1.0 Introduction

To determine if a given program will do what it is intended to do through analysis or testing, we often need to

- interpret the program specification correctly,
- determine if any part of the program specification is violated (i.e., not satisfied),
- prove that a certain assertion is a theorem (i.e., always true),
- argue for (or against) the correctness of a given program,
- relate a subfunction in the specification to the corresponding subprogram, and
- find an input to test-execute a specific subprogram.

These tasks can be facilitated by using the concepts, notations, and formalisms discussed in this chapter.

1.1 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.

Given propositions, we can form new propositions by combining them with connectives such as "not", "and", "or", etc. The propositional calculus is a method for computing the truth values of propositions that involves connectives.

The connectives of the propositional calculus include:

- negation: \( \neg \), not
- conjunction: \( \land \), and
- disjunction: \( \lor \), or
- implication: \( \rightarrow \), implies, if ... then ...
- equivalence: \( \equiv \), ... if and only if ...

The definitions of these connectives are given below:
Formally, the propositional calculus is a mathematical system in which \{T, F\} is the underlying set, and the connectives are the operations defined on this set.

A propositional variable is a variable that may assume the value of T or F. It denotes a proposition.

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 \( p, q, r, s, \ldots \)), constants (T and F), and parentheses.

The syntax of a wff can be 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.

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):

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

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.

In practice, analysis of a statement can often be facilitated by translating it into a well-formed formula first.

**Example:** Suppose the policy of a pharmaceutical company includes 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.

(Proposition 1.1.1)

Now let us further suppose that the company is developing a new drug with an enormous market potential, and an ambitious manager has just decided to put the drug on the market immediately, despite of the fact that the drug has failed the clinical test. Does this decision to market the drug violate the policy stated as Proposition 1.1.1?

The policy is not violated if it is not made false. To facilitate determination of its truth value, we shall translate it into a well-formed formula as shown below:
A₁: \[ a \land c \supset (m \equiv p \land \neg g) \]

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

It is obvious that, if the drug failed the clinical test, i.e., if c is false, the formula A₁ is true regardless of the assignments made to other variables. That is to say, even though the drug failed to pass the clinical test, the decision to market the drug does not violate the policy represented by A₁.

Note, however, that the formula A₁ represents only one possible translation of Proposition 1.1.1. The same statement can also be taken in such a way that it is translated into the following formula:

A₂: \[ a \land c \supset m \equiv p \land \neg g. \]

In this case, "≡" is the main connective (i.e., the one to be evaluated last, in accordance with the precedence relation defined previously). If c is false, the left-hand side of the "≡" connective is always true, and the formula becomes false only when the right-hand side of the connective becomes false. Since the truth values of p and g are not given for the question in hand, there is insufficient information to determine the truth value of the formula, and thus the tool should indicate that there is insufficient data to evaluate that proposal to market the drug.

The second translation (A₂) appears to be more plausible. It is hard to imagine that a company would adopt a policy allowing marketing of a drug that did not pass the clinical test. (End of Example)

Exercise: Rewrite the statement so that it becomes unambiguous. Can its readability be preserved, or even enhanced?

Definition: If for every assignment of values to its variables a wff has the value T, it is said to be valid (or, a tautology); if it always has the value F then it is said to be contradictory (or, a contradiction). A wff is said to be satisfiable if and only if it is not contradictory. A wff is said to be contingent if and only if it is neither valid nor contradictory.

Notation: If A is a tautology, we write \( \models A \), where \( \models \) is “T” written sideways.

Note that A is a tautology if and only if \( \neg A \) is a contradiction.

It is useful to define certain relationships among propositions so that, if we know the truth value of a proposition, we may be able to say something about the truth values of its relatives. The first relation, a strong one, is 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 \(-A \equiv B.\)

A weaker relation, more frequently encountered in practical applications, is logical consequence.

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

A is called the antecedent and B the consequence if \(A \models B.\)

Theorem: \(A \models B\) if and only if \(-A \supset B.\)

One possible application of the above theorems is to establish a rule of inference known as modus ponens. This rule says that, if we can show that A and \(A \supset B\) are both true then we can immediately assert that \(B\) is also true. The validity of this rule can be established by showing that \((A \land (A \supset B)) \supset B.\)

Alternatively, the inference rule can be stated as

if A and \(A \supset B\) then B.

Here A corresponds to the premise, \(A \supset B\) the argument, and B the conclusion.

To show that an argument is valid is to show that, whenever the premise is true, the conclusion is also true. In symbolic form, it is to show that \(-A \supset B.\) If the argument is valid, truthfulness of the conclusion can be established simply by showing that the premise is true.

Note that if the premise is a contradiction then there is no way to establish the truthfulness of the conclusion through the use of argument \(A \supset B.\) Thus, in practical applications, we should check the consistency of the argument. An argument is said to be consistent if its premise is satisfiable.

By 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 \(-A \supset B\) is to show that if B is false, it is not possible to find an assignment of truth values to all prepositional variables involved that will make A true.

We write \(A_1, A_2, \ldots, A_n \models B\) if the antecedent consists of n propositions such that B is true whenever every component in the antecedent is true.

Theorem: \(A_1, A_2, \ldots, A_n \models B\) if and only if \(-A_1 \land A_2 \land \ldots \land A_n \supset B.\)

The relationship \(A_1, A_2, \ldots, A_n \models B\) is useful in that it has the following two properties: (1) when every \(A_i\) is true, B is also true; and (2) B is false only if some \(A_i\) is false.

This relationship can be used to analyze the validity of a decision. A decision is said to be valid if it does not violate any constraints imposed or contradicts any known facts. Constraints may be company policies, government regulations, software requirements, rules of physics, and the like.
Let $A_1, A_2, \ldots, A_n$ be the constraints and $B$ be the decision. Then to show that the decision is valid is to show that $B$ is the consequence of $A_1, A_2, \ldots, A_n$, i.e., $\vdash A_1 \land A_2 \land \cdots \land A_n \supset B$.

A common way to construct the proof of $\vdash A_1 \land A_2 \land \cdots \land A_n \supset B$ is to show that, if we let $B$ be false, it would be impossible to make all $A_i$'s true at the same time.

It is interesting to see what will happen if additional constraints are imposed on the decision-making process.

Let us suppose that, in addition to the policy expressed as Proposition 1.1.1, which is repeated below for convenience,

$$A_1 \text{ or } A_2: \text{If the drug passes both animal test and clinical test, then the company will market it if and only if it can be produced and sold profitably and the government does not intervene.}$$

the company further stipulates that

$$A_3: \text{If the drug cannot be produced and sold profitably, it should not be marketed.}$$

and requires its decision makers to keep in mind that

$$A_4: \text{If the drug failed the animal test or clinical test and the drug is marketed, the government will definitely intervene.}$$

Also remember that the drug failed the clinical test. This fact is to be denoted by $A_5$.

Let us see if the following is a tautology:

$$\vdash A_2 \land A_3 \land A_4 \land A_5 \supset B$$

where

$$A_2: a \land c \supset m \equiv p \land \neg g$$

$$A_3: \neg p \supset \neg m$$

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

$$A_5: \neg c$$

$$B: \neg m$$

The proof can be constructed as follows.

1. Assume that $B$ is false by assigning $F$ to $m$, i.e., $m \leftarrow T$.
2. To make $A_5$ true, $c \leftarrow F$.
3. To make $A_3$ true, $p \leftarrow T$.
4. To make $A_4$ true, $g \leftarrow T$.
5. To make $A_2$ true, we need to do $g \leftarrow F$. This contradicts what we have done in (4).
This shows that it is impossible to make B false and all antecedents (i.e., A₂, A₃, A₄, and A₅) true at the same time, and thus $A₂ \land A₃ \land A₄ \land A₅ \supset B$ is a tautology. That is to say, the policies represented by A₂, A₃, and A₄, and the fact represented by A₅ dictate that the drug should not be marketed. It is impossible to market the drug without contradicting A₅ or violating at least one of the policies represented by A₂, A₃, and A₄.

Note that the constraints A₂ through A₅ are consistent in that it is possible to find an assignment to all propositional variables involved so that all the constraints are true at the same time. See rows 2 and 18 of the truth table in the following.

It is interesting to observe that $A₁ \land A₃ \land A₄ \land A₅ \supset B$ is not a tautology. The disproof can be constructed in exactly the same way as demonstrated above except that, in step (5), we will have to find an assignment to make $A₁$: $a \land c \supset (m \equiv p \land \neg g)$ true. We will have no problem doing that because $A₁$ is already made true in step (2), when we set c to F.

The truth table given at the end of this section may be helpful in clarifying the preceding discussions. Rows corresponding to interesting cases are highlighted. In particular, rows 7 and 23 correspond to the case where a negative decision can be made without violating any constraints if Proposition 1.1.1 is translated into $A₁$ instead of $A₂$.

**Exercise:** Show that no valid decision, on whether or not to market the drug, can be made solely based on the policy stated as Proposition 1.1.1 and the fact that the drug failed the clinical test.
The truth table for

\[ A_1: \ a \land c \supset (m \equiv p \land \neg g) \]
\[ A_2: \ a \land c \supset m \equiv p \land \neg g \]
\[ A_3: \ \neg p \supset \neg m \]
\[ A_4: \ (\neg a \lor \neg c) \land m \supset g \]
\[ A_5: \ \neg c \]
\[ B: \ \neg m \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>m</th>
<th>p</th>
<th>g</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>14</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>16</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>17</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>18</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>19</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>20</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>22</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>23</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>24</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>25</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>26</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>27</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>28</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>29</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>30</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>31</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
1.2 The First-Order Predicate Calculus

The power of the propositional calculus is quite limited in that it can only deal with propositions, i.e., sentences that are either true or false. In many applications, we have to deal with sentences such as

\[ \text{She is a graduate student.} \]
\[ x > 0 \]

Without knowing who she is, or what the value of \( x \) is, we will not be able to tell if these sentences are true or false. However, once a particular person is assigned to the pronoun "she", or a number is assigned to \( x \), these sentences will become either true or false. These are called *sentential forms*. They cannot be treated in the propositional calculus.

The first-order predicate calculus can be viewed as an extension of the propositional calculus that includes facilities for handling sentential forms as well as propositions.

The language of the first-order predicate calculus includes all symbols for the logical operations and for propositions. In addition, it makes use of the following symbols:

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

The syntax of the language can be recursively defined as follows:

**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.

**Definition:** A string is an *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.

**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 the \( \neg A \), \( (A \land B) \), \( (A \lor B) \), \( (A \supset B) \), and \( (A \equiv B) \) are wffs.
4. Those and only those obtained by 1, 2, and 3 are wffs.
The 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. For example, in the wff

\[ P(x) \land (\exists x)(Q(x) \equiv (\forall y)R(y)) \]

the first occurrence of \(x\) is free because it is not within the scope of any quantifier, while the second and third occurrences of \(x\) and the occurrences of \(y\) are all bound. Thus a variable may have both free and bound occurrences within a wff.

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.

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 \(D\).

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.

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.

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

This interpretation of the wff yields the following statement:

- For every integer \(x\), \(x + 1 > 0\),

which is obviously false.

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
\[ P(u, v): \text{ } u \text{ is equal to } v \]
\[ f(x, y): \text{ } x + y \]
\[ a: \text{ } 0 \]

The interpreted formula can be restated as \((\forall x)(\exists y)(x + y = 0)\), and is a true statement.

Observe that the order in which the quantifiers are given is important, and cannot be arbitrarily changed. For instance, if we interchange the quantifiers of the above wff, the interpreted statement will change from

For every integer \( x \), there exists another integer \( y \), such that \( x + y = 0 \),

which is true, to an entirely different statement

There exists an integer \( y \), such that for every integer \( x \), \( x + y = 0 \),

which is obviously false.

Listed below are some theorems in the first-order predicate calculus that can be used to govern the movement of quantifier in a wff.

**Theorems:**

1. \((\exists x)(\exists y)A \equiv (\exists y)(\exists x)A\)
2. \((\forall x)(\forall y)A \equiv (\forall y)(\forall x)A\)
3. \((\forall x)(A \supset B) \equiv ((\exists x)A \supset B)\) where \( x \) does not occur free in \( B \).
4. \((\exists x)(A \supset B) \equiv ((\forall x)A \supset B)\) where \( x \) does not occur free in \( B \).
5. \((\forall x)(A \supset B) \equiv (A \supset (\forall x)B)\) where \( x \) does not occur free in \( A \).
6. \((\exists x)(A \supset B) \equiv (A \supset (\exists x)B)\) where \( x \) does not occur free in \( A \).

The correctness of these theorems can be readily verified informally by considering the cases in which the domain is a finite set, say, \( D = \{x_1, x_2, \ldots, x_n\} \). Theorem (3), for example, can then be rewritten as

\[(A(x_1) \supset B) \land (A(x_2) \supset B) \land \ldots \land (A(x_n) \supset B) \equiv (A(x_1) \lor A(x_2) \lor \ldots \lor A(x_n)) \supset B\]

and Theorem (4) as

\[(A(x_1) \supset B) \lor (A(x_2) \supset B) \lor \ldots \lor (A(x_n) \supset B) \equiv (A(x_1) \land A(x_2) \land \ldots \land A(x_n)) \supset B\]

Since these can be treated as formulas in the propositional calculus, the equivalence relations can be readily verified by using a truth table.

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
- \( A(x, y): x \text{ divides } y \)
B(x, y): x ≤ y

With this interpretation (3) reads \((\forall x)("x divides y" \implies x \leq y) \equiv (\exists x)(x divides y) \implies x \leq y\). Although the left-hand side of the "≡" 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.

Now if we interpret B(x, y) to be \((\exists x)((y \div x)x=y)\), (3) reads \((\forall x)("x divides y" \implies (\exists x)((y \div x)x=y)) \equiv (\exists x)(x divides y) \implies (\exists x)((y \div x)x=y)\). The equivalence relation holds because x does not occur free in B.

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 \((\forall x)("x divides y" \implies "y is not prime") \equiv (\exists x)(x divides y) \implies "y is not prime"\).

In many cases, the truth value of a wff can be more readily evaluated if we transform the wff into a canonical form described below.

**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.

The following rules (logically equivalent relations) can be utilized to transform a given wff into its prenex normal form:

1. \((\exists x)A(x) \equiv (\forall x)(\neg A(x))\)
2. \((\forall x)A(x) \equiv (\exists x)(\neg A(x))\)
3. \((Qx)A(x) \lor B \equiv (Qx)(A(x) \lor B)\), where x does not occur free in B.
4. \((Qx)A(x) \land B \equiv (Qx)(A(x) \land B)\), where x does not occur free in B.
5. \((\exists x)A(x) \lor (\exists x)C(x) \equiv (\exists x)(A(x) \lor C(x))\)
6. \((\forall x)A(x) \land (\forall x)C(x) \equiv (\forall x)(A(x) \land C(x))\)
7. \((Q_1x)A(x) \lor (Q_2x)C(x) \equiv (Q_1x)(Q_2y)(A(x) \lor C(y))\)
8. \((Q_3x)A(x) \land (Q_4x)C(x) \equiv (Q_3x)(Q_4y)(A(x) \land C(y))\)

Here in (4a) and (4b) y is a variable that does not occur in A(x). Q, Q_1, Q_2, Q_3, and Q_4 are either \(\exists\) or \(\forall\).

In order to make the above rules applicable, it may be necessary to rename variables and rewrite the formula into an equivalent one using \(\neg, \land, \text{ and } \lor\) connectives only.

**Example:** Consider the wff \((\forall x)P(x) \land (\exists x)Q(x) \lor (\exists x)R(x)\), which can be rewritten into the prenex normal form as follows.
This sequence of transformation is valid provided that \( x \) does not occur free in \( Q \) and \( R \), \( y \) does not occur free in \( P \) and \( R \), and \( z \) does not occur free in \( P \) and \( Q \). In applying the transformation rules, always select a new variable name such that no free variable becomes bound in the process.

To illustrate, let us consider the following logical expression:

\[
(\forall x)P(x) \land (\exists x)Q(x) \lor \neg(\exists x)R(x)
\]

By (1a)

\[
(\forall x)(\exists y)(P(x) \land Q(y)) \lor (\forall x)(\neg R(x))
\]

By (4b)

\[
(\forall x)(\exists y)(\forall z)(P(x) \land Q(y) \lor \neg R(z))
\]

By (4a)

This sequence of transformation is valid provided that \( x \) does not occur free in \( Q \) and \( R \), \( y \) does not occur free in \( P \) and \( R \), and \( z \) does not occur free in \( P \) and \( Q \). In applying the transformation rules, always select a new variable name such that no free variable becomes bound in the process.

To illustrate, let us consider the following logical expression:

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

(A)

In program testing, the above may represent the condition under which a specific program path will be traversed, and the problem is to find an assignment of values to the input variables \( a \), \( b \), and \( e \) such that the condition is satisfied.

One systematic way to do this is to restate the condition in terms of equality as follows:

\[
(\exists x)D_1(x = b - a - e) \land (\exists x)D_2(x = b + 2a - 6) \land (\exists x)D_2(x = e - 2(b - a)/3)
\]

where \( D_1 \) is the set of all real numbers greater than 0, and \( D_2 \) the set of all real numbers greater than or equal to 0.

The task can be made more manageable by rewriting it into its prenex normal form shown below:

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

(B)

The three equations above are indeterminate because there are more than three variables involved. Therefore, we cannot directly obtain the desired assignment by solving the equations. However, we can combine these three equations to form a new equation in such a way that the number of variables involved in the new equation will be minimal. This can be accomplished by using the same techniques we use in solving simultaneous equations. In the present example, we can combine the three equations to yield

\[
(\exists x)D_1(\exists y)D_2(\exists z)D_2(3x - y + 3z = 6 - 3a)
\]

(C)

As indicated in the above expression, the requirements on the assignments to \( x \), \( y \), and \( z \) are that \( x > 0 \), \( y \geq 0 \), and \( z \geq 0 \). So let us begin by making the following assignments:

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

Then (C) can be satisfied by letting

\[
a \leftarrow 1.9
\]
To satisfy the second component in (B) we must have $0 = b + 2 \times 1.9 - 6 = b - 2.2$, i.e., we have to make the assignment:

$$b \leftarrow 2.2$$

Finally, the first and the third components of (B) can be satisfied by letting

$$e \leftarrow 0.2$$

In summary, logical expression (A) can be satisfied by the following assignment:

$$a \leftarrow 1.9 \quad b \leftarrow 2.2 \quad e \leftarrow 0.2$$

### 1.3 Principle of Mathematical Induction

The set of (non-negative) integers have many interesting properties, chief among them are that (1) the numbers can be constructed (or generated) from 0 uniquely, and (2) if a property that holds for one number also holds for the next number in the generation, then that property holds for all integers.

The second property noted is the gist of the principle of mathematical induction, which has so many applications in computer programming that it requires some discussion.

**Definition:** *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.

The procedure is to prove that

(a) 0 is $P$  

(induction basis),

(b) $n$ is $P$  

(induction hypothesis),

(c) $n+1$ is $P$  

(induction step)

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

(d) $n$ is $P$ for all $n$.

For example, suppose we wish to prove that
\[
\sum_{i=0}^{n} i = \frac{n(n+1)}{2}
\]

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

\[
n \in P \iff \sum_{i=0}^{n} i = \frac{n(n+1)}{2}
\]

(a) For the basis of the induction we have, for \(n = 0\), \(0 = \frac{0(0+1)}{2}\) which is true.

(b) The induction hypothesis is that \(k \in P\) for some arbitrarily choice of \(k\):

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

(c) For the induction step, proving \(k+1 \in P\), we have

\[
\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k + 1) \quad \text{(using the induction hypothesis)}
\]

\[
= \frac{kk + k + 2k + 2}{2} = \frac{(k + 1)((k + 1) + 1)}{2}
\]

(d) Hence \(k+1\) has the property \(P\).

The principle of induction is also valid if at step (b), the induction hypothesis, we assume that every \(k \leq n\) is \(P\). Moreover, one may choose any integer as a basis and then prove that some property of interest holds for the set of integers greater than or equal to the basis.

A closely related concept, which is commonly used in computer programming, is the inductive definition of a set or property having the standard form:

**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)

We have already seen many examples of inductive definitions in the preceding section, where all well-formed formulas are defined inductively.
1.4 Proving Programs Correct

A common task in program verification is to show that, for a given program $S$, if a certain *precondition* $Q$ is true before the execution of $S$ then a certain *postcondition* $R$ is true after the execution, provided that $S$ terminates. This logical proposition is commonly denoted by $Q \{ S \} R$ for short (a notation due to Hoare [HOAR69]). If we succeeded in showing that $Q \{ S \} R$ is a theorem (i.e., always true), then to show that $S$ is *partially correct* [LOND77], with respect to some input predicate $I$ and output predicate $\emptyset$, is to show that $I \supset Q$ and $R \supset \emptyset$ (see, e.g., [ANDE79,MANN74]).

In this conceptual framework, verification of partial 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 $R \supset \emptyset$. 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 due to Basu and Yeh [BAYE75] and Dijkstra [DIJK76] to find the weakest precondition. 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 inductive assertions (see, e.g.,[MANN74, LOND77]).

To fix the idea, we shall first assume that programs are written in a structured Pascal-like language that includes the following constructs:

1. assignment statements: $x := e$;
2. conditional statements: if $B$ then $S$ else $S'$; (note: $S$ and $S'$ are statements, and $B$ is a predicate; the else clause may be omitted);
3. repetitive statements: while $B$ do $S$; or, repeat $S$ until $B$;

and a program is constructed by concatenating such statements.

As a concrete example, consider the following program for performing integer division:

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

In words, this program divides $x$ by $y$, and stores the quotient and the remainder in $q$ and $r$, respectively. Suppose we wish to verify that program INTDIV is partially correct with respect to input predicate $x \geq 0 \land y > 0$ and output predicate $x = r + q \times y \land r < y \land r \geq 0$, 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 $\emptyset$, the basic problem is to find as weak as possible a condition $Q$ such that $Q(S) \subseteq \emptyset$, and then determine if $I \supseteq Q$.

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 $wp(S, R)$ denotes the weakest precondition for the initial state such that an execution of $S$ will properly terminate, leaving it in a final state satisfying the condition $R$.

The entity $wp(S, R)$ can be considered as a function of $R$, is called a predicate transformer in the literature [BAYE75], and has the following properties:

1. For any $S$, $wp(S, F) \equiv F$
2. For any programming construct $S$ and any predicates $Q$ and $R$, if $Q \supseteq R$ then $wp(S, Q) \supseteq wp(S, R)$.
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 $R$,
   \[(wp(S, Q) \lor wp(S, R)) \equiv wp(S, Q \lor R)\]

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 $wp(skip, R) \equiv R$ for any predicate $R$. The statement abort, when executed, will not lead to a final state. Its meaning is defined as $wp(abort, R) \equiv F$ for any predicate $R$.

In terms of the predicate transformer, the meaning of an assignment statement can be given as $wp(x := E, R) \equiv R_{E \to x}$, where $R_{E \to x}$ is a predicate obtained from $R$ by substituting $E$ for every occurrence of $x$ in $R$. The examples listed below should clarify the meaning of this notation.

<table>
<thead>
<tr>
<th>$R$</th>
<th>$x := E$</th>
<th>$R_{E \to x}$</th>
<th>which can be 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>$x \neq y$</td>
<td>$x := x - y$</td>
<td>$x - y \neq y$</td>
<td>$x \neq 2y$</td>
</tr>
</tbody>
</table>

For a sequence of two programming constructs $S_1$ and $S_2$, $wp(S_1; S_2, R) \equiv wp(S_1, wp(S_2, R))$. The weakest precondition of an if-then-else statement is defined to be $wp(if B then S_1 else S_2, R) \equiv B \land wp(S_1, R) \lor \neg B \land wp(S_2, R)$.

For the iterative statement, Basu and Yeh [BAYE75] have shown that $wp(while B do S, R) \equiv (\exists j \geq 0)(A_j(R))$, where $A_0(R) \equiv \neg B \land R$ and $A_{j+1}(R) \equiv B \land wp(S, A_j(R))$ for all $j \geq 0$. 
In practice, the task of finding the weakest precondition of an iterative statement is often hopelessly complex. This difficulty constitutes a major hurdle in proving programs correct by using the predicate transformation method.

To illustrate, consider the example program given previously.

```
INTDIV: begin
    q := 0; r := x;
    while r ≥ y do begin r := r - y; q := q + 1 end
end.
```

We can prove the correctness of this program by first computing

\[ wp(\text{while } r \geq y \text{ do begin } r := r - y; q := q + 1 \text{ end}, x = r + q \times y \land r < y \land r \geq 0) \]

where

- \( B \equiv r \geq y \)
- \( R \equiv x = r + q \times y \land r < y \land r \geq 0 \)
- \( S: r := r - y; q := q + 1; \)

\( A_0(R) \equiv \neg B \land R \equiv r < y \land x = r + q \times y \land r < y \land r \geq 0 \land x = r + q \times y \land r < y \land r \geq 0 \)

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

\( A_2(R) \equiv B \land wp(S, A_1(R)) \equiv x = r + q \times y \land r < 3 \times y \land r \geq 2 \times y \)

\( A_3(R) \equiv B \land wp(S, A_2(R)) \equiv x = r + q \times y \land r < 4 \times y \land r \geq 3 \times y \)

From these we may guess that

\( A_j(R) \equiv B \land wp(S, A_{j-1}(R)) \equiv x = r + q \times y \land r < (j+1) \times y \land r \geq j \times y \)

and we have to prove that our guess is correct by mathematical induction. Assume that \( A_j(R) \) is as given above, then

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

\( A_{j+1}(R) \equiv B \land wp(S, A_j(R)) \equiv r \geq y \land wp(r := r - y; q := q + 1, x = r + q \times y \land r < (j+1) \times y \land r \geq j \times y) \)
\begin{align*}
\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
\end{align*}

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.

Hence

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

Next, we compute

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

which is implied by the input condition \(x \geq 0 \land y > 0\), and hence the proof that

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

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 [MANN74]. 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" [BAYE75].

The Inductive Assertion Method: Top-Down Approach

In the second approach, given a program \(S\) and a predicate \(Q\), the basic problem is to find as strong as possible a condition \(R\) such that \(Q\{S\}R\).

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 \(x = E' \equiv x' = E^{-1}\).

In practice, predicate \(Q'[x := E']_{x' \rightarrow E^{-1}}\) is constructed as follows. Given \(Q\) and \(x := E\),
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).

The following examples should clarify the definition given above.

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( x := E )</th>
<th>( (Q' \land x = E')_{x' \rightarrow E^{-1}} ) which can be simplified to</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 0 )</td>
<td>( x := 10 )</td>
<td>( T \land x = 10 ) ( x = 10 )</td>
</tr>
<tr>
<td>( a &gt; 1 )</td>
<td>( x := 1 )</td>
<td>( a &gt; 1 \land x = 1 ) ( a &gt; 1 \land x = 1 )</td>
</tr>
<tr>
<td>( x &lt; 10 )</td>
<td>( x := x + 1 )</td>
<td>( x - 1 &lt; 10 ) ( x &lt; 11 )</td>
</tr>
<tr>
<td>( x \neq y )</td>
<td>( x := x - y )</td>
<td>( x + y \neq y ) ( x \neq 0 )</td>
</tr>
</tbody>
</table>

In essence, \( (Q' \land x = E')_{x' \rightarrow E^{-1}} \) denotes the strongest postcondition for the final state if an execution of \( x := E \), with the initial state satisfying \( Q \), terminates.

As explained earlier, it is convenient to use \( \vdash P \) to denote the fact that \( P \) is a theorem (i.e., always true). A verification rule may be stated in the form "if \( \vdash X \), then \( \vdash Y \)," which says that if proposition \( X \) has been proved as a theorem then \( Y \) also is thereby proved as a theorem.

Note that \( \vdash Q[S]R \) implies \( \vdash Q[S]R \), but not the other way around. The student should find an example of this fact.

We shall now proceed to give some useful verification rules. As given previously, for an assignment statement of the form \( x := E \), we have

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

For a conditional statement of the form \( \text{if } B \text{ then } S_1 \text{ else } S_2 \) we have

\[
\text{If } \vdash Q \land B \{S_1\}R_1 \text{ and } \vdash Q \land \neg B \{S_2\}R_2 \text{ then } \vdash Q\{\text{if } B \text{ then } S_1 \text{ else } S_2\}R_1 \lor R_2. \quad \text{(Rule 2)}
\]

For a loop construct of the form \( \text{while } B \text{ do } S \) we have

\[
\text{If } \vdash Q \supset R \text{ and } \vdash (R \land B) \{S\}R \text{ then } \vdash Q\{\text{while } B \text{ do } S\}(\neg B \land R). \quad \text{(Rule 3)}
\]

The above relation is commonly known as the invariant-relation theorem, and any predicate \( R \) satisfying the premise is called a loop invariant of the loop construct \( \text{while } B \text{ do } S \).
Thus the partial correctness of program $S$ with respect to input condition $I$ and output condition $\emptyset$ can be proved by showing that $I \{ S \} Q$ and $Q \supset \emptyset$. 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 $I \{ S_1; S_2; \ldots; S_n \} \emptyset$ can be proved by showing that $I \{ S_1 \} P_1$, $P_1 \{ S_2 \} P_2$, $\ldots$, and $P_{n-1} \{ S_n \} \emptyset$ 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.

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.

To illustrate, consider again the problem of proving the partial correctness of

$$\text{INTDIV: begin}$$
$$\quad q := 0; r := x;$$
$$\quad \text{while } r \geq y \text{ do begin } r := r - y; q := q + 1 \text{ end}$$
$$\text{end.}$$

with respect to the input condition $I \equiv x \geq 0 \land y > 0$ and output condition $\emptyset \equiv x = r + q \times y \land r < y \land r \geq 0$. By Rule 1 we have

$$(x \geq 0 \land y > 0) \{ q := 0 \} (x \geq 0 \land y > 0 \land q = 0)$$

and

$$(x \geq 0 \land y > 0 \land q = 0) \{ r := x \} (x \geq 0 \land y > 0 \land q = 0 \land r = x).$$

From the output condition $\emptyset$ we can guess that $x = r + q \times y \land r \geq 0$ is a loop invariant. This can be verified by the fact that

$$(x \geq 0 \land y > 0 \land q = 0 \land r = x) \supset (x = r + q \times y \land r \geq 0)$$

and

$$(r \geq y \land x = r + q \times y \land r \geq 0) \{ r := r - y; q := q + 1 \} (x = r + q \times y \land r \geq 0)$$

Hence, by Rule 3, we have

$$(r \geq y \land x = r + q \times y \land r \geq 0) \{ \text{while } r \geq y \text{ do begin } r := r - y; q := q + 1 \text{ end} \} (r < y \land x = r + q \times y \land r \geq 0)$$

Thus we have shown, by transitivity of implication, that

$$(x \geq 0 \land y > 0) \{ \text{INTDIV} \} (x = r + q \times y \land r < y \land r \geq 0)$$
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. The problem is that most programs contain nested loops and compound statements, which may render applications of Rules 2 and 3 hopelessly complicated. This difficulty can be alleviated by using a variant of the inductive-assertion method described below.

The complication induced by nested loops and compound statements can be eliminated by representing the program as a flowchart. Appropriate assertions are then 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.

Since programs are assumed to be written in a Pascal-like language as stated before, each node in the flowchart of a program is either a branch predicate or an assignment statement. It follows that the flowchart of any program is formed by three types of simple path depicted below:

\[
\begin{align*}
\text{(a) } & \quad Q \quad \xRightarrow{x := E} \quad R \\
\text{(b) } & \quad Q \quad \xRightarrow{T \lor B} \quad R \\
\text{(c) } & \quad Q \quad \xRightarrow{T \land B} \quad R \\
\end{align*}
\]

The intended function of each basic path is described by the associated lemma that in effect states that, if the starting assertion is true, the ending assertion will also become true when the control reaches the end of the path.

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, i.e., always true. 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.

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.

There are other details that can be best explained by using an example. Let us consider program INTDIV used in the previous discussion. The flowchart of that program is shown below.

Three assertions are used for this example: A is the input predicate, C is the output predicate, and B is the assertion used to cut the loop. Note that 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.

There are three paths between assertions in this flowchart: paths AB, BB, and BC. These paths lead us to the following lemmas that must be proved.

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

Path BB: \( x = r + q \cdot y \land r \geq 0 \land r' = r - y \land q' = q + 1 \supset x = r' + q' \cdot y \land r' \geq 0 \land y > 0 \)  
(Here \( q' \) and \( r' \) denote the new values of \( q \) and \( r \) after the loop body is executed.)

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

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: Substitute \( q \) with \( q' - 1 \) and \( r \) with \( r' + y \) for every occurrence of \( q \) and \( r \) and simplify.
Lemma for Path BC: Use the fact that \( \neg(r \geq y) \) is \( r < y \), and simplify.

A common error made in constructing a correctness proof is that the guessed assertion is either stronger or weaker than the correct one. 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. 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.

1.5 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 [AHSU86]. Briefly, 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. For two edges (labeled by) \( a \) and \( b \), we shall use \( ab \) to describe the path formed by connecting \( a \) in cascade with \( b \), use \( a+b \) to describe the path structure formed by connecting \( a \) and \( b \) in parallel, and use \( a^* \) to describe the loop formed by using \( a \), as shown below:

- \( ab \)
- \( a+b \)
- \( a^* \)

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 (\(*\)). For example, the set of paths between nodes 1 and 4 in the graph shown below can be described by the regular expression \( a(e+bc*d) \).
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 time).

Described below is a method for finding the set of all paths between any two nodes in a directed graph [LUNT65].

Let \( G \) be a directed graph in which each edge is labeled by an element of set \( E \) of symbols. If there are \( n \) nodes in \( G \), then \( G \) 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.

For example, the graph given above can be represented by the following matrix:

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

where \( \emptyset \) is a special symbol representing the empty set.

The operation of concatenation, disjunction (+), and star operation (*) are now to be extended over the matrices with regular expressions as elements. Let \( [X] \), \( [Y] \), \( [Z] \), and \( [W] \) be \( n \times n \) matrices. We define

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

where \( z_{ij} = x_{ij} + y_{ij} \),

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

where \( w_{ij} = \sum_{k=1}^{n} x_{ik}y_{kj} \), and
\[ [X]^* = [X]^0 + [X]^1 + [X]^2 + [X]^3 + \ldots, \]

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 0. If we consider concatenation as multiplication and disjunction as addition, then the first two matrix operations defined above are similar to matrix addition and matrix multiplication, respectively, defined in the theory of matrices.

Now given \([G] = [g_{ij}]\), the matrix representation of a graph \(G\) having \(n\) nodes, we may construct a \((n-1) \times (n-1)\) matrix \([B]\) by simultaneously eliminating the \(k\)-th row and the \(k\)-th column of \([G]\) (for some \(1 \leq k \leq n\)) as follows.

\[
[G] = \begin{bmatrix}
g_{11} & \cdots & g_{1k} & \cdots & g_{1n} \\
\vdots & & \vdots & & \vdots \\
g_{k1} & \cdots & g_{kk} & \cdots & g_{kn} \\
\vdots & & \vdots & & \vdots \\
g_{n1} & \cdots & g_{nk} & \cdots & g_{nn}
\end{bmatrix}
\]

\[
[B] = \begin{bmatrix}
g_{11} & \cdots & g_{1(k-1)} & g_{1(k+1)} & \cdots & g_{1n} \\
\vdots & & \vdots & & \vdots & & \vdots \\
g_{(k-1)1} & \cdots & g_{(k-1)(k-1)} & g_{(k-1)(k+1)} & \cdots & g_{(k-1)n} \\
\vdots & & \vdots & & \vdots & & \vdots \\
g_{(k+1)1} & \cdots & g_{(k+1)(k-1)} & g_{(k+1)(k+1)} & \cdots & g_{(k+1)n} \\
\vdots & & \vdots & & \vdots & & \vdots \\
g_{n1} & \cdots & g_{n(k-1)} & g_{n(k+1)} & g_{nn}
\end{bmatrix}
\]

\[
= \begin{bmatrix}
g_{1k} \\
\vdots \\
g_{(k-1)k} \\
\vdots \\
g_{nk}
\end{bmatrix} \ast \begin{bmatrix}
g_{kl} & \cdots & g_{k(k-1)} & g_{k(k+1)} & \cdots & g_{kn}
\end{bmatrix}
\]

It should not be difficult to see that eliminating a row and the corresponding column in \([G]\) in this way does not alter the path information between any pair of the remaining nodes. In other words, matrix \([B]\) represents the graph obtained by eliminating the \(k\)-th node in the original graph without removing the associated edges.

Thus to find the paths leading from the \(i\)-th node to the \(j\)-th node in \(G\), we simply use the method described above to successively eliminate (in any order) all nodes other than the \(i\)-th and the \(j\)-th nodes. We will then be left with a \(2 \times 2\) matrix (assuming \(i < j\)):
Then \( p_{ij} \), the regular expression denoting the paths leading from the i-th node to the j-th node 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 \( b_{ii} = b_{ji} = b_{jj} = 0 \), which is almost always the case in many applications, then we have

\[
p_{ij} = b_{ij}
\]

because \( 0* = \lambda \) and \( \lambda a = a\lambda = a \) for any regular expression \( a \).

To illustrate, let us suppose that we wish to find the set of all paths leading from node 1 to node 4 in the graph given previously. The matrix representation of the graph is repeated below for convenience.

\[
[B'] = \begin{bmatrix}
b_{ii} & b_{ij} \\
b_{ji} & b_{jj}
\end{bmatrix}
\]

According to the method described above, we can eliminate, for example, column 2 and row 2 (i.e., node 2) first to yield the following \( 3 \times 3 \) matrix.

\[
\begin{bmatrix}
0 & a & 0 & 0 \\
0 & 0 & b & e \\
0 & 0 & c & d \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

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

Hence, the set of paths leading from node 1 to node 4 is described by \(ae + abc^*d\).

1.6 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. In the literature, reference to a program component of that nature is usually made indirectly through a graphical description of the execution path, which rarely provides any insight into the problem at hand.

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.

1.6.1 The Concept of a State Constraint

Consider a restrictive clause of the form:

\begin{center}
The program state at this point must satisfy predicate C, or else the program becomes undefined.
\end{center}

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 \(\ldblash \ C \rdblash\).

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: $S_1; \mathcal{C}; S_2$

Program 1.6.1.2 is said to be created from Program 1.6.1.1 by constraining the program states to $C$ prior to execution of $S_2$. Intuitively, Program 1.6.1.2 is a subprogram of Program 1.6.1.1 because its definition is that of program 1.6.1.1 restricted to $C$. Within that restriction, Program 1.6.1.2 performs the same computation as Program 1.6.1.1.

A state constraint is a semantic modifier. The meaning of a program modified by a state constraint can be formally defined in terms of Dijkstra's *weakest precondition* [DIJK76] as follows. Let $S$ be a programming construct and $C$ be a predicate, then for any postcondition $R$,

**Axiom 1.6.1.3:** $\text{wp}(\mathcal{C}; S, R) \equiv C \land \text{wp}(S, R)$.

**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 $S_1 \leftrightarrow S_2$.

**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 $S_1 \Rightarrow S_2$.

With these definitions, one can now determine the relationship between any programs, with or without state constraints. For instance, consider Programs 1.6.1.1 and 1.6.1.2 again. Since $\text{wp}(S_1; \mathcal{C}; S_2, R) \equiv \text{wp}(S_1, \text{wp}(\mathcal{C}; S_2, R)) \equiv \text{wp}(S_1, C \land \text{wp}(S_2, R)) \equiv \text{wp}(S_1, C) \land \text{wp}(S_1, \text{wp}(S_2, R)) \equiv \text{wp}(S_1; \mathcal{C}; S_2, R)$, it follows that $\text{wp}(S_1; \mathcal{C}; S_2, R) \supset \text{wp}(S_1; S_2, R)$. Thus, by Def. 1.6.1.5, Program 1.6.1.2 is a subprogram of Program 1.6.1.1.

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

**Corollary 1.6.1.6:** $\mathcal{T}; S \leftrightarrow S$

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 $\text{wp}(\mathcal{F}; S, R) \equiv F \land \text{wp}(S, R) \equiv F \equiv \text{wp}(\mathcal{F}; S', R)$ for any $S, S'$, and $R$, and therefore

**Corollary 1.6.1.7:** $\mathcal{F} \leftrightarrow \mathcal{F}; S'$.

In words, any two programs are (trivially) equivalent if both are constrained by a predicate that can never be true.

### 1.6.2 Subprogram Simplification

* The weakest precondition of program $S$ with respect to postcondition $Q$, commonly denoted by $\text{wp}(S, Q)$, is defined as the weakest condition for the initial state of $S$ such that activation of $S$ will certainly result in a properly terminating happening, leaving $S$ in a final state satisfying $Q$ [3].
We shall now present three categories of equivalence relations showing how state constraints can be inserted into a program to reduce it to a simpler subprogram. The first indicates how to constrain a program so that the resulting subprogram will have a simpler logical structure; the second shows how the state constraints in a program can be manipulated and simplified. The third shows how to simplify long sequences of assignment statements, which often result from a repeated application of the first two.

Listed below are equivalence relations of the first category, the validity of which is immediately obvious.

**Corollary 1.6.2.1:** \( /\neg B; \text{if } B \text{ then } S_1 \text{ else } S_2 \iff /\neg B; S_1 \)

**Proof** By Axiom 1.6.1.3, \( \text{wp}(/\neg B; \text{if } B \text{ then } S_1 \text{ else } S_2, R) \)
\[ \equiv B \land \text{wp}(\text{if } B \text{ then } S_1 \text{ else } S_2, R) \]
\[ \equiv B \land (B \land \text{wp}(S_1, R) \lor \neg B \land \text{wp}(S_2, R)) \]
\[ \equiv B \land \text{wp}(S_1, R) \]
\[ \equiv \text{wp}(/\neg B; S_1, R). \]

Q.E.D.

**NOTE** Many corollaries and theorems in Sec. 1.6 can be simply proved in a similar manner, and thus are given without proof.

The meaning of a programming construct, when given in the same font as the main text, is defined as usual. Allowable statements include:

1. assignment statements: \( x := e \), where \( x \) is a variable and \( e \) is an expression,
2. conditional statements: \( \text{if } B \text{ then } S_1 \text{ else } S_2 \), where \( S_1 \) and \( S_2 \) are statements, and \( B \) is a predicate, and
3. repetitive statements: \( \text{while } B \text{ do } S \) or, \( \text{repeat } S \text{ until } B \).

A simple statement is to be delimited by a semicolon (;), and a compound statement by a `begin`-`end` pair.

When an example program is given in a particular programming language, or when a reference is made to the part thereof, the text will be printed in Courier. [end of NOTE]

**Corollary 1.6.2.2:** \( /\neg B; \text{if } B \text{ then } S_1 \text{ else } S_2 \iff /\neg B; S_2 \)

**Corollary 1.6.2.3:** \( /\neg B_1 ; \text{if } B_1 \text{ then } S_1 \text{ else } S_2 \text{ ... else if } B_i \text{ then } S_i \text{ ... else if } B_n \text{ then } S_n \)
\[ \iff /\neg B_1 \land /\neg B_2 \land ... \land /\neg B_i \; ; \; S_i \]

**Corollary 1.6.2.4:** \( /\neg B; \text{while } B \text{ do } S \iff /\neg B; S; \text{while } B \text{ do } S \)

**Corollary 1.6.2.5:** \( /\neg B; \text{while } B \text{ do } S \iff /\neg B \)
Corollary 1.6.2.6: /\B; while B do S ⇔ /\B; repeat S until ¬B

To demonstrate how these relations can be utilized to construct a subprogram with a simpler logical structure, consider the following C program:

Program 1.6.2.7:

```c
int atoi(char s[])
{
    int i, n, sign;
    i = 0;
    while (isspace(s[i]))
        i = i + 1;
    if (s[i] == '-'
        sign = -1;
    else
        sign = 1;
    if (s[i] == '+' || s[i] == '-'
        i = i + 1;
    n = 0;
    while (isdigit(s[i]))
    {
        n = 10 * n + (s[i] - '0');
        i = i + 1;
    }
    return sign * n;
}
```

In light of the equivalence relations given above, one can produce a subprogram with simple logical structures by constraining the program states as shown below:

⇒ int atoi(char s[])
{
    int i, n, sign;
    i = 0;
    /\ !(isspace(s[i]));
    while (isspace(s[i]))
        i = i + 1;
    /\ !(s[i] == '-');
    if (s[i] == '-'
        sign = -1;
    else
        sign = 1;
    /\ !(isnum(s[i]));
    while (isnum(s[i]))
    {
        n = 10 * n + (s[i] - '0');
        i = i + 1;
    }
    return sign * n;
}

which can be simplified to

⇔ int atoi(char s[])
{
    int i, n, sign;
\[ i = 0; \]
\[ \text{!isspace(s[i])}; \]
\[ \text{!}(s[i] == '\ '-''); \]
\[ \text{sign} = 1; \]
\[ \text{!}(s[i] == '+' || s[i] == '-'); \]
\[ n = 0; \]
\[ \text{isdigit(s[i])}); \]
\[ n = 10 * n + (s[i] - '0'); \]
\[ i = i + 1; \]
\[ \text{while (isdigit(s[i]))} \{ \]
\[ \quad n = 10 * n + (s[i] - '0'); \]
\[ \quad i = i + 1; \]
\[ \} \]
\[ \text{return sign} * n; \]

Note that Corollary 1.6.2.4 can be repeatedly applied to a loop construct to produce potentially an infinite number of subprograms. Subprograms of a loop construct will be discussed in detail later.

A subprogram usually can be simplified further if it does not contain any control statement at all. To illustrate, Corollary 1.6.2.5 is applied to the "while" statement above to reduce it to a loopless subprogram.

**Program 1.6.2.8:**

\[
\text{int atoi(char s[])} \]
\[
\{ \int i, n, \text{sign}; \]
\[
i = 0; \]
\[
\text{!isspace(s[i])}; \]
\[
\text{!}(s[i] == '\ '-''); \]
\[
\text{sign} = 1; \]
\[
\text{!}(s[i] == '+' || s[i] == '-'); \]
\[
n = 0; \]
\[
\text{isdigit(s[i])}); \]
\[
n = 10 * n + (s[i] - '0'); \]
\[
i = i + 1; \]
\[
\text{!isdigit(s[i])}); \]
\[
\text{return sign} * n; \}
\]

This is one of infinitely many canonical subprograms (defined below) that can be produced from Program 1.6.2.7.

**Definition 1.6.2.9:** A subprogram is said to be canonical if it does not contain any control statement.

This kind of subprogram is of particular interest because it has the simplest logical structure. Note that there are two state constraints at the beginning of Program 1.6.2.8. The meaning of two consecutive state constraints is given by the following relation, which is a direct consequence of Axiom 1.6.1.3.

**Corollary 1.6.2.10:** \[ \text{C}_1 \land \text{C}_2 \Leftrightarrow \text{C}_1 \land \text{C}_2 \land \text{S}. \]
Program 1.6.2.8 can thus be rewritten as

**Program 1.6.2.11:**

```c
int atoi(char s[])
{
    int i, n, sign;
    i = 0;
    \!(isspace(s[i])) && !(s[i] == '-');
    sign = 1;
    \!(s[i] == '+' || s[i] == '-');
    n = 0;
    \!(isdigit(s[i]));
    n = 10 * n + (s[i] - '0');
    i = i + 1;
    return sign * n;
}
```

A state constraint not only directly constrains the program state at the point where it is placed, but also indirectly at other points upstream and downstream in control flow as well. Note that, in Program 1.6.1.2, predicate C is true if and only if \( \wp(S_1, C) \) is true before execution of \( S_1 \). Thus, by constraining the program state between \( S_1 \) and \( S_2 \) to \( C \), it also indirectly constrains the program state before \( S_1 \) to \( \wp(S_1, C) \), and the program state after \( S_2 \) to \( R \), where \( \wp(S_2, R) \equiv C \).

The *scope* of a state constraint, which is defined to be the range of control flow within which the constraint has an effect, may or may not span the entire program. A state constraint will have no effect beyond a statement that undefines, or assigns a constant value to, the variables involved. For instance, a state constraint like \( x > 0 \) will have no effect on the program states beyond statement \( \text{read}(x) \) upstream, statement \( \text{return} \) downstream if \( x \) is a local variable, or statement \( x := 4 \) upstream or downstream.

Another view of this property is that exactly the same constraint on the program states in Program 1.6.1.2 can be affected by placing constraint \( /\!\wp(S_1, C) \) before \( S_1 \) or constraint \( /\!C \) before \( S_2 \). To be more precise, \( S_1;/\!C;S_2 \Leftrightarrow /\!\wp(S_1, C);S_1;S_2 \) if the scope of \( /\!C \) is not terminated by \( S_1 \). In general, this relationship can be stated as follows.

**Theorem 1.6.2.12:** \( S;/\!R \Leftrightarrow /\!Q;S \) if \( Q \equiv \wp(S, R) \).

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

On the other hand, given a program of the form \( /\!Q;S \), the same relation can be applied in other way to move the constraint downstream. In that case, what we need to do is to find a predicate \( R \) such that \( \wp(S, R) \equiv Q \). As discussed in [5], this can be done by letting \( R \equiv \wp(S^{-1}, Q) \), where \( S^{-1} \) is a sequence of statements to be derived from \( S \) and \( Q \). By replacing \( R \) in \( \wp(S, R) \equiv Q \), we obtain \( \wp(S;S^{-1}, Q) \equiv Q \).
Note that, 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. The following examples should be helpful in visualizing how to find $S^{-1}$ for given $Q$ and $S$.

(a) $Q: x \leq 100$
$S: x := x + 10$
$S^{-1}: x := x - 10$
$wp(S^{-1}, Q): x \leq 110$

(b) $Q: x > c$
$S: x := a$
$S^{-1}: \text{(does not exist)}$
$wp(S^{-1}, Q): \text{?}$

(c) $Q: x < c$
$S: y := y - 1; z := x + y$
$S^{-1}: \text{null statement, or } "\text{skip}"$
$wp(S^{-1}, Q): x < c$

(d) $Q: x > a$
$S: y := y + 2; w := w + 1; x := x + y$
$S^{-1}: x := x - y$
$wp(S^{-1}, Q): x > a + y$

Note that $S^{-1}$ does not exist if $S$ assigns a constant value to a variable in $Q$ as exemplified by case (b) above. The problem of finding $S^{-1}$ becomes more involved whenever $S$ contains a conditional statement or repetitive statement.

From the above discussion we see that, for convenience in future applications, Theorem 1.6.2.12 can be alternatively stated as

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

Corollary 1.6.2.10 and Theorem 1.6.2.12 belong to the second category of equivalence relations that one can use to combine and simplify the constraints placed in a program to make it more readable. For instance, by moving all state constraints in Program 1.6.2.11 to the top, it can be reduced to the one listed below:

```c
int atoi(char s[])
{
    int i, n, sign;
    \!
    !(isspace(s[0])) && !(s[0] == '-');
    \!
    !(s[0] == '+' || s[0] == '-');
    \!
    !isdigit(s[0]));
    \!
    !isdigit(s[1]));
```
One reason why the state constraints in a program can be simplified is that some state constraints are implied by the others, and thus can be eliminated. To be more specific, if two state constraints $C_1$ and $C_2$ are such that $C_1 \supset C_2$ then $C_2$ can be discarded because $C_1 \land C_2 \equiv C_1$. For example, the first two state constraints in the above program are implied by the third. Hence the above program can be simplified to

Program 1.6.2.13:

```c
int atoi(char s[])
{
    int i, n, sign;
    if (isdigit(s[0])) && !(isdigit(s[1]));
    i = 0;
    sign = 1;
    n = 0;
    n = 10 * n + (s[i] - '0');
    i = i + 1;
    return sign * n;
}
```

Some state constraints may be eliminated in the simplification process because it is always true due to computation performed by the program. For example, the following program contains such a constraint.

```c
x := 0; ↔ x := 0; ↔ /\ 0 + 1 <> 0; ↔ x := 0;
y := x + 1; ↔ /\x + 1 <> 0; ↔ x := 0; y := x + 1;
/\y <> 0; y := x + 1; ↔ y := x + 1;
```

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 $S_1; /\!\!\!\!\!\!\!\!\!/C; S_2 \leftrightarrow S_1; S_2$.

The properties and possible exploitation of tautological constraints will be discussed in a later section.

As one might have observed in previous examples, moving state constraints interspersed in the statements to the same point in control flow often leaves a long sequence of assignment statements in the program. These assignment statements may be combined and simplified by using the three equivalence relations presented in the following.

Corollary 1.6.2.15: $x:=E_1; x:=E_2 \leftrightarrow x:=(E_2)_{E_1 \rightarrow x}$
Here \((E_2)_{E_1 \rightarrow x}\) denotes the expression obtained from \(E_2\) by substituting \(E_1\) for every occurrence of \(x\) in \(E_2\). For example, applying this corollary to the third and the fourth assignment statements of Program 1.6.2.13 yields

```c
int atoi(char s[]) {
    int i, n, sign;
    /\ (isdigit(s[0])) && !isdigit(s[1]));
    i = 0;
    sign = 1;
    n = s[i] - '0';
    i = i + 1;
    return sign * n;
}
```

Although in general two assignment statements cannot be interchanged, an assignment statement may be moved downstream under certain circumstances. In particular,

**Corollary 1.6.2.16:** If \(x_2\) does not occur in \(E_1\) then

\[
x_1 := E_1; x_2 := E_2 \iff x_2 := (E_2)_{E_1 \rightarrow x_1}; x_1 := E_1.
\]

For example, by applying this relation to the above example it becomes

\[
\iff \ 
\begin{align*}
\text{int } atoi\text{char s[])} \{
    \text{int i, n, sign;}
    /\ (isdigit(s[0])) \&\! (isdigit(s[1]));
    i = 0;
    sign = 1;
    n = s[i] - '0';
    i = i + 1;
    return sign * n;
\}
\end{align*}
\]

\[
\iff \ 
\begin{align*}
\text{int } atoi\textchar s[]) \{
    \text{int i, n, sign;}
    /\ (isdigit(s[0])) \&\! (isdigit(s[1]));
    i = 0;
    n = s[i] - '0';
    sign = 1;
    i = i + 1;
    return sign * n;
\}
\end{align*}
\]

\[
\iff \ 
\begin{align*}
\text{int } atoi\textchar s[]) \{
    \text{int i, n, sign;}
    /\ (isdigit(s[0])) \&\! (isdigit(s[1]));
    i = 0;
    n = s[i] - '0';
    i = i + 1;
    sign = 1;
\}
\end{align*}
\]
return sign * n;
}

The purpose of an assignment statement is to assign a value to a variable so that it can be used in some statements downstream. Now if the above rule is used to move an assignment statement downstream past all statements in which the assigned value is used, the statement becomes redundant and thus can be deleted.

**Definition 1.6.2.17:** A statement in a program is said to be *redundant* if its sole purpose is to define the value of a data structure, and this particular value is not used anywhere in the program. Obviously, a redundant statement can be removed without changing the computation performed by the program.

**Corollary 1.6.2.18:** If \( x_1 := E_1; x_2 := E_2 \) is a sequence of two assignment statements such that, by interchanging these two statements, \( x_1 := E_1 \) becomes redundant, then \( x_1 := E_1; x_2 := E_2 \Leftrightarrow x_2 := (E_2)_{E_1} \rightarrow x_1 \).

For example, by interchanging the last two statements in the above program, the statement \( \text{sign} := 1 \) becomes redundant, and therefore Corollary 1.6.2.18 can be applied to simplify the program to

\[
\Rightarrow \quad \text{int atoi(char s[])}
\begin{array}{l}
\{ \text{int i, n, sign;)
\end{array}
\begin{array}{l}
\{ \text{\( \text{\( (\text{isdigit(s[0])} \&\& \neg (\text{isdigit(s[1]));} \\}
\begin{array}{l}
i = 0;
\end{array}
\begin{array}{l}
n = s[i] - '0';
\end{array}
\begin{array}{l}
i = i + 1;
\end{array}
\begin{array}{l}
\text{return n;)
\end{array}
\end{array}
\end{array}
\end{array}
\]

In general, Corollary 1.6.2.18 becomes applicable when \( x_2 := E_2 \) is the last statement to make use of definition provided by \( x_1 := E_1 \). Corollaries 1.6.2.16 and 1.6.2.18 can be used to reduce the number of assignment statements in a program, and thus the number of steps involved in computation. The end result is often a simpler and more understandable program. For instance, the above program can be further simplified to the one listed below:

\[
\Rightarrow \quad \text{int atoi(char s[])}
\begin{array}{l}
\{ \text{int i, n, sign;)
\end{array}
\begin{array}{l}
\{ \text{\( \text{\( (\text{isdigit(s[0])} \&\& \neg (\text{isdigit(s[1]));} \\}
\begin{array}{l}
i = 0;
\end{array}
\begin{array}{l}
n = s[0] - '0';
\end{array}
\begin{array}{l}
i = i + 1;
\end{array}
\begin{array}{l}
\text{return n;)
\end{array}
\end{array}
\end{array}
\end{array}
\]

\[
\Rightarrow \quad \text{int atoi(char s[])}
\begin{array}{l}
\{ \text{int i, n, sign;)
\end{array}
\begin{array}{l}
\{ \text{\( \text{\( (\text{isdigit(s[0])} \&\& \neg (\text{isdigit(s[1]));} \\}
\begin{array}{l}
i = 0;
\end{array}
\begin{array}{l}
n = s[0] - '0';
\end{array}
\begin{array}{l}
i = i + 1;
\end{array}
\begin{array}{l}
\text{return n;)
\end{array}
\end{array}
\end{array}
\end{array}
\]

\[
\Rightarrow \quad \text{int atoi(char s[])}
\begin{array}{l}
\{ \text{int i, n, sign;)
\end{array}
\begin{array}{l}
\{ \text{\( \text{\( (\text{isdigit(s[0])} \&\& \neg (\text{isdigit(s[1]));} \\}
\begin{array}{l}
i = 0;
\end{array}
\begin{array}{l}
n = s[0] - '0';
\end{array}
\begin{array}{l}
i = i + 1;
\end{array}
\begin{array}{l}
\text{return n;)
\end{array}
\end{array}
\end{array}
\end{array}
\]

Program Analysis and Testing 1-36 © J. C. Huang 2004
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. As usual, a set of n programs is denoted by \{P_1, P_2, ..., 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

\begin{align*}
\text{Axiom: 1.6.3.1: } \wp(\{P_1, P_2, ..., P_n\}, R) & \equiv \wp(P_1, R) \lor \wp(P_2, R) \lor ... \lor \wp(P_n, R).
\end{align*}

The choice of this particular semantics will be explained in detail at the end of this section. A program set so defined has all properties commonly found in an ordinary set. For instance, since the logical operation of disjunction is commutative, a direct consequence of Axiom 1.6.3.1 is that

\begin{align*}
\text{Corollary 1.6.3.2: } \text{The ordering of elements in a program set is immaterial, i.e.,} \\
\{P_1, P_2\} & \Leftrightarrow \{P_2, P_1\}.
\end{align*}

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

\begin{align*}
\text{Corollary 1.6.3.3: } P & \Leftrightarrow \{P\} \Leftrightarrow \{P, P\} \text{ for any program } P.
\end{align*}

In words, a set is unchanged by listing any of its elements more than once. A program set can be used just like a block of statements in program composition. Concatenation of program sets is defined similarly as concatenation of two ordinary sets. In particular,
**Corollary 1.6.3.4:** For any programs P, P₁, and P₂,
(a) \( P;\{P₁, P₂\} \Leftrightarrow \{P;P₁, P;P₂\} \)
(b) \( \{P₁, P₂\};P \Leftrightarrow \{P₁;P, P₂;P\} \)

Given below are a number of useful relations that can be derived from the semantics of a program set.

**Corollary 1.6.3.5:** If \( P \Leftrightarrow P' \) then \( \{P\} \Leftrightarrow \{P, P'\} \).

**Corollary 1.6.3.6:** If \( P \Leftrightarrow P₁ \) and \( P \Leftrightarrow P₂ \) then \( P \Leftrightarrow \{P₁, P₂\} \).

**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.

**Definition 1.6.3.8:** Two programs /\C₁;P₁ and /\C₂;P₂ are said to be *equivalently constrained* if and only if \( P₁ \) and \( P₂ \) are both unconstrained and \( C₁ \equiv C₂ \).

**Theorem 1.6.3.9:** If \( P₁ \Rightarrow P₂ \), and if \( P₁ \) and \( P₂ \) are equivalently constrained, then \( P₁ \Leftrightarrow P₂ \).

The following relations are useful in working with sets of constrained subprograms.

**Corollary 1.6.3.10:** /\C₁ ∨ C₂;P \Leftrightarrow \{/\C₁;P, /\C₂;P\}

**Corollary 1.6.3.11:** If \( C₁, C₂, ..., Cₙ \) are \( n \) constraints such that \( C₁ ∨ C₂ ∨ ... ∨ Cₙ \equiv T \) then
\( P \Leftrightarrow /\C₁ ∨ C₂ ∨ ... ∨ Cₙ;P \Leftrightarrow \{/\C₁;P, /\C₂;P, ..., /\Cₙ;P\} \)

The last corollary serves as the basis for decomposing a program into an equivalent set of subprograms. To distinguish this process from the traditional method of decomposing a program into procedures and functions, 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. The concept of pathwise decomposition will be discussed in more detail in the next section.

Since braces and comma may have a different meaning in a real programming language, and since in practice program statements are written line by line from top to bottom, a program set of the form \( \{P₁, P₂, ..., Pₙ\} \) may be alternatively written as

\[
\{\{\{P₁, P₂, ..., Pₙ\}\}\}\n\]
where triple braces and triple commas are used instead to avoid any possible confusion.

To clarify the concept and notations just introduced, consider Program 1.6.3.12 listed below, which can be decomposed into a set of two subprograms:

Program 1.6.3.12:

```c
main()
{
    int x, y, z;
    scanf("%d", &x);
    y = 1;
    while (x <= 100) {
        x = x + 11;
        y = y + 1;
    }
    while (y != 1) {
        x = x - 10;
        y = y - 1;
        while (x <= 100) {
            x = x + 11;
            y = y + 1;
        }
    }
    z = x - 10;
    printf("z = %d\n", z);
}

⇔ main()
{
    int x, y, z;
    scanf("%d", &x);
    y = 1;
    while (x <= 100) {
        x = x + 11;
        y = y + 1;
    }
    while (y != 1) {
        x = x - 10;
        y = y - 1;
        while (x <= 100) {
            x = x + 11;
            y = y + 1;
        }
    }
    z = x - 10;
    printf("z = %d\n", z);
}

⇔ main()
{
    int x, y, z;
    scanf("%d", &x);
    y = 1;
    while (x <= 100) {
        x = x + 11;
        y = y + 1;
    }
    while (y != 1) {
        x = x - 10;
        y = y - 1;
        while (x <= 100) {
            x = x + 11;
            y = y + 1;
        }
    }
    z = x - 10;
    printf("z = %d\n", z);
}
```

Program Analysis and Testing 1-39 © J. C. Huang 2004
```
x = x + 11;
y = y + 1;
}
'''
\( \text{while } (x > 100) \) {
  x = x - 10;
y = y - 1;
  \( \text{while } (x \leq 100) \) {
x = x + 11;
y = y + 1;
}
}
z = x - 10;
printf("z = %d\n", z);
}
```

⇔

```
Program 1.6.3.13:

```

```

main()
{
  int x, y, z;
  scanf("%d", &x);
y = 1;
  \( \text{while } (x \leq 100) \) {
x = x + 11;
y = y + 1;
  }
  \( \text{while } (y != 1) \) {
x = x - 10;
y = y - 1;
  \( \text{while } (x \leq 100) \) {
x = x + 11;
y = y + 1;
  }
}
z = x - 10;
printf("z = %d\n", z);
}
```

```
Program Analysis and Testing 1-40 © J. C. Huang 2004

by 1.6.2.5
```
There are two more corollaries that we need for discussion in a later section.

**Corollary 1.6.3.14**: \( / C\{P_1, P_2\} \Leftrightarrow \{/ \setminus C;P_1, / \setminus C;P_2\} \)

**Corollary 1.6.3.15**: \( \{P_1; / \setminus C_1, P_2; / \setminus C_2\} \Leftrightarrow \{P_1; / \setminus C_1, P_2; / \setminus C_2\} / \setminus C_1 \lor C_2 \)

### 1.6.4 Pathwise Decomposition

In this section we shall discuss the concept of pathwise decomposition in more detail. 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). A cascade of two edges represents a construct formed by a sequence of two components as depicted below.

![Diagram of two edges connected in series](#)

\( S_1 ; S_2 \)

Two edges are connected in parallel to represent the program set consisting of the associated components as shown below.

![Diagram of two edges connected in parallel](#)

\( \{S_1, S_2\} \)

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

Although Corollary 1.6.3.11 allows us to decompose a program in infinitely many ways, there is one decomposition scheme among them that is of practical importance, i.e., to decompose a program along the branches in its control flow. This can be done by decomposing every conditional statement in the program into an equivalent program set such that every element in the program set is reduced to a non-conditional statement. For instance, if there is a conditional statement of the form `if B then S_1 else S_2`, we shall use the relation `if B then S_1 else S_2 \Leftrightarrow \{/ \setminus B; if B then S_1 else S_2, / \setminus \neg B; if B then S_1 else S_2\} \Leftrightarrow \{/ \setminus B;S_1, / \setminus \neg B;S_2\}` to decompose it into two subprograms, and replace its graph representation by a pair of edges connected in parallel as depicted below.
If the program contains a loop construct of the form `while B do S`, we will do the same, viz., to use the relation

```
while B do S ⇔ \{/\B;while B do S, /\¬B;while B do S\}, ⇔ \{/\B;S;while B do S, /\¬B\}
```

to decompose it into two subprograms, which can be graphically represented as follows.

```
/\¬B;

/\B;S
```

The conditional statement, `while B do S`, associated with the left branch can be recursively represented by the same graphic structure, and thus its presence in the graphic representation can be removed by letting the left edge point to the starting node as shown below.

```
/\¬B;

/\B;S;while B do S
```

Note: Another commonly used conditional statement, `repeat S until ¬B`, can be treated like a `while` statement because `repeat S until ¬B ⇔ S; while B do S`.

Given a program graph, we can replace any conditional statement in it with another construct that contains constrained non-conditional statements as described above. When all conditional statements are replaced, it will have the property that each path from the entry to the exit represents a potential execution path in the program. At that point, the program graph becomes a control-flow graph because that property is the property of a conventionally defined control-flow graph.

**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.

For instance,

```
/\ i < 10;
/\ a[i] < 0;
```
sum = sum - a[i];
i = i + 1;
/\ !(i < 10);

is a trace subprogram of the following C program:

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

which is represented by the following control-flow diagram:

However, neither

```c
/\ i < 10;
/\ a[i] < 0;
sum = sum - a[i];
/\ i > 5
i := i + 1;
/\ !(i < 10);
```

nor

```c
/\ i < 10;
if a[i] < 0
    sum := sum - a[i];
else
    sum := sum + a[i];
i := i + 1;
/\ !(i < 10);
```

is a trace subprogram of that program because they do not represent any path in the control graph. The former represents only a portion of an execution path while the latter contains more than one execution path.

Note that any trace subprogram is a canonical subprogram (cf. Def. 1.6.2.9), or can be rewritten into one. This is so because, by definition, a trace subprogram may contain only one execution path. Such a program can be written without the use of any conditional statement. On the other hand, a canonical subprogram of a program is not necessarily a trace subprogram of that program because it may corresponds to only a portion of an execution path.
For any particular execution path in a program, the associated trace subprogram can be constructed directly from its source code by constraining every conditional statement in the program toward the direction to which the execution is expected to proceed and then simplify the program. For a program with loop constructs, this process may have to be repeated until there is no more conditional statement in the program.

For example, the trace subprogram given above can be constructed from its source text as shown below.

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

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

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

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

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

⇔ \( i < 10; \)
\( i < 10; \)
\( i < 10; \)
\( \text{if } a[i] < 0
    \text{sum} = \text{sum} - a[i];
\text{else}
    \text{sum} = \text{sum} + a[i];
```

Program Analysis and Testing 1-44 © J. C. Huang 2004
Alternatively, the trace subprogram of an execution path can be obtained from its symbolic trace. As defined in [HUAN80], the symbolic trace of an execution path is a linear list of all statements and branch predicates that occur on the execution path. A symbolic trace is transformed into a trace subprogram by rewriting every branch predicate therein as a state constraint. The trace subprogram of any execution path, like its symbolic trace, therefore, can be generated automatically through program instrumentation as described in [HUAN80].

Trace subprograms are of particular interest because, with proper tools, they can be produced and simplified readily. Given a program that is difficult to understand because of complex logical structure, one can often reduce the difficulty by understanding the program in terms of its simplified trace subprograms.

For example, consider the C program listed below:

Program 1.6.4.2:

```c
main () {
    int i, j, k, match;
    scanf("%d %d %d", &i, &j, &k);
    printf("%d %d %d\n", i, j, k);
    if (i <= 0 || j <= 0 || k <= 0) goto L500;
    match = 0;
    if (i != j) goto L10;
    match = match + 1;
    L10: if (i != k) goto L20;
        match = match + 2;
    L20: if (j != k) goto L30;
        match = match + 3;
    L30: if (match != 0) goto L100;
        if (i+j <= k) goto L500;
        if (j+k <= i) goto L500;
        if (i+k <= j) goto L500;
        match = 1;
        goto L999;
    L100: if (match != 1) goto L200;
        if (i+j <= k) goto L500;
    L110: match = 2;
        goto L999;
    L200: if (match != 2) goto L300;
        if (i+k <= j) goto L500;
        goto L110;
    L300: if (match != 3) goto L400;
        if (j+k <= i) goto L500;
        goto L110;
    L400: match = 3;
        goto L999;
    L500: match = 4;
    L999: printf("%d\n", match);
}
```

The main body of this program can be decomposed into a set of 12 trace subprograms. By rewriting the program in terms of the simplified subprograms, it becomes

Program 1.6.4.3:
main()
{
    int i, j, k, match;

    scanf("%d %d %d", &i, &j, &k);
    printf("%d %d %d\n", i, j, k);
    {{
        // (i <= 0) || (j <= 0) || (k <= 0)
        match = 4;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j > k) && (j + k > i) && (i + k <= j)
        // (i != j) && (i != k) && (j != k)
        match = 4;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j > k) && (j + k <= i)
        // (i != j) && (i != k) && (j != k)
        match = 4;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j <= k)
        // (i != j) && (i != k) && (j != k)
        match = 4;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j <= k)
        // (i != j) && (i != k) && (j != k)
        match = 4;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i == j) && (i == k) && (j == k)
        match = 3;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j > k)
        // (i != j) && (i != k) && (j != k)
        match = 2;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (j + k > i)
        // (i != j) && (j == k)
        match = 2;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + k > j)
        // (i != j) && (i == k)
        match = 2;
        
        // (i > 0) && (j > 0) && (k > 0)
        // (i + j > k) && (j + k > i) && (i + k > j)
        match = 1;
    }}
}
printf("%d\n", match);
}

Obviously, Program 1.6.4.3 is more understandable than Program 1.6.4.2 because its logical structure is much simpler.

Let \( f \) be a function defined on a set \( D \). Usually a program \( p \) is designed to implement \( f \) by decomposing it into a set of \( n \) subfunctions, i.e., \( f = \{ f_1, f_2, \ldots, f_n \} \) such that \( D = D_1 \cup D_2 \cup \ldots \cup D_n \) and \( f_i \) is \( f \) restricted to \( D_i \), for all \( 1 \leq i \leq n \). If the program is correctly constructed, it must be composed of \( n \) trace subprograms, i.e., \( p = \{ p_1, p_2, \ldots, p_n \} \) such that each \( p_i \) implements subfunction \( f_i \). The state constraints in \( p_i \) jointly define the input subdomain \( D_i \), and the statements in \( p_i \) describe how \( f_i \) is to be computed.

Thus, the number of execution paths in a program is equal to the number of input subdomains partitioned by the program. It does not necessarily grow exponentially with the number of conditional statements in the program as commonly believed. For instance, there are 14 "if" statements in Program 1.6.4.2. Theoretically, there could be as many as \( 2^{14} = 16384 \) execution paths, while in fact there are only 12.

The fact is that most real programs have relatively few feasible execution paths unless a loop construct is involved.