Chapter 1
Logico-Mathematical Background

J. C. Huang
Department of Computer Science
University of Houston
This chapter contains materials useful for

- interpreting the program specification correctly,
- determining if any part of the program specification is violated (i.e., not satisfied),
- proving that a certain assertion is a theorem (i.e., always true),
- arguing for (or against) the correctness of a given program,
- relating a subfunction in the specification to the corresponding subprogram, and
- finding an input to test-execute a specific subprogram.
Motivating question:

Suppose the specification of the payroll program that is used to print your paycheck contains the following statement:

"If the employee is a US citizen, compute the social security tax and deduct it from the gross amount."

Now suppose that you find an instance of social-security-tax deduction on the pay check of a non-citizen. Can you conclude that the program is incorrect?
Another motivating question:

Suppose the requirements of a computerized decision support system include the following statement:

"If the drug passes both animal test and clinical test, then the company will market it if and only if it can be produced and sold profitably and the government does not intervene."

Now further suppose that the product failed to pass the clinical test and this system supports the company to market the product. Does the system violate this requirement?
The Propositional Calculus

A *proposition* is a declarative sentence that is either true or false. For example,

- Harvard is a private university.
- $x + y = y + x$
- Eleven is divisible by three.
- The number 4 is a prime number.

are propositions. The first two sentences are true, and the last two false.
Connectives

Given propositions, we can form new propositions by combining them with connectives such as

- negation: ¬, not
- conjunction: ∧, and
- disjunction: ∨, or
- implication: ⊃, implies, if ... then ...
- equivalence: ≡, ... if and only if ...
Definition of the connectives:

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>\neg p</th>
<th>p \land q</th>
<th>p \lor q</th>
<th>p \supset q</th>
<th>p \equiv q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Well-formed formula:

A well-formed formula (wff) in the language of the propositional calculus is a syntactically correct expression. It is composed of connectives, propositional variables (such as p, q, r, s, ...), constants (T and F), and parentheses.
The syntax of a wff

is recursively defined as follows:

(1) A propositional variable standing alone is a wff.
(2) If $\alpha$ is a wff then $\neg(\alpha)$ is a wff.
(3) If $\alpha$ and $\beta$ are wffs then $(\alpha) \land (\beta)$, $(\alpha) \lor (\beta)$,
    $(\alpha) \supset (\beta)$, and $(\alpha) \equiv (\beta)$ are wffs.
(4) Those and only those obtained by rules 1, 2, and 3
    are wffs.
The precedence relation:

A wff obtained by the above definition may contain many parentheses and thus not suitable for human consumption. The use of parentheses can be reduced by using the following precedence (listed in the descending order):

\[ \neg, \land, \lor, \supset, \equiv \]
Truth Table

The *truth table* of a wff lists the truth values of the formula for all possible combinations of assignments to the values of variables involved. For example:

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p ⊃ q</th>
<th>p ∧ (p ⊃ q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Practical application

• In practice, a rule or specification is stated as a proposition.

• A rule is not violated, or a specification is satisfied, if it is evaluated to be true.

• Analysis of a statement can often be facilitated by translating it into a well-formed formula first.
Example of analysis

• The requirements of a computerized decision support system include the following statement:

  If the drug passes both animal test and clinical test, then the company will market it if and only if it can be produced and sold profitably and the government does not intervene.

• Now suppose that the product failed to pass the clinical test and this system support the company to market the product. Dose this system violate the above requirement?
Example (continued)

First, translate the above statement into the well-formed formula shown below:

\[ A_1: \ a \land c \supset (m \equiv p \land \neg g) \]

where
- a: the drug passes animal test
- c: the drug passes clinical test
- m: the company will market the drug
- p: the drug can be produced and sold profitably
- g: the government intervenes
Example (continued)

It is obvious that, if the drug failed the clinical test, i.e., if $c$ is false, then the formula is true regardless of the truth values of other variables.

$$a \land c \supset (m \equiv p \land \neg g)$$

? F F T ?

Hence the requirement is not violated.
Example (continued)

Note, however, that this requirement is ambiguous. It can also be interpreted as follows:

\[ A_2: \ a \land c \supset m \equiv p \land \neg g. \]

\[ \begin{array}{ccc}
F & F & T \\
F & F & ? \\
F & T & ? \\
T & ? & ? \\
T & ? & ? \\
\end{array} \]

In this case, there is insufficient data to determine if this requirement is violated.
Tautology and contradiction

**Definition:** If for every assignment of values to its variables a wff has the value T, it is said to be a *tautology*; if it always has the value F then it is said to be a *contradiction*.

**Notation:** If $A$ is a tautology, we write $\vdash A$.

Note that $A$ is a tautology if and only if $\neg A$ is a contradiction.
Relations among propositions

In practice, we often have to deal with a set of propositions, and it is useful to define the following two relations among them.
Logical equivalence

**Definition:** Two wffs $A$ and $B$ are said to be *logically equivalent* if and only if they have the same truth table.

**Theorem:** $A$ and $B$ are logically equivalent if and only if $\vdash A \equiv B$. 
Logical consequence

**Definition:** B is a *logical consequence* of A (denoted by $A \vdash B$) if for each assignment of truth values to the variables of A and B such that A has the value T then B also has the value T.

**Theorem:** $A \vdash B$ if and only if $\vdash \neg A \supset B$. 
Logical consequence

The antecedent $A$ may consist of a set of propositions. In that case, we have

**Theorem:** $A_1, A_2, \ldots, A_n \vdash B$ if and only if

$\vdash A_1 \land A_2 \land \ldots \land A_n \supset B$. 
How to prove that $A \supset B$ is a theorem?

By the definition of the implication ($\supset$) connective, $A \supset B$ can be false only if $A$ is true and $B$ is false.

Hence, a common technique for showing $\neg A \supset B$ is to show that $A$ cannot be true if $B$ is false.
Multiple Antecedents

In practical applications, the antecedent often consists of a conjunction of n parts: $A_1, A_2, \ldots, A_n$. In that case,

• to show that the assumption is consistent is to show that $A_1 \land A_2 \land \ldots \land A_n$ is satisfiable, and

• to show that $A_1 \land A_2 \land \ldots \land A_n \implies B$ is a tautology is to let $B$ be false and show that it is impossible to make all $A_i$'s true at the same time.
Applications

In applications where we have to make a decision to satisfy a number of constraints, a relation of the type \( \vdash A_1 \land A_2 \land \ldots \land A_n \supset B \) is useful. Let \( A_i \)'s be the constraints and \( B \) be the proposed decision. \( B \) is a valid decision if

\[ \vdash A_1 \land A_2 \land \ldots \land A_n \supset B \]

because

• Decision \( B \) leads to satisfaction of all constraints.
• Decision to the contrary (i.e., \( \neg B \)) leads to violation of a rule or contradiction of a fact.
Applications (continued)

• On the other hand, if B is not a logical consequence of $A_i$'s, a decision to the contrary, i.e., $\neg B$, could be a valid decision because in that case it is possible to make B false without violating any $A_i$. 
Example

Now let us get back to the previous example statement that says

"If the drug passes both animal test and clinical test, then the company will market it if and only if it can be produced and sold profitably and the government does not intervene"

which can be translated into a wff as

\[ A_2: \quad a \land c \supset m \equiv p \land \neg g \]
Example (continued)

Now let us suppose that a further reading of the specification led us to the discovery of the following policy:

"if the drug cannot be produced and sold profitably, the company should not market it"

i.e.,

\[ A_3: \neg p \supset \neg m \]
Example (continued)

and a regulatory fact that

"if the drug failed the animal test or clinical test, and the drug is marketed, the government will intervene"

i.e.,

\[ A_4: \ (\neg a \lor \neg c) \land m \supset g \]
Example (continued)

Also recall that the drug failed the clinical test, i.e.,

\[ A_5: \neg c \]

Now if we can show that \( \vdash A_2 \land A_3 \land A_4 \land A_5 \supset B \), where B is \( \neg m \), it means that all constraints can only be satisfied at the same time by not marketing the drug.

That is to say, the company cannot market the drug without violating at least one rule or contradicting a fact.
In other words, if we let

- $A_2: \quad a \land c \supset m \equiv p \land \neg g$
- $A_3: \quad \neg p \supset \neg m$
- $A_4: \quad (\neg a \lor \neg c) \land m \supset g$
- $A_5: \quad \neg c$

--------------------------------

$B: \quad \neg m$

Is $A_2 \land A_3 \land A_4 \land A_5 \supset B$ a tautology?
Example (continued)

It turns out that $A_2 \land A_3 \land A_4 \land A_5 \supset B$ is a tautology.

It can be proved by letting $B$ to be false, and see if we can make all $A_i$'s true at the same time as demonstrated in the following.
Example (continued)

1. Let $m \leftarrow T$ so that $B$ becomes false.

   $B: \neg m$
Example (continued)

1. Let $m \leftarrow T$ so that $B$ becomes false.
2. Let $c \leftarrow F$ to make $A_5$ true.

\[ A_5 : \neg c \]
Example (continued)

1. Let $m \leftarrow T$ so that $B$ becomes false.
2. Let $c \leftarrow F$ to make $A_5$ true.
3. (1) and (2) requires that $g \leftarrow T$ to make $A_4$ true.

$$A_4: \quad (\neg a \lor \neg c) \land m \supset g$$
Example (continued)

1. Let $m \leftarrow T$ so that B becomes false.
2. Let $c \leftarrow F$ to make $A_5$ true.
3. (1) and (2) requires that $g \leftarrow T$ to make $A_4$ true.
4. This requires that $p \leftarrow T$ to make $A_3$ true.

$A_3: \neg p \supset \neg m$
Example (continued)

1. Let \( m \leftarrow T \) so that \( B \) becomes false.
2. Let \( c \leftarrow F \) to make \( A_5 \) true.
3. (1) and (2) requires that \( g \leftarrow T \) to make \( A_4 \) true.
4. This requires that \( p \leftarrow T \) to make \( A_3 \) true.
5. For \( A_2 \) to be true, we need to set \( g \leftarrow F \). This contradicts (3).

\[
A_2: \quad a \land c \supset m \equiv p \land \neg g
\]
Example (continued)

1. Let $m \leftarrow T$ so that $B$ becomes false.
2. Let $c \leftarrow F$ to make $A_5$ true.
3. (1) and (2) requires that $g \leftarrow T$ to make $A_4$ true.
4. This requires that $p \leftarrow T$ to make $A_3$ true.
5. For $A_2$ to be true, we need to set $g \leftarrow F$. This contradicts (3).
6. Hence $A_2 \land A_3 \land A_4 \land A_5 \supset B$ is a tautology.
Example (continued)

That is to say, in views of the three rules/policies and the fact that the drug has failed the clinical test, the company should not market the drug unless the company is willing to violate at least one of the rules or contradict the fact that the drug failed the clinical test.
Example (continued)

Note that $A_2$, $A_3$, $A_4$, and $A_5$ are consistent in that it is possible to make them all true at the same time, e.g. by using the following assignment:

- $c \leftarrow F$
- $m \leftarrow F$
- $p \leftarrow T$
- $g \leftarrow F$
- $a \leftarrow T$
Example (continued)

Also note that $A_1 \wedge A_3 \wedge A_4 \wedge A_5 \supset B$ is not a tautology (prove this!). That is to say, if the first rule is interpreted in the way denoted by $A_1$, the company may decide to market or not to market the drug without violating any rule or contradicting any fact denoted by $A_1, A_3, A_4,$ and $A_5$. 
The First-Order Predicate Calculus

An extension of the propositional calculus that allows us to deal with sentences of the form:

\[ x > 0 \]

She is six feet tall.

Everyone is dressed in blue today.

These are called a sentential form.
Additional vocabulary

In addition to those in the propositional calculus, the first-order predicate calculus includes symbols

for individual constants (names of individuals): a, b, c, …
for individual variables (pronouns): x, y, z, …
for function letters (to denote functions): f, g, h, …
for predicate letters (to denote predicates): F, G, H, …
for quantifiers: universal quantifier (\(\forall x\)),
existential quantifier (\(\exists x\)).
The syntax

**Definition:** A *term* is defined as follows:

(1) Individual constants and individual variables are terms.

(2) If $f$ is an $n$-ary functional letter and $t_1, t_2, \ldots, t_n$ are terms then $f(t_1, t_2, \ldots, t_n)$ is a term.

(3) Those and only those obtained by (1) and (2) are terms.
The syntax (continued)

Definition: A string is an \textit{atomic formula} if it is either

(1) a propositional variable standing alone, or

(2) a string of the form $F(t_1, t_2, \ldots, t_n)$, where $F$ is an n-ary predicate letter and $t_1, t_2, \ldots, t_n$ are terms.
The syntax (continued)

**Definition:** a *well-formed formula (wff)* in the language of the first-order predicate calculus is defined as follows.

1. An atomic formula is a wff.
2. If $A$ is a wff and $x$ is an individual variable then $(\forall x)A$ and $(\exists x)A$ are wffs.
3. If $A$ and $B$ are wffs then $\neg A$, $(A) \land (B)$, $(A) \lor (B)$, $(A) \supset (B)$, and $(A) \equiv (B)$ are wffs.
4. Those and only those obtained by 1, 2, and 3 are wffs.
The quantifiers

The notation

\((\forall x)P\) is to be read as "for all x (in the domain) …" and

\((\exists x)P\) is to be read as "there exists an x (in the domain) such that …".
The scope of a quantifier

is the subexpression to which the quantifier is applied. The occurrence of an individual variable, say, x, is said to be *bound* if it is either an occurrence (\(\forall x\)), (\(\exists x\)), or within the scope of a quantifier (\(\forall x\)) or (\(\exists x\)). Any other occurrence of a variable is a *free* occurrence.

Example:

\[ P(x) \land (\exists x)(Q(x) \equiv (\forall y)R(y)) \]
More about *scope*

A variable may be within the scope of more than one quantifier.

In that case, an occurrence of a variable is bound by the innermost quantifier on that variable within whose scope that particular occurrence lies.
Interpretation

Definition: An interpretation of a wff consists of a non-empty domain $D$ and an assignment to each $n$-ary predicate letter of an $n$-ary predicate on $D$, to each $n$-ary function letter of an $n$-ary function on $D$, and to each individual constant of a fixed element of $D$. 
Satisfiability in a domain

**Definition:** A wff is *satisfiable* in a domain \( D \) if there exists an interpretation with domain \( D \) and assignments of elements of \( D \) to the free occurrences of individual variables in the formula such that the resulting proposition is true.
Validity in a domain

**Definition:** A wff is *valid* in a domain D if for every interpretation with domain D and assignment of elements of D to free occurrences of individual variables in the formula the resulting proposition is true.
Satisfiability and Validity

A wff is *satisfiable* if it is satisfiable in some domain.

A wff is *valid* if it is valid in all domains.
Example

Consider the wff \((\forall x)P(f(x, a), b)\). A possible interpretation of this wff would be

D: the set of all integers

\[ P(u, v): u > v \]

\[ f(y, z): y + z \]

\[ a: 1 \]

\[ b: 0 \]

That is, it becomes \((\forall x)_D(x + 1 > 0)\), and is false.
Another example

Example: Consider the wff $(\forall x)(\exists y)P(f(x, y), a)$. A possible interpretation of this wff would be:

D: the set of all integers
P(u, v): u is equal to v
f(x, y): x + y
a: 0

Thus it reads $(\forall x)(\exists y)(x + y = 0)$, and is true.
The order in which quantifiers occur

Observe that the order in which the quantifiers are given is important, and cannot be arbitrarily changed. For example,

\[(\forall x)(\exists y)(x + y = 0)\] is true, and

\[(\exists x)(\forall y)(x + y = 0)\] is false

in the previous interpretation.
Prenex normal form

**Definition:** A wff is said to be in the *prenex normal form* if it is of the form

\[(Q_1x_1)(Q_2x_2) \ldots (Q_nx_n)M,\]

where each \((Q_ix_i)\) is either \((\forall x_i)\) or \((\exists x_i)\), and \(M\) is a formula containing no quantifiers.

\[(Q_1x_1)(Q_2x_2) \ldots (Q_nx_n)\] is called the *prefix* and \(M\) the *matrix* of the formula.
Theorems useful for moving quantifiers

$$(\exists x)(\exists y)A \equiv (\exists y)(\exists x)A$$

$$(\forall x)(\forall y)A \equiv (\forall y)(\forall x)A$$

$$(\forall x)(A \supset B) \equiv ((\exists x)A \supset B)$$

$$(\exists x)(A \supset B) \equiv ((\forall x)A \supset B)$$

$$(\forall x)(A \supset B) \equiv (A \supset (\forall x)B)$$

$$(\exists x)(A \supset B) \equiv (A \supset (\exists x)B)$$

where $x$ does not occur free in $B$ in (3) and (4), and in $A$ in (5) and (6).
Theorems (continued)

To illustrate the necessity of the qualifier "where x does not occur free in B" for (3), let us consider the following interpretation where x occurs free in B(x, y):

D: the set of all positive integers
A(x, y): x divides y
B(x, y): x ≤ y
Theorems (continued)

With this interpretation (3) reads

$$(\forall x) ("x \text{ divides } y" \supset x \leq y)$$

$$\equiv (\exists x) (x \text{ divides } y) \supset x \leq y.$$  

Although the left-hand side of the "\equiv" is true, the truth value of the right-hand side depends on the assignment made to the free variable $x$, and thus the equivalence relation does not hold.
Theorems (continued)

Now if we interpret $B(x, y)$ to be $(\exists x)((y \div x)x=y)$, (3) reads

\[ (\forall x)("x \text{ divides } y" \supset (\exists x)((y \div x)x=y)) \]

\[ \equiv (\exists x)(x \text{ divides } y) \supset (\exists x)((y \div x)x=y). \]

The equivalence relation holds because $x$ does not occur free in $B$. 

©J. C. Huang 2003
Theorems (continued)

Note that the equivalence relation also holds if $x$ does not occur in $B$ at all. For example, if we interpret $B(x, y)$ to be "$y$ is not prime" then (3) reads

$$(\forall x)("x \text{ divides } y" \supset "y \text{ is not prime"})$$

$$\equiv (\exists x)(x \text{ divides } y) \supset "y \text{ is not prime"}.$$
More theorems (for $\land$ and $\lor$)

(1a) $\neg((\exists x)A(x)) \equiv (\forall x)(\neg A(x))$

(1b) $\neg((\forall x)A(x)) \equiv (\exists x)(\neg A(x))$

(2a) $(Qx)A(x) \lor B \equiv (Qx)(A(x) \lor B)$

(2b) $(Qx)A(x) \land B \equiv (Qx)(A(x) \land B)$

(3a) $(\exists x)A(x) \lor (\exists x)C(x) \equiv (\exists x)(A(x) \lor C(x))$

(3b) $(\forall x)A(x) \land (\forall x)C(x) \equiv (\forall x)(A(x) \land C(x))$

(4a) $(Q_1x)A(x) \lor (Q_2x)C(x) \equiv (Q_1x)(Q_2y)(A(x) \lor C(y))$

(4b) $(Q_3x)A(x) \land (Q_4x)C(x) \equiv (Q_3x)(Q_4y)(A(x) \land C(y))$

where $x$ does not occur in $B$ in (2), $y$ does not occur in $A(x)$ in (4), and $Q$, $Q_1$, $Q_2$, $Q_3$, and $Q_4$ are either $\exists$ or $\forall$. 
Example

\[(\forall x)P(x) \land (\exists x)Q(x) \lor \neg(\exists x)R(x)\]
\[(\forall x)P(x) \land (\exists x)Q(x) \lor (\forall x)(\neg R(x)) \quad \text{by (1a)}\]
\[(\forall x)(\exists y)(P(x) \land Q(y)) \lor (\forall x)(\neg R(x)) \quad \text{by (4b)}\]
\[(\forall x)(\exists y)(\forall z)(P(x) \land Q(y) \lor \neg R(z)) \quad \text{by (4a)}\]
Example application

Find an assignment that satisfies

\[ b - a > e \land b + 2a \geq 6 \land 2(b - a)/3 \leq e \]

which is the path condition of an execution path.
Inequalities expressed as equalities

- $a \neq b$ if and only if there exists an $x \neq 0$ such that $a + x = b$.
- $a < b$ if and only if there exists an $x > 0$ such that $a + x = b$.
- $a \leq b$ if and only if there exists an $x \geq 0$ such that $a + x = b$.
- $a > b$ if and only if there exists an $x < 0$ such that $a + x = b$.
- $a \geq b$ if and only if there exists an $x \leq 0$ such that $a + x = b$. 
Alternative expressions

In the language of the predicate calculus:

\[
\begin{align*}
\text{a = b} & \iff (\exists x)_{\neq 0}(x = b - a) \\
\text{a < b} & \iff (\exists x)_{> 0}(x = b - a) \\
\text{a \leq b} & \iff (\exists x)_{\geq 0}(x = b - a) \\
\text{a > b} & \iff (\exists x)_{> 0}(x = a - b) \\
\text{a \geq b} & \iff (\exists x)_{\geq 0}(x = a - b)
\end{align*}
\]
The path condition

can be expressed as

$$(\exists x)_{D_1}(x = b - a - e)$$

$$\land (\exists x)_{D_2}(x = b + 2a - 6)$$

$$\land (\exists x)_{D_2}(x = e - 2(b - a)/3)$$

where $D_1$ is the set of all real numbers $>0$

and $D_2$ is the set of all real numbers $\geq 0$
The path condition

in its prenex normal form is found to be

\((\exists x)_{D_1}(\exists y)_{D_2}(\exists z)_{D_2}(x = b - a - e \land y = b + 2a - 6 \land z = e - 2(b - a)/3)\)

which can be reduced to:

\((\exists x)_{D_1}(\exists y)_{D_2}(\exists z)_{D_2}(3x - y + 3z = 6 - 3a)\)
Test-case selection

The domain of this formula requires that the value of x, y, and z must satisfy $x > 0$, $y \geq 0$, and $z \geq 0$. Now if we let

\[
x \leftarrow 0.1, \quad y \leftarrow 0, \quad z \leftarrow 0.
\]

then a possible assignment would be

\[
a \leftarrow 1.9 \quad b \leftarrow 2.2 \quad e \leftarrow 0.2
\]

that satisfies the path condition

\[
b - a > e \land b + 2a \geq 6 \land 2(b - a)/3 \leq e
\]
Principle of Mathematical Induction

If 0 has a property P, and if any integer n is P then n+1 is also P, then every integer is P.

The principle is used in proving statements about integers or, derivatively, in proving statements about sets of objects of any kind which can be correlated with integers.
Steps involved in constructing a proof

The procedure is to prove that

(a) 0 is \( P \) (induction basis),

to assume that

(b) \( n \) is \( P \) (induction hypothesis),

to prove that

(c) \( n+1 \) is \( P \) (induction step)

using (a) and (b); and then to conclude that

(d) \( n \) is \( P \) for all \( n \).
Example

Suppose we wish to prove that

\[ \sum_{i=0}^{n} i = 0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \]
Example (continued)

To begin, we must state the property that we want to prove. This statement is called the induction proposition. In this case P is directly given by

\[
\begin{align*}
n & \text{ is } P \iff \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \\
& \text{where } i = 0
\end{align*}
\]
Example (continued)

(a) For the basis of the induction we have, for \( n = 0 \),
\[
0 = 0(0+1)/2,
\]
which is true.

(b) The induction hypothesis is that \( k \) is P for some arbitrarily choice of \( k \):
\[
\sum_{i=0}^{k} i = 0 + 1 + 2 + \ldots + k = k(k+1)/2
\]
Example (continued)

(c) For the induction step, proving $k + 1$ is $P$, we have

\[
\Sigma_{i=0}^{k+1} i = \Sigma_{i=0}^{k} i + (k+1) = \frac{k+1}{2} + (k+1) \\
= k^2 + 3k + 2/2 \\
= (k+1)(k+2)/2 \\
= \frac{(k+1)(k+2)}{2} \\
= \frac{(k+1)(k+3)-1}{2}
\]

(using the induction hypothesis)
Example (continued)

(d) Hence $k+1$ has the property $P$. 
Inductive (or recursive) definition

*Inductive definition* of a set or property $P$:
given a finite set $A$,

(a) the elements of $A$ are $P$ (basis clause)

(b) the elements of $B$, all of which are constructed from $A$, are $P$ (inductive clause)

(c) the elements constructed as in (a) and (b) are the only elements of $P$ (extremal clause)
Recursive definition of a set

say, D

1. Element $d_0$ is in D.
2. If d is in D and $P(d)$ then $f(d)$ is also in D.
3. Those and only those obtained by a finite applications of (1) and (2) are in D.
Example

Recursive definition of set D of even integers between 0 and 32, inclusive:

1. 0 is an element of D.
2. If \( d \in D \) and \( d \leq 30 \) then \( d+2 \) is also an element in D.
3. Those and only those obtained by a finite application of (1) and (2) are elements of D.
Proving Programs Correct

A common task in program verification is to show that, for a given program $S$, if a certain \textit{precondition} $Q$ is true before the execution of $S$ then a certain \textit{postcondition} $R$ is true after the execution, provided that $S$ terminates. This proposition is commonly denoted by $Q\{S\}R$ for short.
Proving Programs Correct (continued)

If we succeeded in showing that $Q\{S\}R$ is a theorem (i.e., always true), then to show that $S$ is partially correct, with respect to some input predicate $I$ and output predicate $Ø$, is to show that $I \supset Q$ and $R \supset Ø$. 

\[
\begin{align*}
I & \quad \uparrow \\
S & \quad \downarrow Q \\
& \quad \downarrow R \\
Ø & 
\end{align*}
\]
Two alternative approaches

Verification of correctness can be carried out in two ways:
Given $S$, $I$, and $\emptyset$ we may first let $R \equiv \emptyset$ and show that $Q\{S\}\emptyset$ for some predicate $Q$, and then show that $I \supset Q$.

Alternatively, we may let $Q \equiv I$ and show that $I\{S\}R$ for some predicate $R$, and then show that $R \supset \emptyset$. 
Bottom-up approach

In the first approach the basic problem is to find as weak as possible a condition $Q$ such that $Q\{S\}\emptyset$ and $I \supset Q$.

A possible solution is to use the method of predicate transformation to find the weakest precondition.
Top-down approach

In the second approach the problem is to find as strong as possible a condition \( R \) so that \( I\{S\}R \) and \( R \supset \emptyset \). This problem is fundamental to the method of \textit{inductive assertions}. 

\[ \text{Diagram:} \]

- \( I \) to \( Q \)
- \( Q \) to \( S \)
- \( S \) to \( \emptyset \)
Assumption about the language used

We assume that programs are written in a language consisting of the following statements:

(1) assignment statements: \( x := e; \)
(2) conditional statements: \( \text{if } B \text{ then } S \text{ else } S'; \)
(3) repetitive statements: \( \text{while } B \text{ do } S; \)
and a program is constructed by concatenating such statements.
INTDIV: an example program

INTDIV:  begin

    q := 0;
    r := x;

    while r ≥ y do
        begin
            r := r - y;
            q := q + 1
        end

end.

©J. C. Huang 2003  Chapter 1  86
Example

Suppose we wish to verify that program INTDIV is partially correct with respect to input predicate $I: x \geq 0 \land y > 0$ and output predicate $\emptyset: x = r + q \times y \land r < y \land r \geq 0$, i.e., to prove that

$$(x \geq 0 \land y > 0) \{\text{INTDIV}\} (x = r + q \times y \land r < y \land r \geq 0)$$

is a theorem.
The Predicate Transformation Method: Bottom-Up Approach

Recall that in the first approach, given S, I, and Ø, the basic problem is to find as weak as possible a condition Q such that Q{S}Ø, and then determine if $I \supset Q$.
Weakest precondition

Let $S$ be a programming construct and $R$ be a predicate or condition (henceforth we shall use the terms predicate, condition, and logical expression interchangeably). Then $\text{wp}(S, R)$ denotes the \textit{weakest precondition} for the initial state such that an execution of $S$ will properly terminate, leaving it in a final state satisfying the condition $R$. 
wp(S, R) is called a *predicate transformer* and has the following properties:

1. For any S, \( wp(S, F) \equiv F \)
2. For any program S and any predicates S, Q, and R, if \( Q \supset R \) then \( wp(S, Q) \supset wp(S, R) \).
3. For any programming construct S and any predicates Q and R, \( (wp(S, Q) \land wp(S, R)) \equiv wp(S, Q \land R) \).
4. For any deterministic programming construct S and any predicates Q and R, \( (wp(S, Q) \lor wp(S, R)) \equiv wp(S, Q \lor R) \).
skip and abort

We shall define two special statements: skip and abort.

The statement skip is the same as the null statement in a high-level language, or the "no-op" instruction in an assembly language. Its meaning can be given as $\text{wp}(\text{skip}, R) \equiv R$ for any predicate $R$.

The statement abort, when executed, will not lead to a final state. Its meaning is defined as $\text{wp}(\text{abort}, R) \equiv F$ for any predicate $R$. 
wp(x:=E, R) ≡ R_{E→x}

<table>
<thead>
<tr>
<th>R</th>
<th>x := E</th>
<th>R_{E→x}</th>
<th>simplified to</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0</td>
<td>x := 0</td>
<td>0 = 0</td>
<td>T</td>
</tr>
<tr>
<td>a &gt; 1</td>
<td>x := 10</td>
<td>a &gt; 1</td>
<td>a &gt; 1</td>
</tr>
<tr>
<td>x &lt; 10</td>
<td>x := x + 1</td>
<td>x + 1 &lt; 10</td>
<td>x &lt; 9</td>
</tr>
<tr>
<td>x ≠ y</td>
<td>x := x - y</td>
<td>x - y ≠ y</td>
<td>x ≠ 2y</td>
</tr>
</tbody>
</table>
For a sequence of two programming constructs $S_1$ and $S_2$,

$$wp(S_1; S_2, R) \equiv wp(S_1, wp(S_2, R)).$$
\[ \text{wp}(\text{if } B \text{ then } S_1 \text{ else } S_2, R) \]

\[ \text{wp}(\text{if } B \text{ then } S_1 \text{ else } S_2, R) \equiv B \land \text{wp}(S_1, R) \lor \neg B \land \text{wp}(S_2, R). \]
\[ wp(\text{while } B \text{ do } S, R) \]

\[ wp(\text{while } B \text{ do } S, R) \equiv (\exists j)_{j \geq 0}(A_j(R)), \]

where

\[ A_0(R) \equiv \neg B \land R \text{ and } \]
\[ A_{j+1}(R) \equiv B \land wp(S, A_j(R)) \text{ for all } j \geq 0. \]
Example: proving INTDIV correct

We first compute

\[
\text{wp(while } r \geq y \text{ do begin } r := r - y; \ q := q + 1 \ \text{end,}
\]

\[
x = r + q \times y \land r < y \land r \geq 0
\]

where

\[
B \equiv r \geq y
\]

\[
R \equiv x = r + q \times y \land r < y \land r \geq 0
\]

\[
S: \ r := r - y; \ q := q + 1;
\]
Example (continued)

\[ A_0(R) \equiv \neg B \land R \]
\[ \equiv r < y \land x = r + q \times y \land r < y \land r \geq 0 \]
\[ \equiv x = r + q \times y \land r < y \land r \geq 0 \]
\[ A_1(R) \equiv B \land wp(S, A_0(R)) \]
\[ \equiv r \geq y \land wp(r := r - y; q := q + 1, x = r + q \times y \land r < y \land r \geq 0) \]
\[ \equiv r \geq y \land x = r - y + (q + 1) \times y \land r - y < y \land r - y \geq 0 \]
\[ \equiv x = r + q \times y \land r < 2 \times y \land r \geq y \]
Example (continued)

$A_2(R) \equiv B \land wp(S, A_1(R))$

$\equiv x = r + q \times y \land r < 3 \times y \land r \geq 2 \times y$

$A_3(R) \equiv B \land wp(S, A_2(R))$

$\equiv x = r + q \times y \land r < 4 \times y \land r \geq 3 \times y$
Example (continued)

From these we may guess that

\[ A_j(R) \equiv B \land \text{wp}(S, A_{j-1}(R)) \]

\[ \equiv x = r + q \times y \land r < (j+1) \times y \land r \geq j \times y \]

and we have to prove that our guess is correct by mathematical induction.
Example (continued)

Assume that \( A_j(R) \) is as given above, then

\[
A_0(R) \equiv x = r + q \times y \land r < (0+1) \times y \land r \geq 0 \times y
\]

\[
\equiv x = r + q \times y \land r < y \land r \geq 0
\]

\[
A_{j+1}(R) \equiv B \land wp(S, A_j(R))
\]

\[
\equiv r \geq y \land wp(r := r - y; q := q + 1, x = r + q \times y \land r < (j+1) \times y \land r \geq j \times y)
\]

\[
\equiv r \geq y \land x = r - y + (q + 1) \times y \land r - y < (j+1) \times y \land r - y \geq j \times y
\]

\[
\equiv x = r + q \times y \land r < ((j+1) + 1) \times y \land r \geq (j+1) \times y
\]
Example (continued)

These two instances of $A_j(R)$ show that if $A_j(R)$ is correct then $A_{j+1}(R)$ is also correct as given above.
Example (continued)

Hence

\[ wp(\textbf{while } r \geq y \textbf{ do begin } r := r - y; \ q := q + 1 \ \textbf{ end,} \]
\[ x = r + q \times y \land r < y \land r \geq 0) \]
\[ \equiv (\exists j)_{j \geq 0}(A_j(R)) \]
\[ \equiv (\exists j)_{j \geq 0}(x = r + q \times y \land r < (j+1) \times y \land r \geq j \times y) \]
Example (continued)

\[ \text{wp}(q:=0; r:=x, (\exists j)_{j \geq 0}(x=r+q \times y \land r<(j+1) \times y \land r \geq j \times y)) \equiv (\exists j)_{j \geq 0}(x < (j+1) \times y \land x \geq j \times y) \]

which is implied by \( x \geq 0 \land y > 0 \), and hence the proof that the following is a theorem:

\((x \geq 0 \land y > 0) \{\text{INTDIV}\}(x=r+q \times y \land r < y \land r \geq 0)\).
Partial correctness and strong verification

Recall that $Q\{S\}R$ is a shorthand notation for the proposition: "if $Q$ is true before the execution of $S$ then $R$ is true after the execution, provided that $S$ terminates". Termination of the program has to be proved separately.

If $Q \equiv \text{wp}(S, R)$, however, termination of the program is guaranteed. In that case, we can write $Q[S]R$ instead, which is a shorthand notation for the proposition: "if $Q$ is true before the execution of $S$ then $R$ is true after the execution of $S$, and the execution will terminate".
The Inductive Assertion Method: Top-Down Approach

In the top-down approach, given a program S and a predicates Q, the basic problem is to find as strong as possible a condition R such that $Q(S) \rightarrow R$. 

\[ \text{Q} \quad \text{S} \quad \text{R} \]
Assignment statement

If $S$ is an assignment statement of the form $x := E$, where $x$ is a variable and $E$ is an expression, we have

$$Q\{x := E\}(Q' \land x = E')_{x' \rightarrow E^{-1}}$$

where $Q'$ and $E'$ are obtained from $Q$ and $E$, respectively, by replacing every occurrence of $x$ with $x'$, and then replace every occurrence of $x'$ with $E^{-1}$, such that $x = E' \equiv x' = E^{-1}$.
Given Q and x := E, construct \((Q' \land x = E')_{x' \rightarrow E^{-1}}\) as follows.

1. Write \(Q \land x = E\).
2. Replace every occurrence of x in Q and E with x' to yield \(Q' \land x = E'\).
3. If x' occurs in E' then construct \(x' = E^{-1}\) from \(x = E'\) such that \(x = E' \equiv x' = E^{-1}\), else \(E^{-1}\) does not exist.
4. If \(E^{-1}\) exists then replace every occurrence of x' in \(Q' \land x = E'\) with \(E^{-1}\). Otherwise, replace every atomic predicate in \(Q' \land x = E'\) having at least one occurrence of x' with T (the constant predicate TRUE).
Example

\[ Q \quad x := E \quad (Q' \land x = E')_{x' \to E^{-1}} \quad \text{simplified to} \]

- \( x = 0 \quad x := 10 \quad T \land x = 10 \quad x = 10 \)
- \( a > 1 \quad x := 1 \quad a > 1 \land x = 1 \quad a > 1 \land x = 1 \)
- \( x < 10 \quad x := x + 1 \quad x - 1 < 10 \quad x < 11 \)
- \( x \neq y \quad x := x - y \quad x + y \neq y \quad x \neq 0 \)
A notational convention

As explained earlier, it is convenient to use \( |-P \) to denote the fact that \( P \) is a theorem (i.e., always true).

A verification rule may be stated in the form "if \( |-X \) then \( |-Y \)," which says that if proposition \( X \) has been proved as a theorem then \( Y \) also is thereby proved as a theorem.
An important fact

Note that \( Q[S]R \supset Q\{S\}R \), but not the other way around.

Can you prove that \( Q[S]R \nsubseteq Q\{S\}R \)?
Rule 1:

For an assignment statement of the form $x := E$

$$|-Q\{x := E\}(Q' \land x = E')_{x' \rightarrow E^{-1}}$$
Rule 2:

For a conditional statement of the form

\[ \text{if } B \text{ then } S_1 \text{ else } S_2 \]

If \( |-Q \land B \{ S_1 \} R_1 \) and \( |-Q \land \neg B \{ S_2 \} R_2 \)

then \( |-Q \{ \text{if } B \text{ then } S_1 \text{ else } S_2 \} R_1 \lor R_2 \).
Rule 3

For a loop construct of the form \textbf{while} B \textbf{do} S

If $|-Q \supset R$ and $|-(R\land B)\{S\}R$

then $|-Q\{\textbf{while} B \textbf{ do} S\}(\neg B \land R)$.

This rule is commonly known as the \textit{invariant-relation theorem}, and any predicate R satisfying the premise is called a \textit{loop invariant} of the loop construct \textbf{while} B \textbf{ do} S.
The top-down strategy

Thus the partial correctness of program S with respect to input condition I and output condition Ø can be proved by showing that $I \{S\} Q$ and $Q \supseteq \emptyset$. 

\[ 
\begin{align*} 
I & \downarrow \\
S & \downarrow Q \\
\emptyset & 
\end{align*} 
\]
The proof can be constructed in smaller steps

if $S$ is a long sequence of statements. Specifically, if $S$ is $S_1; S_2; \ldots; S_n$ then $I\{S_1; S_2; \ldots; S_n\} \varnothing$ can be proved by showing that $I\{S_1\} P_1$, $P_1\{S_2\} P_2$, $\ldots$, and $P_{n-1}\{S_n\} \varnothing$ for some predicates $P_1$, $P_2$, $\ldots$, and $P_{n-1}$. $P_i$s are called inductive assertions, and this method of proving program correctness is called the inductive assertion method.
Proof requires guesswork

Required inductive assertions for constructing a proof often have to be found by guesswork, based on one's understanding of the program in question, especially if a loop construct is involved. No algorithm for this purpose exists, although some heuristics have been developed to aid the search.
Proving the correctness of INTDIV

I: $x \geq 0 \land y > 0$

begin
    q := 0;
    r := x;
    while $r \geq y$ do
        begin $r := r - y$; $q := q + 1$ end
end.

$\emptyset$: $x = r + q \times y \land r \geq 0 \land r < y$
Proving INTDIV (continued)

I: \( x \geq 0 \land y > 0 \)

begin
  \( q := 0; \)
  \( x \geq 0 \land y > 0 \land q = 0 \) \hspace{1cm} (by Rule 1)
  \( r := x; \)
  while \( r \geq y \) do
    begin \( r := r - y; \) q := q + 1 end
end.

\( \emptyset: x = r + q \times y \land r \geq 0 \land r < y \)
Proving INTDIV (continued)

I: x ≥ 0 ∧ y > 0

begin
    q := 0;
    x ≥ 0 ∧ y > 0 ∧ q = 0
    r := x;
    x ≥ 0 ∧ y > 0 ∧ q = 0 ∧ r = x (by Rule 1)
while r ≥ y do
    begin r := r - y; q := q + 1 end
end.

∅: x = r + q × y ∧ r ≥ 0 ∧ r < y
Proving \textsc{IntDiv} (continued)

I: $x \geq 0 \land y > 0$

begin

\begin{align*}
q & := 0; \\
r & := x; \\
x & \geq 0 \land y > 0 \land q = 0 \land r = x
\end{align*}

\textbf{while} $r \geq y$ \textbf{do}

\begin{align*}
\text{begin} & \ r := r - y; \ q := q + 1 \ \text{end} \\
x & = r + q \times y \land r \geq 0 \land r < y
\end{align*}

end.

\emptyset: x = r + q \times y \land r \geq 0 \land r < y
Proving INTDIV (continued)

Obviously

\[ x = r + q \times y \land r \geq 0 \land r < y \]

implies (in fact it is identical to)

∅

and hence the proof.
Comment on the above method

There are many variations to the inductive-assertion method. The above version is designed, as an integral part of this section, to show that a correctness proof can be constructed in a top-down manner. As such, we assume that a program is composed of a concatenation of statements, and an inductive assertion is to be inserted between such statements only.
Comment (continued)

The problem is that most programs contain nested loops and compound statements, which may render applications of Rules 2 and 3 hopelessly complicated.

The complication induced by nested loops and compound statements can be eliminated by representing the program as a flowchart.
A variation of the inductive assertion method

In this method, the program is represented as a flowchart, and appropriate assertions are placed on various points in the control flow. These assertions "cut" the flowchart into a set of paths. A path between assertions Q and R is formed by a single sequence of statements that will be executed if the control flow traverses from Q to R in an execution, and contains no other assertions. It is possible that Q and R are the same.
Basic path 1

\[
\begin{align*}
& Q \\
\xrightarrow{\text{x := E}} \\
& R
\end{align*}
\]

Associated lemma: \((Q' \land x = E')_{x' \rightarrow E^{-1}} \supset R\)
Basic path 2

Associated lemma: $Q \land B \supset R$
Basic path 3

Associated lemma: $Q \land \neg B \supset R$
The proof

In this method, we shall let the input predicate be the starting assertion at the program entry, and let the output predicate be the ending assertion at the program exit. To prove the correctness of the program is to show that every lemma associated with a basic path is a theorem.
The proof (continued)

If we succeeded in doing that, then due to transitivity of the implication relation, it implies that, if the input predicate is true at the program entry, the output predicate will be true also if and when the control reaches the exit (i.e., if the execution terminates). Therefore it constitutes a proof of the partial correctness of the program.
The proof (continued)

In practice, we work with composite paths instead of simple paths to reduce the number of lemma needs to be proved. A composite path is a path formed by a concatenation of more than one simple path. The lemma associated with a composite path can be constructed by observing that the effect produced by a composite path is the conjunction of that produced by its constituent simple paths.
At least one assertion should be inserted into each loop so that any path is of finite length.
Flowchart of program INTDIV

Entry

- - A: \( x \geq 0 \) and \( y > 0 \)

\[
\begin{align*}
q &:= 0 \\
r &:= x
\end{align*}
\]

- - B: \( x = r + q \times y \) and \( r \geq 0 \) and \( y > 0 \)

\[
\begin{align*}
r := r - y \\
q := q + 1
\end{align*}
\]

- - T: \( r \geq y \)

- - F: \( r < y \) and \( r \geq 0 \)

Exit

- - C: \( x = r + q \times y \) and \( r < y \) and \( r \geq 0 \)
Example (continued)

Three assertions are used: A is the input predicate, C is the output predicate, and B is the assertion used to cut the loop. Assertion B cannot be simply $q = 0$ and $r = x$ because B is not merely the ending point of path AB, it is also the beginning and ending points of path BB. Therefore, we have to guess the assertion at that point that will lead us to a successful proof. In this case, it is not difficult to guess because the output predicate provides a strong hint as to what we need at that point.
Example (continued)

There are three paths: AB, BB, and BC.

Path AB: \( x \geq 0 \land y > 0 \land q = 0 \land r = x \Rightarrow x = r + q \cdot y \land r \geq 0 \land y > 0 \)

Path BB: \( x = r + q \cdot y \land r \geq 0 \land y > 0 \land r \geq y \land r' = r - y \land q' = q + 1 \Rightarrow x = r' + q' \cdot y \land r' \geq 0 \land y > 0 \)

Path BC: \( x = r + q \cdot y \land r \geq 0 \land y > 0 \land \neg(r \geq y) \Rightarrow x = r + q \cdot y \land r < y \land r \geq 0 \)
Example (continued)

These three lemmas can be readily proved as follows.

Lemma for Path AB: Substitute 0 for q and r for x in the consequence.

Lemma for Path BB: Eliminate q' and r' and simplify.

Lemma for Path BC: Use the fact that \( \neg(r \geq y) \) is \( r < y \), and simplify.
Common error

A common error made in constructing a correctness proof is that the guessed assertion is either stronger or weaker than what is needed. Let $P$ be the correct inductive assertion to use in proving $I\{S_1;S_2\}O$, that is, $I\{S_1\}P$ and $P\{S_2\}O$ are both a theorem. If the guessed assertion is too weak, say, $P \lor \Delta$, where $\Delta$ is some extraneous predicate, $I\{S_1\}(P \lor \Delta)$ is still a theorem, but $(P \lor \Delta){S_2}O$ may not be. On the other hand, if the guessed assertion is too strong, say, $P \land \Delta$, $(P \land \Delta){S_2}O$ is still a theorem but $I\{S_1\}(P \land \Delta)$ may not be.
Common error (continued)

Consequently, if one failed to construct a proof by using the inductive assertion method, it does not necessarily mean that the program is incorrect. Failure of a proof could result either from an incorrect program or incorrect choices of inductive assertions. In comparison, the bottom-up (predicate transformation) method does not have this disadvantage.
Directed Graphs and Path Description

When we study the logical structure of a program, we need to be able to speak of a path structure precisely and concisely. This can be accomplished by making use of the language of regular expressions.
Three basic connection schemes

A set of paths between any two nodes in a (directed) graph can be described in terms of symbols associated with the constituent edges as follows.
Extension

The same rules also apply to the cases where $a$ and $b$ are expressions describing complex path structures.

Hence a set of paths can be described by an expression composed of edge symbols and three connectives: concatenation, disjunction (+), and looping (*).
Example

The set of paths between nodes 1 and 4 in the graph shown above can be described by the regular expression

$$a \ (e + b \ c \ * \ d) \ .$$
Loop

If \( p \) describes a path, then \( p^* \) describes a loop formed by \( p \) and hence a set of paths obtained by iterating the loop for any number of times. Formally, \( p^* = \lambda + p + pp + ppp + \ldots \). Here \( \lambda \) is a special symbol denoting the identity under concatenation (i.e., \( x\lambda = \lambda x = x \) for any \( x \)) and is to be interpreted as a path of length zero (obtained by iterating the loop zero times).
Matrix representation of a graph

Let G be a directed graph with n nodes. G can be represented by an $n \times n$ matrix as follows. First, the nodes in G are to be order in some way. Then we form an $n \times n$ matrix $[G] = [g_{ij}]$, where $g_{ij}$ (the element on the i-th row and j-th column) is a regular expression denoting the set of all paths of length 1 (i.e., the paths formed by a single edge) leading from the i-th node to the j-th node.
Example

The graph given previously can be represented as

\[
\begin{array}{cccc}
\emptyset & a & \emptyset & \emptyset \\
\emptyset & \emptyset & b & e \\
\emptyset & \emptyset & c & d \\
\emptyset & \emptyset & \emptyset & \emptyset \\
\end{array}
\]

\(\emptyset\): empty set.
Matrix operation

Let \([X], [Y], [Z],\) and \([W]\) be \(n \times n\) matrices. We define

\[
[X] + [Y] = [Z] = [z_{ij}]
\]

where \(z_{ij} = x_{ij} + y_{ij}\)
Matrix operation (continued)

\[ [X][Y] = [W] = [w_{ij}] \]

where \( w_{ij} = \sum_{k=1}^{n} k y_{kj} \), and

\[ [X]^* = [X]^0 + [X]^1 + [X]^2 + [X]^3 + ..., \]

where \([X]^0\) is defined to be an \( n \times n \) matrix in which every element on the main diagonal is \( \lambda \), and all other elements are identically \( \emptyset \).
General form of a matrix

\[ [G] = \begin{bmatrix}
  g_{11} & \cdots & g_{1k} & \cdots & g_{1n} \\
  \vdots & & \vdots & & \vdots \\
  g_{k1} & \cdots & g_{kk} & \cdots & g_{kn} \\
  \vdots & & \vdots & & \vdots \\
  g_{n1} & \cdots & g_{nk} & \cdots & g_{nn}
\end{bmatrix} \]
Elimination of the $k$-th column and row

\[
\begin{bmatrix}
  g_{11} & \cdots & g_{1(k-1)} & g_{1(k+1)} & \cdots & g_{1n} \\
  \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
  g_{(k-1)1} & \cdots & g_{(k-1)(k-1)} & g_{(k-1)(k+1)} & \cdots & g_{(k-1)n} \\
  g_{(k+1)1} & \cdots & g_{(k+1)(k-1)} & g_{(k+1)(k+1)} & \cdots & g_{(k+1)n} \\
  \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
  g_{n1} & \cdots & g_{n(k-1)} & g_{n(k+1)} & g_{nn}
\end{bmatrix}
\]
Elimination of node k (continued)

\[
\begin{bmatrix}
  g_{1k} \\
  \vdots \\
  g_{(k-1)k} \\
  g_{(k+1)k} \\
  \vdots \\
  g_{nk}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g_{kk} \\
  \vdots \\
  g_{kk}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  g_{k1} & \cdots & g_{k(k-1)} & g_{k(k+1)} & \cdots & g_{kn}
\end{bmatrix}^*
\]
Elimination of node k (continued)

By repeatedly eliminating unessential nodes, we will be left with a $2 \times 2$ matrix shown below:

$$[B'] = \begin{bmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{bmatrix}$$
Elimination of node $k$ (continued)

Then $p_{ij}$ can be constructed from the elements in $[B']$ as follows:

$$p_{ij} = (b_{ii} + b_{ij} b_{jj} * b_{ji}) * b_{ij} (b_{jj} + b_{ji} b_{ii} * b_{ij}) *$$

If $b_{ii} = b_{ji} = b_{jj} = \emptyset$, which is almost always the case in many applications, then we have

$$p_{ij} = b_{ij}$$
Example

Consider again the graph represented by

\[
\begin{pmatrix}
0 & a & 0 & 0 \\
0 & 0 & b & e \\
0 & 0 & c & d \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
Example (continued)

To find the set of all paths between nodes 1 and 4, we shall first eliminate node 2, i.e., column 2 and row 2 to yield

\[
\begin{bmatrix}
\emptyset & \emptyset & \emptyset \\
\emptyset & c & d \\
\emptyset & \emptyset & \emptyset
\end{bmatrix}
\]

\[
\begin{bmatrix}
a \\
\emptyset \\
\emptyset
\end{bmatrix}
\times
\begin{bmatrix}
\emptyset \\
\emptyset
\end{bmatrix}^\ast
\begin{bmatrix}
\emptyset & b & e
\end{bmatrix}
\]
Example (continued)

\[
\begin{pmatrix}
\emptyset & \emptyset & \emptyset \\
\emptyset & c & d \\
\emptyset & \emptyset & \emptyset \\
\end{pmatrix}
+ \begin{pmatrix}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
\emptyset & ab & ae \\
\emptyset & c & d \\
\emptyset & \emptyset & \emptyset \\
\end{pmatrix}
= \begin{pmatrix}
\emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset \\
\end{pmatrix}
\]
Example (continued)

Now column 2 and row 2 corresponds to the node labeled by integer 3. It can be similarly eliminated to yield the following $2 \times 2$ matrix.
Example (continued)

\[
\begin{bmatrix}
0 & ae \\
0 & 0 \\
0 & 0
\end{bmatrix} + \begin{bmatrix}
ab \\
0 \\
0
\end{bmatrix} [c]^{*} \begin{bmatrix}
0 \\
0 \\
d
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & ae \\
0 & 0 \\
0 & 0
\end{bmatrix} + \begin{bmatrix}
0 & abc^{*}d \\
0 & 0 \\
0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & ae + abc^{*}d \\
0 & 0 \\
0 & 0
\end{bmatrix}
\]
Example (continued)

Hence, the set of paths leading from node 1 to node 4 is described by

\[ ae + abc \cdot d. \]
Remark

Generally speaking, the nodes can be eliminated in any order.

By eliminating the nodes in different order, the method may produce different regular expressions (with different complexities) as the result. But they all represent the same set of paths.
A Formal Basis for Program Analysis

In many problem areas, such as proving program-correctness, symbolic execution, and program testing, one often needs to deal with portions of a program associated with certain execution paths.

Although conceptually it is useful to treat each of these components as a subprogram, there is no programming language or notational convention that allows us to do so.
State constraints

Introduced in the following section is a new programming construct, called a *state constraint*, that can be used to construct from a given program a subprogram with some of its execution paths. Such a subprogram can be systematically manipulated and simplified.
The Concept of a State Constraint

Consider a restrictive clause of the form:

*The program state at this point must satisfy predicate $C$, or else the program becomes undefined.*

By program state here we mean the aggregate of values assumed by all variables involved. Since this clause constrains the states assumable by the program, it is called a *state constraint*, or a *constraint* for short, and is denoted by $\land C$. 
A constrained program

State constraints are designed to be inserted into a program to create another program. For instance, given a program of the form

Program 1.6.1.1: \( S_1; S_2 \)

a new program can be created as shown below:

Program 1.6.1.2: \( S_1; \text{/}\text{\}/C; S_2 \)
Subprogram

Program $S_1; \backslash C; S_2$ is said to be created from program $S_1; S_2$ by constraining the program states to $C$ prior to execution of $S_2$. Intuitively, $S_1; \backslash C; S_2$ is a subprogram of $S_1; S_2$ because its definition is that of program $S_1; S_2$ restricted to $C$. Within that restriction, $S_1; \backslash C; S_2$ performs the same computation as $S_1; S_2$. 
Formal definition

A state constraint is a semantic modifier. The meaning of a program modified by a state constraint can be formally defined in terms of the weakest precondition as follows. Let S be a programming construct and C be a predicate, then for any postcondition R,

Axiom 1.6.1.3: $\text{wp}(\land \neg C; S, R) \equiv C \land \text{wp}(S, R)$. 
Equivalence relation

**Definition 1.6.1.4:** Program $S_1$ is said to be *equivalent* to $S_2$ if $\text{wp}(S_1, R) \equiv \text{wp}(S_2, R)$ for any postcondition $R$. This relation is denoted by

$$S_1 \iff S_2.$$
Subprogram relation

Definition 1.6.1.5: Program $S_2$ is said to be a subprogram of program $S_1$ if $\text{wp}(S_2, R) \supset \text{wp}(S_1, R)$ for any postcondition $R$. This relation is denoted by

$S_1 \Rightarrow S_2$. 
Example

Consider programs $S_1; S_2$ and $S_1; \setminus C; S_2$ again. Since $wp(S_1; \setminus C; S_2, R) \equiv wp(S_1, wp(\setminus C; S_2, R))$
$\equiv wp(S_1, C \land wp(S_2, R)) \equiv wp(S_1, C) \land wp(S_1, wp(S_2, R)) \equiv wp(S_1, C) \land wp(S_1; S_2, R)$, it follows that
$wp(S_1; \setminus C; S_2, R) \supset wp(S_1; S_2, R)$.

Thus, by Def. 1.6.1.5, Program $S_1; \setminus C; S_2$ is a subprogram of Program $S_1; S_2$. 
A trivial relation

Note that if $C \equiv T$, i.e., if $C$ is always true, then $\wp(\land \neg T; S, R) \equiv T \land \wp(S, R) \equiv \wp(S, R)$, and therefore, by Definition 1.6.1.4,

**Corollary 1.6.1.6:** $\land \neg T; S \Leftrightarrow S$
Another trivial relation

That is to say, a state constraint will have no effect on a program if it is always true. On the other hand, if $C \equiv F$, i.e., if $C$ is always false, then $\wp(\not\land F; S, R) \equiv F \land \wp(S, R) \equiv F \equiv \wp(\not\land F; S', R)$ for any $S, S', \text{ and } R$, and therefore

**Corollary 1.6.1.7:** $\not\land F; S \Leftrightarrow \not\land F; S'$. 
Relations for subprogram simplification

There are three categories of equivalence relations that can be inserted into a program to reduce it to a simpler subprogram.

The first is designed to constrain a program so that the resulting subprogram will have a simpler logical structure.
Relations for subprogram simplification

The second can be used to manipulate and simplify the state constraints contained in a program.
Relations for subprogram simplification

The third allows us to simplify long sequences of assignment statements, which often result from a repeated application of the first two.
A Corollary

\( \land B; \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \Leftrightarrow \land B; S_1 \)

[Proof] By Axiom 1.6.1.3,

\[
wp(\land B; \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, R) \\
\equiv B \land wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, R) \\
\equiv B \land (B \land wp(S_1, R) \lor \neg B \land wp(S_2, R)) \\
\equiv B \land wp(S_1, R) \\
\equiv wp(\land B; S_1, R). \quad Q.E.D.
\]
More rules of the first category

Corollary 1.6.2.2:

\[ \land \neg B; \text{ if } B \text{ then } S_1 \text{ else } S_2 \Leftrightarrow \land \neg B; S_2 \]

Corollary 1.6.2.3:

\[ \land B_i; \text{ if } B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_2 \text{ else if } B_i \text{ then } S_i \text{ ... else if } B_n \text{ then } S_n \Leftrightarrow \land \neg B_1 \land \neg B_2 \land \ldots \land B_i; S_i \]
More rules of the first category

Corollary 1.6.2.4:
\( \text{while } B \text{ do } S \iff \text{while } B \text{ do } S \)

Corollary 1.6.2.5:
\( \text{while } B \text{ do } S \iff \text{while } \neg B \)

Corollary 1.6.2.6:
\( \text{while } B \text{ do } S \iff \text{repeat } S \text{ until } \neg B \)
Canonical subprogram

Definition 1.6.2.9: A subprogram is said to be \textit{canonical} if it does not contain any control statement.
Rule of the second category

Corollary 1.6.2.10: \( \land C_1; \land C_2; S \iff \land C_1 \land C_2; S. \)
Rule of the second category

Theorem 1.6.2.12: \( S;\bigwedge R \Leftrightarrow \bigwedge Q;S \) if \( Q \equiv \text{wp}(S, R) \).

This relation can be used repeatedly to move a constraint upstream, i.e., to constrain the program equivalently at a different point upstream.
Moving a constraint downstream

On the other hand, given a program of the form \( \land Q; S \), we may want to move the constraint downstream. In that case, what we need to do is to find a predicate \( R \) such that \( \text{wp}(S, R) \equiv Q \).

This can be done by letting \( R \equiv \text{wp}(S^{-1}, Q) \), where \( S^{-1} \) is a sequence of statements to be derived from \( S \) and \( Q \). By replacing \( R \) in \( \text{wp}(S, R) \equiv Q \), we obtain \( \text{wp}(S; S^{-1}, Q) \equiv Q \).
Computation of $S^{-1}$

If $S;S^{-1}$ is a sequence of assignment statements, $wp(S;S^{-1}, Q) \equiv Q$. is true as long as an execution of $S;S^{-1}$ will not alter the value of any variable that occurs in $Q$. Thus a suitable $S^{-1}$ may be found by noting that, if $S$ contains a statement that changes the value of a variable in $Q$, $S^{-1}$ should contain a statement that restores the old value of that variable.
Computation of $S^{-1}$ (continued)

$S^{-1}$ may or may not exist.

That implies that a constraint may not be moved downstream beyond a certain point.
More rule of the second category

Theorem 1.6.2.12:  (a) $S;\land R \iff \land \wp(S, R);S$, or
(b) $\land Q;S \iff S;\land \wp(S^{-1}; Q)$.  

Tautological constraints

Definition 1.6.2.14: A state constraint is said to be *tautological* if it can be eliminated without changing the function implemented by the program. To be more precise, the constraint \( C \) in the program \( S_1;C; S_2 \) is tautological if and only if \( S_1;C; S_2 \leftrightarrow S_1;S_2 \).
Rules of the third category

Corollary 1.6.2.15: \( x := E_1; x := E_2 \iff x := (E_2)_{E_1 \rightarrow x} \)

Corollary 1.6.2.16: If \( x_2 \) does not occur in \( E_1 \) then
\[
x_1 := E_1; x_2 := E_2 \iff x_2 := (E_2)_{E_1 \rightarrow x_1}; x_1 := E_1.
\]

Corollary 1.6.2.18: If \( x_1 := E_1; x_2 := E_2 \) is such that, by interchanging these two statements, \( x_1 := E_1 \) becomes redundant, then
\[
x_1 := E_1; x_2 := E_2 \iff x_2 := (E_2)_{E_1 \rightarrow x_1}.
\]
Pathwise Decomposition

As mentioned before, by inserting a constraint into a program, we shrink the domain for which it is defined. To reverse this process, we need to be able to speak of, and make use of, a set of subprograms. To this end, we shall now introduce a new programming construct called a program set. The meaning of a program set, or a set of programs, is identical to the conventional notion of a set of other objects.
Definition of a program set

As usual, a set of n programs is denoted by \( \{P_1, P_2, \ldots, P_n\} \). When used as a programming construct, it describes the computation prescribed by its elements. Formally, the semantics of such a set is defined as

\[
\text{Axiom: 1.6.3.1: } \text{wp}(\{P_1, P_2, \ldots, P_n\}, R) \\
\equiv \text{wp}(P_1, R) \lor \text{wp}(P_2, R) \lor \ldots \lor \text{wp}(P_n, R).
\]
Properties of a program set

**Corollary 1.6.3.2:** The ordering of elements in a program set is immaterial, i.e.,

\[ \{P_1, P_2\} \iff \{P_2, P_1\} \]

Furthermore, since every proposition is an idempotent under the operation of disjunction,

**Corollary 1.6.3.3:** \( P \iff \{P\} \iff \{P, P\} \)

for any program \( P \).
Properties of a program set (continued)

Corollary 1.6.3.4: For any programs $P$, $P_1$, and $P_2$,

(a) $P;\{P_1, P_2\} \iff \{P;P_1, P;P_2\}$

(b) $\{P_1, P_2\};P \iff \{P_1;P, P_2;P\}$

Corollary 1.6.3.5: If $P \iff P'$ then $\{P\} \iff \{P, P'\}$.

Corollary 1.6.3.6: If $P \iff P_1$ and $P \iff P_2$ then $P \iff \{P_1, P_2\}$. 
Unconstrained subprograms

Definition 1.6.3.7: A program is said to be unconstrained if it contains no state constraint at all, or if every state constraint in the program is tautological.
Equivalently constrained programs

Definition 1.6.3.8: Two programs $\land C_1; P_1$ and $\land C_2; P_2$ are said to be *equivalently constrained* if and only if $P_1$ and $P_2$ are both unconstrained and $C_1 \equiv C_2$. 
Equivalently constrained programs (continued)

**Theorem 1.6.3.9**: If $P_1 \Rightarrow P_2$, and if $P_1$ and $P_2$ are equivalently constrained, then $P_1 \Leftrightarrow P_2$. 
Basis for pathwise decomposition

Corollary 1.6.3.10:
\[ \land C_1 \lor C_2; P \iff \{\land C_1; P, \land C_2; P\} \]

Corollary 1.6.3.11:
If \(C_1, C_2, \ldots, C_n\) are \(n\) constraints such that \(C_1 \lor C_2 \lor \ldots \lor C_n \equiv T\) then
\[ P \iff \land C_1 \lor C_2 \lor \ldots \lor C_n; P \iff \{\land C_1; P, \land C_2; P, \ldots, \land C_n; P\} \]
Pathwise vs. traditional decomposition

The present method is called *pathwise decomposition* because the program is divided *along* the control flow, whereas in the traditional method of decomposing a program into procedures and functions, a program is divided *across* the control flow.
An alternative notation

\{\{\{ \ P_1 \}

\,, \ P_2 \}

\,, \ P_3 \}

\,, \, \, \, \, \, \, \, 

\,, \ P_n \}

\} \} \}
Constraining a program set

Corollary 1.6.3.14:

\[ \land C \{ P_1, P_2 \} \iff \{ \land C; P_1, \land C; P_2 \} \]

Corollary 1.6.3.15:

\[ \{ P_1; \land C_1, P_2; \land C_2 \} \iff \{ P_1; \land C_1, P_2; \land C_2 \} \land C_1 \lor C_2 \]
Pathwise Decomposition

Since the path structure in a program can be more readily visualized in a graphic form, we shall now introduce a graphic representation of programs.

In this representation scheme, a program is to be represented by a directed graph, called a *program graph*, in which each edge is associated with a one-entry one-exit programming construct (or its symbolic name).
Program graph

A cascade of two edges represents a programming construct formed by a sequence of two components as depicted below.
Program graph (continued)

Two edges are connected in parallel to represent the program set consisting of the associated components as shown below.
Program graph (continued)

A program graph introduced here has an important property, viz., a path (or a set of paths) completely and uniquely represents a subprogram, and vise versa.
Program graph (continued)

if $B$ then $S_1$ else $S_2$
\[\iff \{\land B; \text{if } B \text{ then } S_1 \text{ else } S_2, \land \neg B; \text{if } B \text{ then } S_1 \text{ else } S_2\}\]
\[\iff \{\land B; S_1, \land \neg B; S_2\}\]
Program graph (continued)

while \( B \) do \( S \)

\[ \iff \{ \land B; \text{while } B \text{ do } S, \land \neg B; \text{while } B \text{ do } S \} \]

\[ \iff \{ \land B; S; \text{while } B \text{ do } S, \land \neg B \} \]
Program graph (continued)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)

\( \neg B; \)
Trace subprogram

Definition 1.6.4.1: A subprogram of some program $P$ is called a *trace subprogram* of $P$ if it constitutes one and only one path from the entry to the exit of its control-flow graph.
An example C program

while (i < 10) {
    if a[i] < 0
        sum = sum - a[i];
    else
        sum = sum + a[i];
    i := i + 1;
}

Its program graph

\( i = i + 1 \)

\( \wedge i < 10 \)

\( \wedge (a[i] < 0); \)
\( \text{sum = sum } - a[i] \)

\( \wedge ! (a[i] < 0); \)
\( \text{sum = sum } + a[i] \)
A trace subprogram

\[ \land i < 10; \]
\[ \land a[i] < 0; \]
\[ \text{sum} = \text{sum} - a[i]; \]
\[ i = i + 1; \]
\[ \land !(i < 10); \]