and factor and …) or …)

Examples:
The formula (F or (not x and y) or (x and y)) is in DNF

The formula (not x and y) is in DNF

The formula (not (x and y)) is not in DNF

Theorem:
For each formula f(x) there is a formula g(x) in CNF and a formula h(x) in DNF where f(x) = g(x) and f(x) = h(x)

Additional Boolean Operators

Exclusive-Or:

(f(x) xor g(x))   =   (not f(x) and g(x) or f(x) and not g(x))

The operator xor reads “exclusive or”

The “exclusive-or” operators have the fifth strongest binding powers

Implication:

(f(x) -> g(x))   =   (not f(x) or g(x))

The operator -> reads “implies”

The “implies” operators have the sixth strongest binding powers

Equivalence:

(f(x) = g(x))   =   (f(x) and g(x) or not f(x) and not g(x))

The operator = reads “equals”

The ”equals” operators have the seventh strongest binding powers

2 Predicates

A predicate P is a statement that can be either false or true

Example:
Examples of predicates are

(2 is even)
(5 is even)
(2 >= 5)

If P is a false predicate, then we say that the Boolean value of P is F.
If P is a true predicate, then we say that the Boolean value of P is T

Example:

The Boolean value of (2 is even) is T
The Boolean value of (5 is even) is F
The Boolean value of (2 >= 5) is F

Parameterized Predicates

A predicate P can involve a parameter, say x, whose value is taken from some domain, say Dx. This parameterized predicate is denoted P(x)

For each value u in Dx, the Boolean value of predicate P(u) is either F or T

Example:
Let Dx be the set of all non-negative integers {0, 1, …}. Also let P(x) be the parameterized predicate (x is even).

The Boolean value of P(0) is T
The Boolean value of P(1) is F
The Boolean value of P(2) is T
…

A parameterized predicate can involve two parameters …

More on Parameterized Predicates

A predicate P can involve two parameters x and y whose values are taken from some domains Dx and Dy, respectively. This parameterized predicate is denoted P(x, y)

For each value u of x in Dx and each value v of y in Dy, the Boolean value of predicate P(u, v) is either F or T

Example:
Let Dx and Dy be the set of all non-negative integers {0, 1, …}. Also let P(x, y) be the parameterized predicate (x >= y).

> The Boolean value of P(0, 0) is T
> The Boolean value of P(0, 1) is F
> The Boolean value of P(1, 0) is T
> The Boolean value of P(1, 1) is T
> …

A predicate P can involve more than two parameters …

Quantifiers and Quantified Predicates

There are two quantifiers, “All” and “Exist”, that can be used to transform each parameterized predicate P(x) into two quantified predicates

(All u in Dx, P(u))

(Exist u in Dx, P(u))

The Boolean value of the predicate (All u in Dx, P(u)) is T iff for each value u in Dx, the Boolean value of P(u) is T

Note that if the domain Dx is empty, then the Boolean value of the predicate (All u in Dx, P(u)) is T

The Boolean value of predicate (Exist u in Dx, P(u)) is T iff for some value u in Dx, the Boolean value of P(u) is T

If the domain Dx is well-understood, then

(All u in Dx, P(u)) can be written as (All u, P(u))

(Exist u in Dx, P(u)) can be written as (Exist u, P(u))

To reduce the number of identifiers used, a value u in domain Dx can be renamed x. Thus

(All u, P(u)) can be written as (All x, P(x))

(Exist u, P(u)) can be written as (Exist x, P(x))

Example of Quantified Predicates

Let Dx be the set of all non-negative integers and let P(x) and Q(x) be the two parameterized predicates:

P(x)	= (x is even)
Q(x)	= (x is odd)

Then

P(2)	= T
Q(2)	= F
not (Q(2))	= T
P(2) and Q(2)	= F
P(2) or Q(2)	= T
P(2) -> Q(2)	= F
P(2) xor Q(2)	= T
(P(2) = Q(2))	= F
(All x, P(x))	= F
(Exist x, P(x))	= T
(All x, Q(x))	= F
(Exist x, P(x))	= T
(All x, P(x) and Q(x))	= F
(Exist x, P(x) and Q(x))	= F
(All x, P(x) or Q(x))	= T
(Exist x, P(x) or Q(x))	= T

Nested Quantifiers

A parameterized predicate P(x, y) can be transformed into a (non-parameterized) predicate by

(1) quantifying parameter x using “All” or “Exist” and
(2) quantifying parameter y using “All” or “Exist”

Thus, P(x, y) can be transformed into one of the following four predicates:

(All x, All y, P(x, y))
(All x, Exist y, P(x, y))
(Exist x, All y, P(x, y))
(Exist x, Exist y, P(x, y))

Equivalence Laws of Quantified Predicates

Each equivalence law of quantified predicates is in one of the following two forms:

<left quantified predicate> = <right quantified predicate>

<left quantified predicate> =? <right quantified predicate>

The first form indicates that the two quantified predicates are guaranteed to always have the same Boolean value. The second form indicates that the two quantified predicates sometimes have the same Boolean value and sometimes have distinct Boolean values

Symmetry of “All” and “Exist”:

(All x, All y, P(x, y)) = (All y, All x, P(x, y))
(Exist x, Exist y, P(x, y)) = (Exist y, Exist x, P(x, y))
(All x, Exist y, P(x, y)) =? (Exist y, All x, P(x, y))

De Morgan’s:

not (All x, P(x)) = (Exist x, not (P(x)))
not (All x, All y, P(x, y)) = (Exist x, Exist y, not (P(x, y)))

not (Exist x, P(x)) = (All x, not (P(x)))
not (Exist x, Exist y, P(x, y)) = (All x, All y, not (P(x, y)))

not (All x, Exist y, P(x, y)) = (Exist x, All y, not (P(x, y)))

Distribution of “All” over “and” and “or”:

(All x, P(x) and Q(x)) = ((All x, P(x)) and (All x, Q(x)))
(All x, P(x) or Q(x)) =? ((All x, P(x)) or (All x, Q(x)))

Distribution of “Exist” over “and” and “or”:

(Exist x, P(x) and Q(x)) =? ((Exist x, P(x)) and (Exist x, Q(x)))
(Exist x, P(x) or Q(x)) = ((Exist x, P(x)) or (Exist x, Q(x)))

Notation

The predicate (All x, P(x)) can be written as P(x)

The predicate (All x, All y, P(x, y)) can be written as (All x, y, P(x, y))

The predicate (Exist x, Exist y, P(x, y)) can be written as (Exist x, y, P(x, y))

3 Proof Styles

There are eight styles to prove that the Boolean value of a predicate is T:

1. Guessing
2. Case Analysis
3. Direct Inference
4. Indirect Inference
5. By-contradiction
6. Two-Sided Inference
7. Using Proven Predicates
8. Induction

Guessing

This proof style can be used to prove a predicate of the form (Exist x in Dx, P(x))

This proof consists of two steps:

Step 1:
Choose any value v from the domain Dx

Step 2:
If P(v) is T, then the proof is complete, else go back to Step 1

Example:
Show that the formula f(x, y, z) = (x or (not(y) and z)) is satisfiable, i.e. prove the predicate (Exist x, y, z, f(x, y, z)). The proof proceeds as follows:

Step 1: Choose (x, y, z) to be (F, T, F)
Step 2: f(F, T, F) = (F or (F and F)) = F, go back to Step 1
Step 1: Choose (x, y, z) to be (T, T, T)
Step 2: f(T, T, T) = (T or (F and T)) = T, proof is complete

Case Analysis

This proof style can be used to prove a predicate of the form (All x in Dx, P(x))

This proof consists of two steps:

Step 1:
Partition the domain Dx into non-overlapping subsets d1, …, dk

Step 2:
For each subset di, show that the predicate (All x in di, P(x)) is T

Example:
Let Dx be the set {1, 2, …} of all positive integers. Also let P(x) be the parameterized predicate P(x) = (x^2 + x^3 != 100). Prove the predicate (All x in Dx, P(x)).

Before we develop the proof, we make three observations. First, the value of (x^2 + x^3) increases with each increase in the value of x. Second, for every value of x that is at most 4, the value of (x^2 + x^3) is less than 100. Third, for every value of x that is at least 5, the value of (x^2 + x^3) is more than 100.

The case analysis proof proceeds as follows:

Step 1:
Partition Dx into two subsets d1 and d2, where subset d1 is the set of all positive integers that are at most 4, and subset d2 is the set of all positive integers that are at least 5.

Step 2:
For every x in d1, P(x) =< (4^2 + 4^3) = 80 != 100
For every x in d2, P(x) >= (5^2 + 5^3) = 150 != 100

Direct Inference

This proof style can be used to prove (or infer) a predicate Q from a predicate P, denoted P => Q.

(Note that this proof style can also be used to prove (or infer) a predicate Q from the true predicate T, denoted T => Q.)

A direct inference proof of P => Q consists of (k+1) inference steps as follows:

```
        P
=>      {hint on why (P -> P1) is T}
        P1
…
        Pk
=>      {hint on why (Pk -> Q) is T}
        Q
```

More on Direct Inference

This proof style can be also used to prove (All x, P(x) => Q(x))

A direct inference proof of (All x, P(x) => Q(x)) consists of (k+1) inference steps as follows:

```
            P(x)
    =>      {hint on why (All x, P(x) -> P1(x)) is T}
            P1(x)
    …
            Pk(x)
    =>      {hint on why (All x, Pk(x) -> Q(x)) is T}
            Q(x)
```

Example:
Let Dx and Dy be sets of all non-negative integers. A direct inference proof of (All x, y, (x is square) and (y is square) => (x*y is a square)) is as follows:

```
            (x is a square) and (y is a square)
    =>      {definition of square}
            (x = m^2) and (y = n^2) for some integers m and n

    =>      {multiply x*y}
            (x*y = (m*n)^2) for some integers m and n

    =>      {m*n is an integer r}
            (x*y = r^2) for some integer r

    =>      {definition of square}
            (x*y is a square)
```

Example:
Let Dx be set of all nonnegative integers. A direct inference proof of (All x, (x is odd) => (x^2 is odd)) is as follows:

```
            (x is odd)
    =>      {definition of odd}
            (x = (2m + 1)) for some integer m

    =>      {compute x^2}
            (x^2 = (4*m^2 + 4m +1)) for some integer m

    =>      {arithmetics}
            (x^2 = 2n + 1) for some integer n

    =>      {definition of odd}
            (x^2 is odd)
```

Indirect Inference

This proof style can be used to prove any of the following:

```
P => Q
Q
(All x, P(x) => Q(x))
```

An indirect inference proof of (P => Q)
is a direct inference proof of (not Q => not P)

An indirect inference proof of (Q)
is a direct inference proof of (not Q => F)

An indirect inference proof of (All x, P(x) => Q(x))
is a direct inference proof of (All x, not Q(x) => not P(x))

Example:
Let Dx be set of all nonnegative integers. A direct inference proof of (All x, (x^2 is odd) => (x is odd)) will not go through. But an indirect inference proof goes through as follows:

```
        not (x is odd)
=>      {x is either odd or even}
        (x is even)

=>      {definition of even}
        (x = 2m) for some integer m

=>      {compute x^2}
        (x^2 = 2*2*m^2) for some integer m

=>      {definition of even}
        (x^2 is even)

=>      {x^2 is either odd or even}
        not (x^2 is odd)
```

Note that this indirect inference proof contains a direct inference proof of (All x, (x is even) => (x^2 is even))

By-Contradiction

This proof style can be used to prove any of the following:

```
P => Q
Q
(All x, P(x) => Q(x))
```

A by-contradiction proof of (P => Q)
is a direct inference proof of ((P and not Q) => F)

A by-contradiction proof of (Q)
is a direct inference proof of (not Q => F)

A by-contradiction proof of (All x, P(x) => Q(x))
is a direct inference proof of ((Exist x, P(x) and not Q(x)) => F)

Example:
A direct inference proof of the predicate (sqrt(2) is irrational) will not go through since there is no simple definition of “irrational”. But a by-contradiction proof of this predicate goes through as follows:

```
        not (sqrt(2) is irrational)
=>      {sqrt(2) is either rational or irrational}
        (sqrt(2) is rational)

=>      {Exist x, y, sqrt(2) = x/y, y != 0, x and y share no factors,
        x^2 = 2*y^2, x^2 is even, x is even, x = 2n for some integer n,
        x^2 = 4*n^2 from some integer n,
        y^2 = 2*n^2 for some integer n, y^2 is even, y is even,
        both x and y are even, x and y share factor 2}

        (Exist x, y, x and y share no factors and x and y share factor 2)

=>      {(Exist x, y, P(x, y) and not P(x, y)) = F}

        F
```

Two-Sided Inference

This proof style can be used to prove any of the following:

```
P <=> Q
Q
(All x, P(x) <=> Q(x))
```

A two-sided inference proof of (P <=> Q)
can consist of two direct inference proofs of (P => Q) and (Q => P)

Another two-sided inference proof of (P <=> Q)
can consist of (k+1) two-sided inference steps as follows:

```
        P
<=>     {hint on why (P = P1) is T}
        P1
…
        Pk
<=>     {hint on why (Pk = Q) is T}
        Q
```

A two-sided inference proof of (Q)
is a two-sided inference proof of (Q <=> T), where T is the true predicate

Example:
Let Dx and Dy be sets of all non-negative real numbers. A two-sided inference proof of the predicate (All x, y, ((x+y)/2 >= sqrt(x*y)) <=> T) proceeds as follows:

```
        (x+y)/2  >=  sqrt(x*y)
<=>     {squaring and square rooting maintain >=}
        (x+y)^2  >=  4*x*y

<=>     {arithmetics}
        (x^2 – 2*x*y + y^2)  >=  0

<=>     {arithmetics}
        (x-y)^2  >=  0

<=>     {squaring of a real number is at least zero}
        T
```

Using Proven Predicates

Once we prove a predicate, we can later use this predicate in any inference step

Example:
Let Dx be a set of all non-negative integers. A direct inference proof of (All x, (x^2 is even) => (x is even)) will not go through. But an indirect inference proof goes through as follows:

not (x is even)
=> {x is either odd or even}
(x is odd)

=> {we have proven earlier (All x, (x is odd) => (x^2 is odd))}
(x^2 is odd)

=> {x^2 is either odd or even}
not (x^2 is even)

Induction

This proof style can be used to prove a predicate of the form (All x in Dx, x >= b, P(x)), where Dx is the set of all integers.

There are two “flavors” of induction proofs: regular induction proofs and strong induction proofs.

A regular induction proof of the predicate (All x, x >= b, P(x)) consists of two parts:

1. Base case: (x = b):
   A two-sided inference proof of (P(b) <=> T)

2. Induction step:
   A direct inference proof of the predicate
   (All x, x >= b, P(x) => P(x+1))

A strong induction proof of the predicate (All x, x >= b, P(x)) consists of two parts:

1. Base case: (x = b):
   A two-sided inference proof of (P(b) <=> T)

2. Induction step:
   A direct inference proof of the predicate
   (All x, x >= b, (P(b) and P(b+1) and … and P(x)) => P(x+1))

Example 1 of a Regular Induction Proof

A regular induction proof of predicate (All x, x >= 1, 1+2+…+x = x*(x+1)/2) proceeds as follows:

Let P(x) be the predicate (1 + 2 +…+ x = x*(x+1)/2)

Base case: x = 1:
A two-sided inference proof of (P(1) <=> T) proceeds as follows:
P(1) <=> {1 = 1} T

Induction step:
A direct inference proof of (All x, x >= 1, P(x) => P(x+1)) proceeds as follows:

```
        P(x)
=>      {definition of P(x)}
        1+2+…+x = x*(x+1)/2

=>      {add x+1 to both sides}
        1+2+…+(x+1) = (x*(x+1)/2) + (x+1)

=>      {arithmetics}
        1+2+…+(x+1) = (x+1)*(x+2)/2

=>      {definition of P(x+1)}
        P(x+1)
```

Example 2 of a Regular Induction Proof

A regular induction proof of predicate (All x, x >= 0, (2^0 + 2^1 +…+ 2^x) = (2^(x+1) – 1)) proceeds as follows:

Let P(x) be the predicate (2^0 + 2^1 +…+ 2^x = (2^(x+1) – 1))

Base case: x = 0:
A two-sided inference proof of (P(0) <=> T) proceeds as follows:
P(0) <=> {1 = 1} <=> T

Induction step:
A direct inference proof of (All x, x >= 0, P(x) => P(x+1)) proceeds as follows:

```
        P(x)
=>      {definition of P(x)}
        2^0  + 2^1 +…+ 2^x  =  (2^(x+1) – 1)

=>      {add 2^(x+1) to both sides}
        2^0  + 2^1 +…+ 2^(x+1)  =  (2^(x+1) – 1) + 2^(x+1)

=>      {arithmetics}
        2^0  + 2^1 +…+ 2^(x+1)  =  (2^(x+2) – 1)

=>      {definition of P(x+1)}
        P(x+1)
```

4 Program Specification and Verification

In this section we discuss:

1. How to define programs
2. How to specify what a defined program is supposed to do
3. How to verify that a defined program does what it is supposed to do

For simplicity, we limit our discussion to programs that has only two variables, x and y, of type non-negative integers.

In this case,

1. The program input is the initial values of x and y.
2. Each program step updates the value of x or the value of y.
3. The program output is the final values of x and y.

Program Definition

A <program> is defined recursively as follows:

```
1. skip
2. x := E(x, y)
3. y := E(x, y)
4. <program>; <program>
5. if   B(x,y)
   then <program>
   else <program> fi
6. while   B(x, y)
   do <program> od
```

Note that E(x, y) is an expression that involves x and y, and B(x, y) is a predicate that involves variables x and y.

Program 1:

```
x := 0;
while  y != 0
do x:= x+5; y:= y-1 od
```

This program assigns to x the value (5*Y), where Y is the initial value of y, and assigns to y the value 0.

Program 2:

```
if x >= y
then x := y
else skip fi
x := x+1
```

This program assigns to x the value (min(X, Y) + 1), where X and Y are the initial values of x and y, and leaves the value of y unchanged.

Program Specification

A specification of a program s involves two predicates P(x, y) and Q(x, y) and is written as follows:

(P(x, y)) s (Q(x, y))

This specification states that if program s starts its execution at a state where the values x and y make P(x, y) true, and if execution of program s terminates, then program s terminates its execution at a state where the values of x and y make Q(x, y) true.

Theorem:
Let s be a program and let P(x, y), Q(x, y), P'(x, y), and Q'(x, y) be four parametrized predicates:

If (P(x, y)) s (Q(x, y)) and
if (All x, y, P'(x, y) -> P(x, y) and
if (All x, y, Q(x, y) -> Q'(x, y)
then (P'(x, y)) s (Q'(x, y))

How to verify that a specification (P(x, y)) s (Q(x, y)) holds for program s?

Program Verification

To verify that a specification
    (P(x, y)) skip (Q(x, y))
holds for a program skip, prove the following predicate
    (All x, y, (P(x, y)) => (Q(x, y)))

To verify that a specification
    (P(x, y)) x := E(x, y) (Q(x, y))
holds for a program x := E(x, y), prove the following predicate
    (All x, y, (P(x, y)) => (Q(E(x, y), y)))

To verify that a specification
    (P(x, y)) y := E(x, y) (Q(x, y))
holds for a program y := E(x, y), prove the following predicate
    (All x, y, (P(x, y)) => (Q(x, E(x, y)))

Example 1:
To verify that a specification
    (x >= 0) x := x+1 (x > 0)
holds for a program x := x+1, prove the following predicate
    (All x, y, (x >= 0) => (x+1 > 0)))

Example 2:
To verify that a specification
    (y = 10) x := y (y > 5))
holds for a program x := y, prove the following predicate
    (All x, y, (y = 10) => (y > 5))

Example 3:
To verify that a specification
    (y = 10) x := y (x > 5)
holds for a program x := y, prove the following predicate
    (All x, y, (y = 10) => (y > 5))

More on Program Verification

To verify that a specification

    (P(x, y)) (s1; s2) (Q(x, y))

holds for a program (s1; s2), exhibit a predicate R(x, y) and verify that the following two program specifications hold

    (P(x, y)) s1 (R(x, y))
    (R(x, y)) s2 (Q(x, y))

To verify that a specification

    (P(x, y)) (if B(x, y) then s1 else s2 fi) (Q(x, y))

holds for a program (if B(x, y) then s1 else s2 fi), verify that the following two program specifications hold

    (P(x, y) and B(x, y)) s1 (Q(x, y))
    (P(x, y) and not B(x, y)) s2 (Q(x, y))

To verify that a specification

    (P(x, y)) (while B(x, y) do s od) (Q(x, y))

holds for a program (while B(x, y) do s od), exhibit a predicate I(x, y) then prove the following two predicates and verify that the following program specification holds

    (All x, y, P(x, y) => I(x, y))
    (All x, y, (I(x, y) and not B(x, y)) => (Q(x, y))

    (I(x, y) and B(x, y)) s (I(x, y))

Predicate I(x, y) is called an invariant of the loop (while B(x, y) do s od)

5 Graphs

A graph G is defined as a pair (V, E), where V is a nonempty set of vertices and E is a set of edges. Each two distinct vertices in V can be connected by at most one edge in E. An edge that connects two (distinct) vertices u and v is denoted (u, v) or (v, u)

Two distinct vertices u and v are said to be neighbors iff they are connected by an edge (u, v). The neighborhood of a vertex u, denoted N(u), is the set of every vertex v where u and v are neighbors

The degree of a vertex u, denoted deg(u), is the number of neighbors of vertex u. The maximum degree of graph G = (V, E), denoted max-deg(G), is the maximum, over vertex u in V, of deg(u)

Example:
Consider a graph G = (V, E), where V = {1, 2, 3, 4, 5} and E = {(1, 2), (1, 3), (2, 3), (3, 4)}.

This graph has five vertices and four edges

In this graph, vertices 1 and 3 are neighbors, vertices 1 and 4 are not neighbors, and vertices 4 and 5 are not neighbors. The neighborhood of vertex 3 is the set {1, 2, 4} and the neighborhood of vertex 5 is the empty set

The degrees of vertices in this graph are as follows:

deg(1) = 2
deg(2) = 2
deg(3) = 3
deg(4) = 1
deg(5) = 0

The maximum degree of this graph is max{2, 2, 3, 1, 0} = 3

Handshake Theorem and Corollaries

Handshake Theorem:
For any graph G = (V, E), the sum, over vertex u in V, of deg(u) equals twice the number of edges in E. In other words,

Sum of deg(u)'s = 2*|E|

Proof:
This theorem follows from the fact that each edge (u, v) contributes 1 to deg(u) and contributes 1 to deg(v). Thus, each edge contributes 2 to the sum of the deg(u)'s.

Handshake Corollary 1:
For any graph G = (V, E), the sum, over vertex u in V, of deg(u) is even. In other words,

Sum of deg(u)'s is even

Handshake Corollary 2:
For any graph G = (V, E), the number of vertices whose degrees are odd is even

Proof:

Let	S	= Sum, over vertex u in V, of deg(u)
	S1	= Sum, over vertex u whose degree is even, of deg(u)
	S2	= Sum, over vertex u whose degree is odd, of deg(u)

From	S	= S1 + S2	{from definitions of S, S1, S2}
	S	is even	{from Handshake Corollary 1}
	S1	is even	{from definition of S1}

Thus	S2	is even

Thus Handshake Corollary 2 follows from definition of S2

Subgraphs

A subgraph of a graph G = (V, E) is a graph G’ = (V’, E’) such that V’ is a subset of V and E’ is a subset of E

A subgraph G’ = (V’, E’) of a graph G = (V, E) is called induced iff every edge in E that connects two vertices in V’ is in E’

Example:
Let G = (V, E), where V = {1, 2, 3, 4, 5} and E {(1, 2), (1, 3), (2, 3), (3, 4)}, be a graph

The graph G’ = (V’, E’), where V’ = {2, 3, 4} and E’ = {(2, 3), (3, 4)}, is an induced subgraph of G

The graph G’ = (V’, E’), where V’ = {2, 3, 4} and E’ = {(2, 3), (2, 4)}, is not a subgraph of G (because E’ is not a subset of E) and so G’ is not an induced subgraph of G

The graph G’ = (V’, E’), where V’ = {2, 3, 4} and E’ = {(2, 3)} is a subgraph but not an induced subgraph of G (because the edge (3, 4) is in G but not in G’)

G’ = (V’, E’), where V’ is the empty set { } and E’ is the empty set { }, is not a graph. And so G’ is not a subgraph, and not an induced subgraph of G

Complete Graphs

A graph is complete iff every two distinct vertices in the graph are connected by an edge

A complete graph with n vertices is denoted Kn:

The number of edges in K1 = 0
The number of edges in K2 = 1
The number of edges in K3 = 3
The number of edges in K4 = 6
…

Theorem:
The number of edges in Kn is n*(n-1)/2

Proof (by direct inference):

```
                T
=>              {let Kn be a complete graph with n vertices and |E| edges;
                from Handshake Theorem, we have}
                2*|E| = Sum of deg(u)'s

=>              {for every vertex u in Kn, deg(u) = (n-1)}
                2*|E| = n*(n-1)

=>              {arithmetics}
                |E| = n*(n-1)/2
```

Graph Coloring

A coloring of a graph G is an assignment that assigns a color to each vertex in G such that no two neighboring vertices are assigned the same color.

A graph G is said to be k-colorable iff G can be colored using k or fewer colors

The chromatic number of a graph G is the smallest number of colors that can be used to color G

Theorem 1:
Any coloring of a graph with n vertices uses at most n colors

Theorem 2:
Any coloring of a complete graph Kn with n vertices uses at least n colors

Proof: (by contradiction)

```
            (Exist a coloring of Kn that uses less than n colors)
=>          {Kn has n vertices}
            (This coloring colors two vertices, u and v, with same color)

=>          {Kn is complete and so it has an edge (u, v)}
            F
```

Corollary of Theorems 1 and 2:
The chromatic number of a complete graph Kn with n vertices is n

Graph Coloring Theorem

Theorem:
Any graph G can be colored using max-deg(G) + 1 colors

Proof:
In order to prove this theorem by induction, we need to introduce a parameter into the statement of the theorem. Thus, we restate the theorem as follows: (All n, n >= 1, any graph G with n vertices can be colored using max-deg(G) + 1 colors). We can now prove the theorem by induction on n

Let P(n) be the predicate: any graph G with n vertices can be colored using max-deg(G) + 1

Base case: n = 1:
P(1) <=> {any graph G with 1 vertex can be colored using 1 color} T

Induction step:
Prove by direct inference that (All n, n >= 1, P(n) => P(n+1))

```
        P(n)
=>      {let G be any graph with (n+1) vertices and let u be any vertex in
        G; also let G’ be the graph that results from removing vertex u
        from graph G; from the induction hypothesis P(n), G’ which has n
        vertices can be colored using max-deg(G’) + 1 colors; thus graph
        G’ can be colored using max-deg(G) + 1 colors; because the
        degree of vertex u is at most max-deg(G), the neighbors of vertex
        u are colored using at most max-deg(G) colors; therefore there is
        at least one color that can be used to color vertex u from the max-
        deg(G) + 1 colors that are used to color the vertices of graph G’;
        thus graph G can be colored using max-deg(G) + 1 colors}
        P(n+1)
```

Checking whether any Graph G is 2-Colorable

Step 1:
Color Black any vertex u in the given graph G

Step 2:
Color Black each vertex in G whose shortest distance (i.e. #edges) from u is even

Step 3:
Color White each vertex in G whose shortest distance (i.e. #edges) from u is odd

Step 4:
If every two neighboring vertices in G are colored using distinct colors
then the given graph G is 2-colorable
else the given graph G is not 2-colorable

Application of Graph Coloring to Scheduling

The scheduling problem of assigning time slots to the classes offered in a department d can be viewed as a coloring problem of a certain graph Gd

Each vertex u in graph Gd corresponds to a class u that is offered in department d

There is an edge between every two vertices u and v in graph Gd iff there are students who are registered in the two corresponding classes u and v

The colors that can be used in coloring the vertices of graph Gd are the time slots that are assigned to the corresponding classes offered in department d

Paths, Circuits, and Cycles

A path in a graph G is a nonempty sequence of vertices in G such that every two adjacent vertices in the sequence are neighbors in G

A circuit in G is a path in G where the first and last vertices are the same

The length of a path or circuit is the number of edges in the path or circuit

A path or circuit is called simple iff it has no repeated edges

A cycle is a simple circuit that has no repeated vertices. (Note that not all simple circuits are cycles. For example, (1, 2, 3, 4, 5, 3, 1) is a simple circuit but not a cycle.)

A graph is connected iff it has a path between every two distinct vertices

An unconnected graph consists of two or more connected components

Example:
Consider a graph G = (V, E) where V = {1, 2, 3, 4, 5, 6, 7} and E = {(1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 5), (6, 7)}.

(1, 2, 1, 3, 5, 4, 3) is a (not simple) path of length 6 in graph G

(1, 2, 1, 3, 5, 4, 3, 1) is a (not simple) circuit of length 7 in graph G

(1, 2, 3, 4, 5, 3, 1) is a simple circuit but not a cycle of length 6 in graph G

Graph G has two simple cycles (1, 2, 3, 1) and (3, 4, 5, 3) of length 3 each

Graph G is unconnected because it has no path between vertices 1 and 6

Graph G consists of two connected components {1, 2, 3, 4, 5} and {6, 7}

Theorem of Odd Length Cycles

Theorem:
Every circuit of odd length has a cycle of odd length. (For example, a circuit (1, 2, 3, 4, 3, 1, 2, 1) of length 7 has a cycle (1, 2, 3, 1) of length 3.)

Proof: (by strong induction)
We need to introduce a parameter to the statement of the theorem. Thus, we restate the theorem as follows:

(All odd n, n >= 3, every circuit of length n has a cycle of odd length)

Let P(n) be the predicate: every circuit of length n has a cycle of odd length

Base case: n = 3:
P(3) <=> {every circuit of length 3 is a "triangle" and is a cycle} T

Induction step:
We need to prove by direct inference the following predicate
(All odd n, n >= 3, (P(3) and … and P(n)) => P(n+2))

Let C be any circuit of length (n+2) which is odd. There are two cases to consider:

Case 1: C has no repeated vertices and is a cycle and the proof is complete

Case 2: C has at least one vertex v that is repeated at least twice. Thus,

C = (u, x, ..., x, v, y, ..., y, v, z, ..., z, u)

In this case, C can be partitioned into two circuits C1 and C2, where

C1 = (u, x, …, x, v, z, …,z, u)
C2 = (v, y, …, y, v)

Length of C which is odd = length of C1 + length of C2

Without loss of generality, assume that length of C1, k, is odd. Thus, length of C1, k, is both odd and less than n+2

From the induction hypothesis, (P(3) and … and P(n)) is true and so P(k) is true. Thus, circuit C1 has a cycle of odd length and so circuit C has a cycle of odd length

Bipartite Graphs

A graph G = (V, E) is called bipartite iff set V can be partitioned into two subsets V1 and V2 such that for every edge (u, v) in E, u is in V1 and v is in V2

Example:
Let G = (V, E) be a graph where V = {1, 2, 3, 4} and E = {(1, 3), (1, 4), (2, 3), (2, 4)}.
Graph G is bipartite because V can be partitioned into V1 = {1, 2} and V2 = {3, 4}

Example:
Let G = (V, E) be a graph where V = {1, 2, 3, 4} and E = {(1, 2), (1, 3), (2, 3), (2, 4)}.
Graph G is not bipartite because no two of the three vertices 1, 2, and 3 can belong to the same partitions

Proving that a graph G = (V, E) is bipartite can be achieved by exhibiting a partitioning of set V

Proving that a graph G = (V, E) is not bipartite can be achieved, thanks to the next theorem, by showing that G has a cycle of odd length

Theorem of Bipartite Graphs

Theorem:
Let G be a graph. Then,
(G is bipartite) <=> (G is 2-colorable) <=> (Every cycle in G is of even length)

Proof of first <=> in forward direction: (by direct inference)
Because G is bipartite, its set of vertices can be partitioned into V1 and V2. All vertices in V1 can be colored Black and all vertices in V2 can be colored White. Thus, G is 2-colorable

Proof of first <=> in backward direction: (by direct inference)
Color the vertices in G using two colors, Black and White. The vertices of G can be partitioned into V1 (containing all Black vertices) and V2 (containing all White vertices). Thus, G is bipartite

Proof of second <=> in forward direction: (by contradiction)

```
        (G is 2-colorable) and (G has a cycle C of odd length)
=>      {cycle C cannot be colored using two colors}
        (G is 2-colorable) and (G cannot be colored using 2 colors)
=>      {P and not P = F}
        F
```

Proof of second <=> in backward direction: (by indirect inference)

```
        (G is not 2-colorable)
=>      {applying the 2-colorability check to G yields two neighboring vertices v
        and w that are colored with the same colors in G; the shortest distance
        between the initial vertex u and vertex v has the same parity, even or
        odd, as the shortest distance between vertex u and vertex w; length of
        circuit C involving vertices u, v, and w is of odd length; from the Theorem
        of Odd Length Cycles, circuit C has a cycle of odd length}
        (G has a cycle of odd length)
```

Complete Bipartite Graphs

Kp,q denotes a bipartite graph where the set of vertices can be partitioned into two subsets: a subset V1 of p vertices and a subset V2 of q vertices and where there is an edge from every vertex in V1 to every vertex in V2

Graph Kp,q is called a complete bipartite graph

Graph K1,1 has 2 vertices and 1 edge
Graph K1,2 has 3 vertices and 2 edges
Graph K2,2 has 4 vertices and 4 edges
Graph K2,3 has 5 vertices and 6 edges
…
Graph Kp,q has p+q vertices and p*q edges

Trees

A tree is a graph that (1) is connected and (2) has no cycle of length at least 3

A vertex whose degree is 1 is called a leaf

Example:
The graph (V, E) where V = {1, 2, 3, 4, 5} and E = {(1, 2), (2, 3), (3, 4), (3, 5)} is a tree that has three leaves, namely vertices 1, 4, and 5

Theorem of Bipartite Graphs and Trees

Theorem:
Every tree is a bipartite graph

Proof: (by direct inference)

```
        (Graph TR is a tree)
=>      {definition of tree}
        (Graph TR has no cycle of length at least 3)

=>      {length of a cycle is a non-negative integer}
        (All cycles of in TR are of length 0, 1, or 2)

=>      {All cycles of length 0, 1, or 2 are of even length}
        (All cycles in TR are of even length)

=>      {Theorem of Bipartite Graphs}
        (Graph TR is a bipartite graph)
```

Corollary:
Every tree is 2-colorable

Theorem of Tree Leaves

Theorem:
Every tree with at least 2 vertices has at least 1 leaf

Proof: (by contradiction)

```
        (Exist a tree TR that has at least two vertices but no leaves)
=>      {definitions of leaves and vertex degrees}
        (Every vertex in tree TR is of degree 2 or higher)

=>      {from Exercise 5 Problem 11}
        (Tree TR has a cycle)

=>      {no tree has a cycle}
        F
```

Theorem of Tree edges

Theorem:
Each tree with n vertices has (n-1) edges

Proof: (by induction on n)

Let P(n) be the predicate: any tree with n vertices has (n-1) edges

Base case: n = 1:
P(1) <=> {any tree with 1 vertex has no edges} T

Induction step:
A direct inference proof of (All n, n >= 1, P(n) => P(n+1)) proceeds as follows:

```
        P(n)
=>      {let TR be any tree with (n+1) vertices; from the Theorem of Tree Leaves,
        TR has a leaf u that is connected with an edge (u, v) to a tree TR' with n
        vertices; from the induction hypothesis P(n), tree TR' has (n-1) edges;
        thus, tree TR has n edges}
        P(n+1)
```

Planar Graphs

A graph is called planar iff (1) it has at least 3 edges, and (2) can be dawn on a plane without any two edges being crossed

Examples of planar graphs:

- Any tree with at least 3 edges
- Any complete graph Kn where n is 3 or 4
- Any complete bipartite graph Kp,q where p is at most 2 or q is at most 2

Examples of non-planar graphs:

- Any complete graph Kn where n is at least 5
- Any complete bipartite graph Kp,q where p is at least 3 and q is at least 3
- Any graph that has K5 or K3,3 as a subgraph

Regions

When a planar graph G is drawn on a plane, without any two edges being crossed, G partitions the plane into non-overlapping “regions”. (To go from any point in a region to any point t in another region, one has to cross at least one edge.)

The region degree of a region r, denoted rdeg(r), is the number of edges that border region r

Example:
The planar graph G = (V, E) where V = {1, 2, 3} and E = {(1, 2), (1, 3), (2, 3)} has two regions r1 and r2. Each of the two regions is bordered by all three edges of G. Thus, rdeg(r1) = 3 and rdeg(r2) = 3

Example:
The planar graph G = (V, E) where V = {1, 2, 3, 4, 5, 6} and E = {(1, 2), (1, 3), (1, 5), (2, 3), (2, 5), (3, 4), (5, 6)} has three regions r1, r2, and r3.

Region r1 is bordered by the three edges (1, 2), (1, 3), and (2, 3). Thus, rdeg(r1) = 3

Region r2 is bordered by the five edges (1, 3), (1, 5), (2, 3), (2, 5), and (3, 4). Thus, redeg(r2) = 5

Region r3 is bordered by the four edges (1, 2), (1, 5), (2, 5), and (5, 6). Thus, rdeg(r3) = 4

Example:
The planar graph G = (V, E) where V = {1, 2, 3, 4, 5} and E = {(1, 2), (1, 3), (3, 4), (3, 5)} is a tree and so it has only one region that is bordered by all the edges in the graph

Region Degree Theorem

Theorem:
For each region r in a planar connected graph, rdeg(r) >= 3

Region Handshake Theorem

Theorem:
For each planar graph G = (V, E),

Sum of rdeg(r)’s in G = (number of edges in G whose two sides belong to same region) + 2*(number of edges in G whose two sides belong to distinct regions)

Proof:
This theorem follows from the fact that two sides of each edge in graph G either belong to the same region r or belong to two distinct regions r1 and r2. In the first case, the edge contributes 1 to rdeg(r). And in the second case, the edge contributes 1 to rdeg(r1) and contributes 1 to rdeg(r2)

Corollary:
For each planar graph G = (V,E),

|E| =< Sum of rdeg(r)’s in G =< 2*|E|

Euler's Formula

Theorem:
For each planar connected graph G = (V, E),

|V| - |E| + |R| = 2

where |R| is the number of regions in graph G

Proof: (by Case Analysis)

Case 1: (graph G has no cycles):
In this case, G is a tree. Thus, |V| = n, |E| = (n-1), and |R| = 1, and so

|V| - |E| + |R| = 2

Case 2: (graph G has at least one cycle C):
In this case, the proof proceeds by induction on the number e of edges in graph G

Let P(e) be the predicate: for any planar connected graph G = (V, E) that has e edges, |R| regions, and one cycle C, |V| - |E| + |R| = 2

Base case: (e = 3):
P(3) <=> {graph G is a "triangle" where |V| = 3, |E| = 3, and |R| = 2 and so |V| - |E| + |R| = 2} T

Induction step:
A direct inference proof of (All e, e >= 3, P(e) => P(e+1)) proceeds as follows:

P(e)
=> {let G = (V, E) be a planar connected graph that has (e+1) edges and at least one cycle C; let G' = (V, E') be the graph that results after removing one edge from cycle C in graph G; note that G' is a planar connected graph; also note that |E| = |E'|+1 and |R| = |R'|+1; if G' has no cycles then by Case 1 above, (|V| - |E'| + |R'| = 2) and so (|V| - |E| + |R| = 2); on the other hand, if G' has at least one cycle then by the induction hypothesis P(e), (|V| - |E'| + |R'| = 2) and so (|V| - |E| + |R| = 2)}
P(e+1)

Euler’s Corollary

Theorem:
For each planar connected graph G = (V, E), (3*|V| - 6) >= |E|

Proof: (by direct inference)

```
            T
=>          {Region Degree Theorem}
            For each region r in G, rdeg(r) >= 3

=>          {Region Handshake Theorem}
            3*|R| =< Sum of rdeg(r)’s =< 2*|E|

=>          {Euler’s Formula: |R| = (2 - |V| + |E|)}
            3*(2 - |V| + |E|) =< 2*|E|

=>          {arithmetics}
            (3*|V| - 6) >= |E|
```

Theorem of Small Vertex Degrees in Planar Graphs

Theorem:
Any planar connected graph has at least one vertex whose degree is at most 5

Proof: (by contradiction)

(Exist a planar connected graph G = (V, E) where every vertex is of degree 6 or larger)

=> {Handshake Theorem}

2*|E| = Sum of deg(u)'s >= 6*|V|

=> {Euler's Corollary}

(|E| >= 3*|V|) and (|E| =< 3*|V| - 6)

=> {P and not P = F}

F

Theorem of Planar Graph Coloring

Theorem:
Any planar connected graph with n vertices, where n is at least 6, is 6-colorable

Proof: (by strong induction on n)
Let P(n) be the predicate: Any planar connected graph with n vertices, where n is at least 6, is 6-colorable

Base case: n = 6:
P(6) <=> {any graph with 6 vertices is 6-colorable} T

Induction step:
A direct inference proof of (All n, n >= 6, (P(6) and P(7) and … and P(n)) => P(n+1)) proceeds as follows:

(P(6) and P(7) and … and P(n))
=> {let G be a planar connected graph with (n+1) vertices and let u be a vertex of degree at most 5 in G; note that the existence of vertex u in graph G is guaranteed by the Theorem of Small Vertex Degrees in Planar Graphs; also let G' be the graph that results from removing vertex u and its incident edges from graph G; graph G' consists of one or more connected components; each connected component either has 5 or fewer vertices or is a planar connected graph where the number of vertices is at least 6 and at most n; in the first case the component can be colored using 6 or fewer colors and in the second case the component is 6-colorable by the induction hypothesis; because each component of G' is 6-colorable, G' is 6-colrable; because vertex u has at most 5 neighbors in G', one of the 6 colors that are used to color G' can be used to color vertex u; thus G is 6-colorable}
P(n+1)

Proving Planarity and Non-Planarity

To prove that a given graph G is planar, show that G can be drawn on a plane without any pair of its edges being crossed. To prove that G is non-planar, use a by-contradiction proof

Theorem:
The complete graph K5 is non-planar

Proof: (by contradiction)

```
        (K5 = (V, E) is a planar connected graph)
=>      {definition of K5 and Region Degree Theorem}
        (|V| = 5, |E| = 10, and for each region r in K5, rdeg(r) >= 3)

=>      {Euler's Formula}
        |R| = 2 - |V| + |E| = 7

=>      {Region Handshake Theorem}
        20 = (2*|E|) >= (3*|R|) = 21

=>      {20 >= 21 is false}
        F
```

Theorem:
The complete bipartite graph K3,3 is non-planar

Proof: (by contradiction)

```
        (K3,3 = (V, E) is a planar connected graph)
=>      {definition of K3,3}
        (|V| = 6, |E| = 9, and for each region r in K3,3, rdeg(r) >= 4)

=>      {Euler's Formula}
        |R| = 2 - |V| + |E| = 5

=>      {Region Handshake Theorem}
        18 = (2*|E|) >= (4*|R|) = 20

=>      {18 >= 20 is false}
        F
```

6 Sets

A set is an unordered collection of distinct elements

That an element s is in a set A is defined by the predicate (x in A)

A set that has no element is called empty set and is denoted { }

A set A is a subset of a set B, denoted (A sub B), iff

    (All x, (x in A) => (x in B))

Two sets A and B are equal, denoted (A = B), iff

    (All x, (x in A) <=> (x in B))

The following predicates hold:

    ({ } sub A)
    (x in A) <=> ({x} sub A))
    (A = B) <=> ((A sub B) and (B sub A))

The power set of a set A, denoted PS(A), is the set that contains as elements all subsets of A:

    (All x, (x sub A) <=> (x in PS(A))

Five Operators on Sets

The five operators are

union	denoted v
intersection	denoted ^
set difference	denoted -
Cartesian product	denoted *
complement	denoted c

These five operators are defined as follows:

(x in (A v B)) <=> ((x in A) or (x in B))

(x in (A ^ B)) <=> ((x in A) and (x in B))

(x in (A - B)) <=> ((x in A) and not (x in B))

((x, y) in (A * B)) <=> ((x in A) and (y in B))

(x in c(A)) <=> (not (x in A))

Cardinality of a Finite Set

The cardinality of a finite set A, denoted |A|, is the number of elements in A

|{ }| = 0

(A sub B) => (|B - A| = |B| - |A|)
Note that the first - denotes “set difference” whereas the second - denotes “integer subtraction”

|A v B| = |A| + |B| - |A ^ B|

|A ^ B| = |A| + |B| - |A v B|

|A - B| = |A| - |A ^ B|
Note that the first - denotes “set difference” whereas the second - denotes “integer subtraction”

|A * B| = |A| * |B|
Note that the first * denotes “Cartesian product” whereas the second * denotes “integer multiplication”

Proving (A sub B)

To prove

(predicate P) => (A sub B)

use a direct inference proof of the following form:

```
            x in A
=>      {hint on why (P and (x in A) => P1)}
            P1
…
            Pk
=>      {hint on why (P and Pk => (x in B))}
            x in B
```

Example 1 on Proving (A sub B)

Prove

((A sub B) and (B sub C)) => (A sub C)

Proof by direct inference:

```
                x in A
=>      {(A sub B)}
                x in B
=>      {(B sub C))}
                x in C
```

Example 2 on Proving (A sub B)

Prove

```
                    (A sub B)  <=>  (PS(A) sub PS(B))
```

Proof of (A sub B) => (PS(A) sub PS(B)) by direct inference

```
                    x in PS(A)
=>        {definition of PS(A)}
                    x sub A
=>        {(A sub B) and previous example}
                    x sub B
=>        {definition of PS(B)}
                    x in PS(B)
```

Proof of (A sub B) <= (PS(A) sub PS(B)) by direct inference

```
                    x in A
=>        {(x in A) <=> ({x} sub A)}
                    {x} sub A
=>        {definition of PS(A)}
                    {x} in PS(A)
=>        {PS(A) sub PS(B)}
                    {x} in PS(B)
=>        {definition of PS(B)}
                    x in B
```

Proving (A = B)

1. To prove

```
                    (predicate P)  =>  (A = B)
```

use a two-sided inference proof of the following form:

```
                    x in A
<=>       {hint on why (P and (x in A) <=> P1)}
                    P1
…
                    Pk
<=>       {hint on why (P and Pk <=> (x in B))}
                    x in B
```

2. To prove

```
                    (predicate P)  =>  (A = B)
```

use two direct inference proofs to prove the two predicates:

```
                    (predicate P)  =>  (A sub B)
                    (predicate P)  =>  (B sub A)
```

Example 1 on Proving (A = B)

Prove

c(c(A)) = A

Proof by two-sided inference:

```
            x in c(c(A))
<=>    {definition of c}
            not (x in c(A))
<=>    {definition of c}
            not (not (x in A))
<=>    {(not (not (P))) <=> P}
            x in A
```

Example 2 on Proving (A = B)

Prove

c(A ^ B) = c(A) v c(B)

Proof by two-sided inference:

```
            x in c(A ^ B)
<=>     {definition of c}
            not (x in (A ^ B))
<=>     {definition of ^}
            not ((x in A) and (x in B))
<=>     {De-Morgan's}
            not (x in A)  or  not (x in B)
<=>     {definition of c}
            (x in c(A)) or (x in c(B))
<=>     {definition of v}
            x in (c(A) v c(B))
```

7 Functions

A function f from set A to set B, denoted f: A -> B, is an assignment that assigns to every element x in A, a unique element, denoted f(x), in B.

Set A is called domain of function f
Set B is called codomain of function f

Element x is called pre-image of element f(x)
Element f(x) is called image of element x

The following predicate holds:

```
Let f: A -> B be a function, then

(All x1, x2 in A,
        (x1 = x2)  =>  (f(x1) = f(x2))
)
```

Range of a Function

Let f: A -> B be a function. Then, range (f) is the set of every element y in B such that there is an element x in A where f(x) = y.

(All y in B, (y in range(f)) => (Exist x in A, f(x) = y))

Example:
Let A be the set of all integers. Define f: A -> A to be the function f(x) = x^2. Then,

Range (f) = {0^2, 1^2, 2^2, …}

Types of Functions

A function f: A -> B is injective or 1-1 iff

(All x1, x2 in A, (f(x1) = f(x2)) => (x1 = x2))

A function f: A -> B is surjective or onto iff

(All y in B, (Exist x in A, f(x) = y))

A function f: A -> B is bijective iff

f is both injective and surjective

Bijective functions are good because they have inverses …

Inverse Functions

Let f: A -> B be a bijective function. Define function f’: B -> A as follows:

(All x in A, f’(f(x)) = x)

Theorem:
Let f: A -> B be a bijective function. Function f’ is bijective and satisfies the predicate:

(All y in B, f(f’(y)) = y)

Function f’ is called inverse of function f
Function f is called inverse of function f’

Function Composition

Let f: A -> B and g: B -> C be two functions. The composition of f and g is a function from A to C denoted (g.f) and defined as follows:

(All x in A, (g.f)(x) = g(f(x)))

Proving (f is injective)

To prove

```
                    (predicate P)  =>  (f is injective)
```

use a direct inference proof of the following form:

```
                    f(x1) = f(x2)
=>        {hint on why (P and (f(x1) = f(x2)) => P1)}
                    P1
…
                    Pk
=>        {hint on why (P and Pk => (x1 = x2))}
                    x1 = x2
```

Proving (f is surjective)

To prove

(predicate P) => (f is surjective)

use a direct inference proof of the following form:

```
          y in B
=>    {hint on why (P and (y in B) => P1)}
          P1
...
          Pk
=>    {hint on why (P and Pk => (Exist x in A, f(x) = y))}
          Exist x in A, f(x) = y
```

Example 1 on Proving Functions

Prove

((f is injective) and (g is injective)) => ((g.f) is injective)

Proof by direct inference:

((g.f)(x1)) = ((g.f)(x2))

=> {definition of "."}

(g(f(x1)) = g(f(x2))

=> {g is injective}

f(x1) = f(x2)

=> {f is injective}

x1 = x2

Example 2 on Proving Functions

Prove

((g.f) is bijective) => ((f is injective) and (g is surjective))

Proof of (f is injective) by direct inference:

f(x1) = f(x2)

=> {g is a function}

g(f(x1)) = g(f(x2))

=> {definition of "."}

(g.f)(x1) = (g.f)(x2)

=> {(g.f) is bijective and so injective}

x1 = x2

Proof of (g is surjective) by direct inference:

z in C

=> {(g.f) is bijective and so surjective}

Exist x in A, (g.f)(x) = z

=> {definition of "."}

Exist x in A, y in B, f(x) = y and g(y) = z

=>{ {simplify}

Exist y in B, g(y) = z

8 Recurrences

A recurrence is an infinite sequence of real numbers

R(0), R(1), R(2), …

Where R(n) denotes the maximum number of steps that an associated recursive algorithm executes when its input is of length n.

A recurrence can be specified by a recurrence equation or by a closed equation

A recurrence equation is of the form:

R(0)	=	some real number	
R(n+1)	=	f(R(n))	for n >= 0

or it is of the form:

R(0)	=	some real number	
R(1)	=	some real number	
R(n+2)	=	f(R(n+1), R(n))	for n >= 0

and so on.

A closed equation is of the form:

R(n)	=	f(n)	for n>= 0

How to get the closed equation of a recurrence from the recurrence equation of the same recurrence?

How to Get the Closed Equation?

Method 1:

1. Start with the recurrence equation.
2. Use Poor-man's Induction to derive a maybe-correct closed equation.
3. Use Induction to verify that the maybe-correct closed equation is indeed correct.

Method 2:

1. Start with the recurrence equation.
2. Use the Characteristic Polynomial method to derive the correct closed equation.

Use of Poor-man’s Induction

Consider the recurrence equation:

R(0)	=	5		(1)
R(n+1)	=	3 * R(n)	for n >= 0	(2)

From (1) and (2), derive a maybe-correct closed equation using Poor-man’s induction:

```
R(n)   =   {from (2)}
           3 * R(n-1)

       =   {from (2)}
           3^2 * R(n-2)

       =   {from (2)}
           3^3 * R(n-3)

       =   {using Poor-man’s Induction}
           3^i * R(n-i)                    for i >= 1

       =   {choosing i to be n}
           3^n * R(0)

       =   {from (1)}
           3^n * 5
```

The resulting closed equation (R(n) = 3^n * 5) still needs to be proven correct using Induction.

Use Induction

Prove that from the recurrence equation

R(0) = 5 (1)

R(n+1) = 3 * R(n) for n >= 0 (2)

we can infer the closed equation

R(n) = 3^n * 5 for n >= 0 (3)

Proof: (by Induction):

Let P(n), for n >= 0, be the predicate (R(n) = 3^n * 5) (4)

Base case: (n=0):

P(0) <=> {from (4)} R(0) = 3^0 * 5 = 5 <=> {from (1)} T

Induction step:

Prove for n >= 0, P(n) => P(n+1)

```
        P(n)
=>      {definition of P(n)}
        R(n) = 3^n * 5

=>      {from (2)}
        R(n+1) = 3 * 3^n * 5 = 3^(n+1) * 5

=>      {definition of P(n+1)}
        P(n+1)
```

Use of Characteristic Polynomial – One Root

Consider the recurrence equation:

R(0)	=	5		(1)
R(n+1)	=	3 * R(n)	for n >= 0	(2)

From (2), the characteristic polynomial is

r = 3

This polynomial has only one root r1 = 3 (3)

From (3), the closed equation is

R(n) = x * (r1)^n (4)

From (1), (3), and (4), we get x:

5 = x * (3)^0

5 = x (5)

From (3), (4), and (5), we get the closed equation:

R(n) = 5 * (3)^n

Use of Characteristic Polynomial – Two Distinct Roots

Consider the recurrence equation:

R(0)	=	4		(1)
R(1)	=	3		(2)
R(n+2)	=	3*R(n+1) + 4*R(n)	for n >= 0	(3)

From (3), the characteristic polynomial is

r^2	=	3*r + 4	
0	=	r^2 - 3*r – 4	
0	=	(r+1)*(r-4)	

This polynomial has two distinct roots r1 = -1 and r2 = 4 (4)

From (4), the closed equation is

R(n)	=	x * (r1)^n + y * (r2)^n	(5)

From (1), (2), (4), and (5), we get two equations in x and y:

4	=	x + y	(6)
3	=	-x + 4*y	(7)

From (6) and (7), we get x and y:

x	=	13/5	(8)
y	=	7/5	(9)

From (4), (5), (8), and (9), we get the closed equation:

R(n)	=	13/5 * (-1)^n + 7/5 * (4)^n

Use of Characteristic Polynomial – Two Identical Roots

Consider the recurrence equation:

$$R(0) = 6 \qquad (1)$$

$$R(1) = 8 \qquad (2)$$

$$R(n+2) = 4*R(n+1) - 4*R(n) \qquad \text{for } n >= 0 \qquad (3)$$

From (3), the characteristic polynomial is

$$r^2 = 4*r - 4$$

$$0 = r^2 - 4*r + 4$$

$$0 = (r-2)^2$$

This polynomial has two identical roots r1 = 2 and r2 = 2 (4)

From (4), the closed equation is

$$R(n) = x*((r1)^n) + y*n*((r2)^n) \qquad (5)$$

From (1), (4), and (5), we get x:

$$x = 6 \qquad (6)$$

From (2), (4), (5) and (6), we get y:

$$y = -2 \qquad (7)$$

From (4), (5), (6), and (7), we get the closed equation:

$$R(n) = 6*(2^n) - 2*n*(2^n)$$

Limitations of the two methods

Induction Method:
This method can be used only when the right-hand side of the recurrence part of the recurrence equation has only one R(n). For example, this method can be used when the recurrence equation is:

$$\begin{aligned} R(1) &= 0 \\ R(n+1) &= R(n) + n \end{aligned}$$

and can't be used when the recurrence equation is:

$$\begin{aligned} R(0) &= 0 \\ R(1) &= 1 \\ R(n+2) &= R(n+1) + R(n) \end{aligned}$$

Characteristic Polynomial Method:
can be used when the each term on the right hand side of the recurrence part of the recurrence equation is of the form "value*R(n+i)". For example this method can be used when the recurrence equation is:

$$\begin{aligned} R(0) &= 0 \\ R(1) &= 1 \\ R(n+2) &= R(n+1) + R(n) \end{aligned}$$

and can't be used when the recurrence equation is:

$$\begin{aligned} R(1) &= 0 \\ R(n+1) &= R(n) + n \end{aligned}$$

Case Study: Sorting Algorithm

Consider an algorithm for recursively sorting an array x in an ascending order using interchange steps.

In an interchange step, the values of two neighboring elements x[i] and x[i+1] in array x are compared, and the value of x[i] is found to be larger than the value of x[i+1], and so the values of x[i] and x[i+1] are interchanged.

Let R(n) denote the maximum number of interchange steps that need to be executed to sort array x with n elements. Thus, R(1) = 0, R(2) = 1, R(3) = 3, R(4) = 6, and so on.

The value of R(n) can be defined by the recurrence equation:

$$R(1) = 0 \qquad (1)$$

$$R(n+1) = R(n) + n \qquad \text{for } n >= 1 \qquad (2)$$

We use the Induction method to derive the closed equation for the sorting algorithm.

Deriving the Closed Equation for Sorting Algorithm

From (1) and (2), derive a maybe-correct closed equation using Poor-man's induction:

$R(n)$ = {from (2)}
$R(n-1) + (n-1)$

= {from (2)}
$R(n-2) + (n-2) + (n-1) = R(n-2) + 2*n - (2+1)$

= {from (2)}
$R(n-3) + 3*n - (3+2+1)$

= {using Poor-man's Induction}
$R(n-i) + i*n - (i+\ldots+2+1)$ for $i \geq 1$

= {choosing i to be (n-1)}
$R(1) + (n-1)*n - ((n-1)+\ldots+2+1)$

= {from (1)}
$(n-1)*n - ((n-1)*n/2)$

= $(n-1)*n/2$

The resulting closed equation ($R(n) = (n-1)*n/2$) still needs to be proven correct using Induction.

Verifying Closed Equation of Sorting Algorithm

Prove that from the recurrence equation

R(1)	=	0		(1)
R(n+1)	=	R(n) + n	for n >= 1	(2)

we can infer the closed equation

R(n)	=	(n-1)*n/2	for n >= 1	(3)

Proof: (by Induction):
Let P(n), for n >= 1, be the predicate (R(n) = (n-1)*n/2) (4)

Base case: (n=1):
P(1) <=> {from (4)} R(1) = 0 <=> {from (1)} T

Induction step:
Prove for n >= 1, P(n) => P(n+1)

```
        T
=>      {from (2)}
        R(n+1) = R(n) + n

=>      {from definition of P(n) in (4)}
        R(n+1) = ((n-1)*n/2 + n) = (n*(n+1)/2)

=>      {from definition of P(n+1) in (4)}
        P(n+1)
```

Case Study: Fibonacci Numbers

The Fibonacci numbers are defined by the recurrence equation:

$R(0)$	=	0		(1)
$R(1)$	=	1		(2)
$R(n+2)$	=	$R(n+1) + R(n)$	for $n >= 0$	(3)

From (1), (2), and (3), we derive the closed equation using the Characteristic Polynomial method.

From (3), the characteristic polynomial is:

$r^2 = r + 1$

This polynomial has two distinct roots:

r1	=	(-b + sqrt (b^2 – 4ac))/2a	=	(1+sqrt(5))/2	(4)
r2	=	(-b – sqrt (b^2 - 4ac))/2a	=	(1-sqrt (5))/2	(5)

From (4) and (5), the closed equation is:

R(n) = x * (r1)^n + y * (r2)^n (6)

From (1), (2), (4), (5), and (6), x and y are:

x	=	1/sqrt(5)	(7)
y	=	-1/sqrt(5)	(8)

From (4), (5), (6), (7), and (8), the closed equation is:

R(n) = 1/sqrt(5) * (((1 + sqrt(5))/2)^n - ((1 – sqrt(5))/2)^n)

9 Big Notation

Consider a program that takes an input of length n, executes at most $f(n)$ steps, and produces an output then terminates.

A step is defined vaguely as the addition, subtraction, multiplication, division, or comparison of two integers.

Function $f(n)$ cannot be defined accurately. So it is intentionally defined vaguely.

Be “sloppy” in describing function $f(n)$ by ignoring smaller terms and factors.

Examples:
Write $f(n) = (2n + 5)$ as $f(n) = (n)$
Write $f(n) = (2*(n^2) + 5*n + 6)$ as $f(n) = (n^2)$
Write $f(n) = (2^n + n^2)$ as $f(n) = (2^n)$

Big O, Omega, Theta Notation

Let f(n) and g(n) be two functions from integers to real numbers.

```
f(n) is O(g(n)) iff
    (Exist positive real numbers K and C,
        (All integer n, n>K, |f(n)| =< C*|g(n)|)
    )
```

```
f(n) is Omega(g(n)) iff
    (Exist positive real numbers K and C,
        (All integer n, n>K, |f(n)| >= C*|g(n)|)
    )
```

```
f(n) is Theta(g(n)) iff
    (the following two conditions hold:
        f(n) is O(g(n)) and
        f(n) is Omega(g(n))
    )
```

Example 1 on Big Notation

Show by direct inference that the function f(n) = 10 is Theta(g(n)) where g(n) = 1

Proving f(n) is O(g(n)):

|f(n)|	=	|10|	
	=	10	
	=	10*1	
	=	C*|1|	for C = 10
	=<	C*|g(n)|	for any K and C = 10

Proving f(n) is Omega(g(n)):

|f(n)|	=	|10|	
	=	10	
	=	10*1	
	=	C*|1|	for C = 10
	>=	C*|g(n)|	for any K and C = 10

Example 2 on Big Notation

Show by direct inference that the function f(n) = (3*n – 7) is Theta(g(n)) where g(n) = n

Proving f(n) is O(g(n)):

$$\begin{aligned}
|f(n)| \quad & = && |3*n - 7| && \\
& = && 3*n - 7 && \text{for } n > 3 \\
& =< && 3*n && \\
& = && C*|n| && \text{for } C = 3 \\
& =< && C*|g(n)| && \text{for } K = 3 \text{ and } C = 3
\end{aligned}$$

Proving f(n) is Omega(g(n)):

$$\begin{aligned}
|f(n)| \quad & = && |3*n - 7| && \\
& = && 3*n - 7 && \text{for } n > 3 \\
& >= && 2*n && \text{for } n > 7 \\
& = && 2*|n| && \\
& = && C*|n| && \text{for } C = 2 \\
& >= && C*|g(n)| && \text{for } K = 7 \text{ and } C = 2
\end{aligned}$$

Example 3 on Big Notation

Show by direct inference that the function f(n) = (n^2 – n – 1) is Theta(g(n)) where g(n) = n^2

Proving f(n) is O(g(n)):

|f(n)|	=	|n^2 – n - 1|	
	=	n^2 – n – 1	for n > 2
	=<	n^2	
	=	1*|n^2|	
	=	C*|n^2|	for C = 1
	=<	C*|g(n)|	for K = 2 and C = 1

Proving f(n) is Omega(g(n)):

|f(n)|	=	|n^2 – n - 1|	
	=	n^2 – n -1	for n > 2
	=	(n^2)/2 + (n^2)/2 – n – 1	
	>=	(n^2)/2	for n > 4
	=	1/2 * |n^2|	
	=	C*|n^2|	for C = 1/2
	=	C*|g(n)|	for K = 4 and C = 1/2