A brief introduction to the derivation of programs

Juris Reinfelds

University of Wollongong
A BRIEF INTRODUCTION TO THE DERIVATION OF
PROGRAMS

Juris Reinfelds

DEPARTMENT OF COMPUTING SCIENCE
UNIVERSITY OF WOLLONGONG

Abstract

These notes introduce a method of program derivation. We start by inventing a skeleton structure of the program and by writing down a statement of what conditions should program variables satisfy upon termination of the program. We then derive the rest of the program by algebraic techniques and logical reasoning. We use the methods of Dijkstra and Gries but introduce only a minimum of formalism and theory. Emphasis is on practical application of these methods to classes of problems where success is most likely. The theoretical basis for the methods can be established in later courses.
0.0 INTRODUCTION

The first book which employed the help of Boolean algebra and predicate calculus to derive correct programs was "A Discipline of Programming" by Dijkstra [1976]. Each chapter of this book derives a beautiful and remarkably simple program for problems which range from easy to very difficult. The derivation of most of these programs requires creative ability of a very high order. The second book is by David Gries [1981]. Gries shows that for a large class of problems the derivation of the program follows well defined algebraic steps and procedures.

These notes introduce the absolute minimum of Boolean arithmetic, a few notions of Boolean algebra and predicate calculus and some of the concepts required for the derivation of programs. Examples are provided in problem areas where the program derivation follows simple algebraic procedures. Theoretical results to justify the formalism and a thorough introduction to Boolean and predicate calculus are left to later studies in computing science or mathematical logic.

1.0 INTRODUCTION TO BOOLEAN ARITHMETIC

This chapter provides a very simple introduction to the concepts and operations of Boolean Arithmetic. We will introduce only those concepts and mechanisms which will be essential for the understanding of the subsequent chapters on program derivation. The use of Boolean arithmetic in mathematical logic is acknowledged but not explored much further. The use of Boolean arithmetic in philosophy is regarded as a potential source of great confusion.

1.1 Boolean Arithmetic

Boolean Arithmetic defines a set of operations on a binary number system which admits only two numbers. By contrast, ordinary arithmetic defines a set of operations such as addition, subtraction, multiplication and division on a binary number system where an infinite number of numbers is permitted.
The operations of Boolean Arithmetic are self contained or as we say mathematically they form a closed system defined so that the result of any legitimate operation is always one or the other of the two permitted numbers, never anything else. By contrast, addition, subtraction and multiplications are closed operations in ordinary arithmetic but division is not closed because division by zero yields no valid number as a result.

Obviously, the operations of ordinary arithmetic will not be very useful in a number system that admits only a finite set of numbers. We will define new operations by giving the rules as to what result is obtained by any combination of the argument values. Since only two distinct numbers are possible, there are only two distinct cases for a unary operator and four cases for a binary operator. Such a definition would obviously not be practical in ordinary arithmetic!

When some of these new operations will resemble arithmetical operations we will point out the similarities. Such comparisons help to assimilate new concepts, but they can become misleading when taken too far. We will use the conventional names for the operations. These are common words in the English language and because Boolean arithmetic is used to grapple with real world concepts in philosophy, a lot of unnecessary confusion is caused by a misplaced insistence to interpret the underlying meaning of the word that is used to denote an operation in Boolean arithmetic.

Since arithmetic is concerned with structure and not with meaning, there is no deeper meaning to the arithmetical operation $2 + 2$ and similarly there is no deeper philosophical meaning to any operation of Boolean arithmetic. Budding philosophers who are too tempted by the common meaning of the words should rename the operations using words that do not have any common language meaning. The same arguments apply to the names given to the two numbers permitted by Boolean arithmetic. Because of the unfortunate connection with philosophy they are often denoted by the two words.

true false

The notion of truth, eternal or otherwise, belongs to philosophy. In mathematics or program derivation it confuses the issue with concepts which do
not apply to the manipulation of symbols or text or program fragments. To avoid such philosophical connotations engineers often use the symbols

1 0

but these carry with them notions of conventional arithmetic which are sometimes as confusing as the notions of philosophy. We will use the words *true* and *false* without any regard for their common language meaning. If this is too hard then the reader should choose words with less specific meaning such as perhaps *blue* and *brown* or *alpha* and *beta* or *Jack* and *Jill*.

### 1.2 NOT

Logical negation *not* is a unary operator often denoted by a symbol

\[ \neg \]

which denotes the inversion of the value of the argument. There are only two possibilities

\[ \neg \text{true} \text{ yields } \text{false} \]
\[ \neg \text{false} \text{ yields } \text{true} \]

### 1.3 AND

Logical conjunction *and* is a binary operator often denoted by a symbol that looks like a capital A without the crossbar

\[ \wedge \]

There are four possibilities for the result of such an operation and these are defined to be

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>true</th>
<th>yields</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>\wedge</td>
<td>false</td>
<td>yields</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>\wedge</td>
<td>true</td>
<td>yields</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>\wedge</td>
<td>false</td>
<td>yields</td>
<td>false</td>
</tr>
</tbody>
</table>
1.4 OR

Logical disjunction or is a binary operator often denoted by a symbol that is an upside down version of the symbol for and

\[ \lor \]

There are four possibilities for the result of such an operation and these are defined to be

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>yields</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>V</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>V</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>V</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>V</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

In everyday language this version of "or" is described as "this or that or both". The operation which yields false when both arguments are true is called exclusive or.

1.5 Exclusive OR

The exclusive or is used mainly by electrical engineers and computer scientists. It does not have a universally accepted symbol so the letter xor are often used. The four possible results of this operation are defined to be

<table>
<thead>
<tr>
<th></th>
<th>xor</th>
<th>yields</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>xor</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>xor</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>xor</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>xor</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

1.6 Implies

The unfortunate choice of the word "implies" as the name of an otherwise well defined and innocuous operation has kept generations of philosophers in endless debates causing enormous amounts of unnecessary confusion in the minds of students who have to master the rules of Boolean arithmetic. There is a well established symbol for this operation. It is a right pointing arrow

\[ \Rightarrow \]
The four possible results of this operation are defined to be:

\[
\begin{align*}
    \text{true} & \implies \text{true} \quad \text{yields true} \\
    \text{true} & \implies \text{false} \quad \text{yields false} \\
    \text{false} & \implies \text{true} \quad \text{yields true} \\
    \text{false} & \implies \text{false} \quad \text{yields true}
\end{align*}
\]

Those readers who are too tempted to grapple with the philosophical explanation of how a false statement can imply anything, should observe that given two Boolean variables \(p\) and \(q\), the result of

\[p \implies q\]

is equivalent to the result of \(\neg p \lor q\)

\[\neg p \lor q\]

and instead of the verbal description "\(p\) implies \(q\)" they should consistently say and think "\(\text{nor } p \text{ or } q\)"

### 1.7 Equivalence

Boolean arithmetic forms the basis of Boolean algebra. Boolean algebra concerns the manipulation of Boolean expressions, mostly for simplification or to establish whether two expressions are equivalent or not.

Two Boolean expressions are defined to be equivalent if they produce identical results for every combination of truth values of all variables contained in the expressions. The number of different possibilities for \(n\) variables is \(2^n\) so that only the simplest of expressions can be explored by direct substitution of truth values into all variables. Successful program derivation requires substantial skills in Boolean algebra but it is beyond the scope of this introduction to provide these. For further reading see Gries [1981].

Dijkstra has introduced equivalence into Boolean arithmetic as a binary operation with the symbol

\[\equiv\]
and with the four possible results of the operation defined as

\[
\begin{array}{ccc}
\text{true} & \equiv & \text{true} & \text{yields} & \text{true} \\
\text{true} & \equiv & \text{false} & \text{yields} & \text{false} \\
\text{false} & \equiv & \text{true} & \text{yields} & \text{false} \\
\text{false} & \equiv & \text{false} & \text{yields} & \text{true}
\end{array}
\]

To familiarize ourselves with this operation let us establish whether we correctly stated in the previous section that for any pair of Boolean variables \( p \) and \( q \) the expression

\[
(p \implies q) \quad (\neg p \lor q)
\]

always yields the value \text{true} regardless of the values of \( p \) or \( q \).

The direct way to establish the result of this operation is to draw up a table that is called a truth table. On the left are the variables of the expression, in the middle are the values of the various sub-expressions as needed and on the right is the value of the whole expression. Since there are two variables, the truth table will have four lines.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p \implies q )</th>
<th>( \neg p )</th>
<th>( \neg p \lor q )</th>
<th>((p \implies q) \equiv (\neg p \lor q))</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

1.8 Universal Quantifiers

Sometimes it is useful to make a statement about all possible values of an arithmetic or Boolean variable or expression, for example one such statement could be

"the variable ceiling contains a value that exceeds every value in the array \( B \)"

Let us rewrite the above statement so that it becomes easier to introduce a formal expression to denote it:
"for all values of the subscript i such that i is at least zero and less than N, we have ceiling > B[i]."

This sentence is more concisely expressed by the more formal expression

\[(Ai: 0 \leq i < N : \text{ceiling} > B[i])\]

for all i such that we have

This expression has a Boolean result. In a given situation with well defined arithmetic variables \(\text{ceiling}, \text{i}\) and \(B[i]\) it yields true or false in an obvious way. Such Boolean expressions are called predicates and this particular predicate is called a universal quantification over all values of the variable i. The symbol A is called the universal quantifier. This notation is due to Dijkstra. Mathematicians use a slightly different notation to denote universal quantification.

1.9 Existential Quantifiers

Sometimes it is useful to make a statement about at least one value of an arithmetical or Boolean variable or expression. One such statement could be

"the value contained in variable x occurs at least once in array B"

Once again the above sentence may be transliterated into the more formal expression

\[(Ei : 0 \leq i < N : x = B[i])\]

there exists at least one i such that we have

This expression has a Boolean result. For well defined values of x and all i and \(B[i]\) the expression yields true or false in an obvious way. Such Boolean expressions are called predicates and this particular predicate is called an existential quantification over all values of the variable i. The symbol E is called the existential quantifier. This notation is due to Dijkstra. Mathematicians use a slightly different notation to denote existential qualification.
1.10 A Predicate

A predicate is defined as a Boolean expression which may contain Boolean variables, constants and operations including universal and existential quantification. It is beyond the scope of these notes to enter into a discussion of predicate calculus. This is left to subsequent studies of program derivation or of mathematical logic.

2.0 BASIC CONCEPTS, TERMS AND DEFINITIONS

Computing Science is often called informatics because it concerns the transmission, manipulation and storage of information. Numerical information is a small subset of all information that is handled by computers. Most information that mankind possesses is stored in alphabetical form.

The purpose of information is to preserve and communicate ideas through space (from one place to another) or through time (for later reference). Information may be transmitted by any property of the physical world that can be detected and reproduced reliably. Let us call any such property a mark. A mark represents information; it does not constitute the information it carries. A mark is no longer useful if the creator of the mark and the user of the mark cannot agree on what idea the mark represents. Rules concerning the appearance of marks are called syntax rules. Rules concerning the meaning of marks are called semantic rules. Typical marks are: ink on paper, magnetized domains on tape, notched sticks, notched grooves in a long spiral on a gramaphone record.

The computer is a device that manipulates marks. A computer can manipulate marks at electronic speeds (less than $10^{-6}$ sec/mark). A computation is a sequence of operations on marks. A program is the prescription for a computation. A program is a sequence of marks, such as symbols or words or numbers, written, punched or typed by a programmer. A program is a static object. A computation involves the manipulation of marks according to a program. A computation is a dynamic object. A computation is sometimes called a process.
A computation is a sequence of simple actions called operations. Operations are defined and provided by the engineers who designed and built the computer. The sequences of operations that are possible in a computation are prescribed in the program by instructions sometimes called statements or program steps.

Every computation that is to be executed on a computer requires a program that is written according to the rules of some programming language. Otherwise there will be no way to translate the program steps into operations. Programs which translate program steps into operation sequences are called assemblers, compilers or interpreters.

A program which represents the solution of a problem but which is not designed to run on a particular computer need not be written in any particular programming language. Such a program is called an algorithm. It is often easier to derive the solution of a problem in a notation which does not adhere strictly to the rules of any one programming language. When the correctness of the algorithm is established, the translation of the program into a specific programming language is in most cases easy.

3.0 PROGRAMMING CONCEPTS

Almost all of our currently most widely used programming languages are built around the concept of assignment. An assignment statement changes the value of a variable. If we regard the input of data into the computer as a special kind of assignment then assignment is the only mechanism which changes the value of a variable.

The state of a computation is defined by taking a snapshot of the values of all variables which are involved in the computation. Each variable is said to define a dimension of the state space. The state space of a computation which involves n variables is n-dimensional. An assignment changes the value of a variable, hence an assignment changes the state of the computation.
The specification of a problem defines initial values and properties of initial values for all variables which are not initialized by the computation itself. The set of these values and properties defines the initial state of the computation. The specification of a problem also defines the desired properties of the results of a computation. The set of these values and properties defines the final state of the computation.

A computation may be seen as a path through state space from an initial state to a final state. The path through state space is achieved by the execution of a suitable sequence of assignment statements. The programmer's task is to discover such a sequence. Normally it is not possible to solve a problem by writing a simple sequence of assignment statements. Algorithms often require dynamic selection of which assignments are executed and which are not (conditional statements) and the repetition of a group of statements (loop). Statements which alter the sequence of execution of assignment statements are called control statements.

Traditionally the sequence of assignment statements and the necessary control structures have been discovered by intuition, guess or other intensive exertion of creative capabilities. The purpose of these notes is to show that for a large class of problems correct programs can be derived by algebraic manipulation following a small number of rules. Intuitive and creative demands are thereby reduced so that they can be applied to more challenging problems later.

4.0 ALGORITHMS AND PROGRAMS

A program written in no particular programming language is often called an algorithm. This permits the programmer to separate the concerns of problem solving from the concerns of syntax and semantics of a particular programming language.

The main problem with this approach is that the way in which an algorithm may be specified is too vague. Pseudo English or pseudo Something is not well defined. Novice programmers are never certain what language forms or constructs may or may not be used and most such algorithm descriptions
eventually look like one's favourite programming language with incorrect punctuation and misnamed keywords.

One can certainly start with a program description in English but such a description is not always easily translatable into programming languages. One way to achieve a useful notation is by abstracting the most fundamental, most generic programming constructs from our favourite programming languages and construct programs (algorithms) using these constructs.

The best such mini-language was proposed some years ago by Dijkstra [1976] and it has formed the basis for the derivation of programs in the discipline that is now known as the Science of Programming. The next sections will introduce the most important concepts of this notation. This section concludes with a description of a simple but adequate notation that is widely used to define programming languages.

4.0.1 BNF NOTATION

It is always difficult to use a language to make statements about itself. In English we use quotes such as for example in saying

*The sentence 'time flies' has two meanings*

we observe that the sentence between the quotes is not a part of the sentence outside the quotes. The sentence outside the quotes says something about the sentence inside.

The description of a programming language deals with definitions of language constructs, such as assignment statements or loops. It also deals with concepts which can be understood without further definitions such as arithmetic expressions or integer numbers. It uses symbols which appear in programming language statements such as := or IF as well as symbols which are used to describe the definition itself rather than the programming language such as the quotes in the above example from English. In programming language descriptions the names of definitions of programming language components and statements are placed between angular brackets. For example the symbol for the definition of a statement is written

<statement>
The names of concepts which are regarded as sufficiently well understood without further definitions are written as simple phrases such as

- arithmetic expression
- integer
- identifier

A definition of a component of a programming language is expressed in terms of names of other such definitions as well as in terms of phrases representing concepts which are understood without further definitions. For a well defined language all definitions should eventually lead to definitions which contain well understood symbols only. Such substitution sequences are often called "definition".

The place where the chain of definitions is broken by a concept that is accepted without further definitions depends upon the concepts that are accepted by those who participate in the discussion. One can be very formal and require definitions down to the level of digits and single characters while in a different situation one may accept identifiers or expressions or assignment statements as concepts that are well understood without further definitions.

Symbols which appear in programming languages are as varied and different as there are language designs. For example we can define a simple assignment statement as

\[ \text{identifier} := \text{expression} \]

where we assume that identifier and expression are well defined concepts which are separated by the symbol :=. More formally we can define a simple assignment statement as

\[ <\text{identifier}> := <\text{expression}> \]

where we are now obliged to give further definitions for identifier and expression.

Finally let us introduce three symbols that are used by definitions themselves. The first such symbol we have already met. It is the pair of angular brackets to denote the name of a definition.
The symbol ::= is used to separate the name of a definition from its explanation and another symbol | is used to denote alternatives. This gives us:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>pronounced as</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>&quot;is defined as&quot;</td>
<td>separates definition name from definition body</td>
</tr>
<tr>
<td></td>
<td>&quot;or&quot;</td>
<td>separates alternatives in the definition body</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>silent</td>
<td>indicates definition name</td>
</tr>
</tbody>
</table>

For example we can define a simple assignment statement as:

\[
\text{< simple assignment statement > ::= name := expression}
\]

To define a multiple assignment statement as described in Section 4.1 we have to define:

\[
\text{<name list> ::= name | name, <name list>}
\]
\[
\text{<expr list> ::= expr | expr, <expr list>}
\]

where we have defined a name list as a sequence of names separated by commas and an expression list as a sequence of expressions separated by commas. When the name of a definition appears on the righthand side of the definition, then the definition is called recursive. A properly constructed recursive definition will have at least one non-recursive alternative so that the definition chain can terminate. We can define the assignment statement as:

\[
\text{<assignment statement> ::= <name list> ::= <expr list>}
\]

This notation is called BNF (Backus Naur Form), named after John Backus, the originator of Fortran and Peter Naur who wrote the first concise programming language definition for Algol 60.

### 4.1 Assignments & Multiple Assignments

An assignment statement assigns the value of an expression to a variable (name). The variable represents this new value until another assignment changes it. Up to the first assignment the value of a variable is undefined. A variable can therefore participate in an expression only if it has already received a value in an assignment.
A multiple assignment simultaneously assigns the values of a sequence of expressions to a sequence of variables. The matching of variables to values occurs by position in the sequence in which they are written down. For example, the values of two variables \( x \) and \( y \) can be swapped without the need for an auxiliary variable by the multiple assignment statement

\[
x, y := y, x
\]

Swapping of values using a sequence of single assignments requires an auxiliary variable \( temp \). This is a typical example of an unnecessary complication that is necessitated by the inadequate design of a programming language:

\[
temp := x; \ x := y; \ y := temp
\]

Pathological cases such as the assignment of several values to one variable in a single multiple assignment statement as in

\[
x, x := 3, 5
\]

are not allowed.

4.2 Guarded Statements

The most fundamental way to dynamically alter the order of execution of a sequence of assignment statements is to introduce the concept of a guarded statement written as

\[
\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle
\]

which consists of a guard which is a Boolean expression which is separated from a statement by an arrow. The statement is executed if and only if the value of the guard expression yields \text{true}. If the value of the guard is \text{false}, the statement is not executed and the computation proceeds to the next statement of the program. The definition of the above informal guarded statement can be stated more formally as

\[
\langle \text{guarded statement} \rangle ::= \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle
\]

\[
\langle \text{guard} \rangle ::= \text{Boolean expression}
\]
While a sequence of statements in our mini/language may also be regarded as a single more complex statement. All these notions are succinctly summarized by the definition

\[
<\text{statement}> ::= <\text{assignment statement}> \\
| <\text{guarded statement}> \\
| <\text{statement}> ; <\text{statement}>
\]

Instead of using guarded statements directly, let us introduce two constructions which are fundamental to the control structures of most programming languages.

### 4.3 The IF Statement

The **IF-Statement** is a \(<\text{statement}>\) which has the following form.

\[
\text{if} \\
\text{guard 1 } \rightarrow \text{ statement 1} \\
\text{"fat bar"} \quad [\quad \text{guard 2 } \rightarrow \text{ statement 2} \\
\quad [\quad \text{guard n } \rightarrow \text{ statement n} \\
\text{fi}
\]

The **fat bar** \([\quad\text{]}\) is a symbol that appears in the programming language. It separates the guarded statements of an **If-statement** so that each of the statements (such as statement 1, statement 2...) may contain any number of guarded or unguarded statements or further IF-statements.

The **IF-statement** must contain at least one guarded statement. When the computation requires that an **IF-Statement** be executed, the computation may select any one of the guarded statements whose guard yields \textit{true} and execute its associated statement. Nothing whatever is assumed about the sequence in which the guards are examined. Not even random choice may be assumed. If several guards yield \textit{true} in an **IF-statement**, then at this stage of the computation any one of the corresponding statements will change the state of the computation on a proper path from initial state to the result (final state) and therefore any one of these statements may be executed.
It is an error if all guards are false. In such a case none of the statements contained in the IF-statement will contribute a change of state of the computation which will eventually lead to an acceptable final state. The IF-statement is more general than the if-then-else of most programming languages. An if-then-else statement such as for example

\[
\text{if } x \geq 0 \text{ then } y := \sqrt{x} \text{ else } y := \sqrt{-x} \\]

is expressed as an IF-statement by guarding the then-statement with the given condition and the else-statement with the logical complement of the given condition

\[
\text{if } \\
\begin{align*}
x \geq 0 & \rightarrow y := \sqrt{x} \\
[x] x < 0 & \rightarrow y := \sqrt{-x} \\
\end{align*}
\text{fi}
\]

4.4 A Statement That Does Nothing

The IF-statement requires that all options for progress be mentioned explicitly. Hence a statement that does nothing is needed to express a simple if-then statement which does not have an else part. Such a statement is called skip so that the if statement of conventional programming languages

\[
\text{if } x < 0 \text{ then } x := -x
\]

is expressed by the IF-statement

\[
\text{if } \\
\begin{align*}
x < 0 & \rightarrow x := -x \\
[x] x \geq 0 & \rightarrow \text{skip} \\
\end{align*}
\text{fi}
\]

4.5 The Loop

The repetitive statement (sometimes called the loop) has a form that is very similar to the IF-statement:
The repetitive statement must contain at least one guarded statement. Initially we will study loops with only one or at most two guarded statements but the definition of the repetitive statement permits any number of them.

When a computation commences the execution of a repetitive statement it checks to see if all guards yield false. If this is the case, no statements contained in the loop body are executed and execution continues with the next statement after the repetitive statement. If at least one guard yields true however, then any one of the statements which has a true guard is executed and the guards are checked again. The cycle is repeated until all guards yield false. As for the IF-statement, we are not allowed to make any assumption about the order in which guards are examined or which statements are selected for execution. All we know is that the statement belonging to one and only one true guard is executed at each iteration of the loop. When all guards yield false we say that the loop has terminated and execution continues with the next statement after the loop.

Note that a loop which initially has at least one true guard cannot terminate unless the statement of the true guard makes an assignment to at least one of the variables which occurs in the guard. Each cycle around the loop is called an iteration. A loop containing one guarded statement is equivalent to a while-do loop of Pascal

\[ \text{while } x < N \text{ do } x := x + 1 \]
describes the same program fragment as the repetitive statement

\[
\begin{align*}
\text{do} & \\
& \quad x < N \rightarrow x := x + 1 \\
\text{od}
\end{align*}
\]

In both cases the loop terminates when the guard becomes \textit{false}. The iterations of a loop can be viewed as the execution of a long sequence of assignment statements by an equivalent program that does not contain the loop. The text of the statements would repeat in regular groups but the values handled would change until all guards become \textit{false}. How many statements would be executed is determined by the state of the computation just before entry into the loop but this number is normally not known to the program or to the programmer.

Fortunately we can write correct programs without the need to know the exact number of iterations required to terminate a repetitive statement. It suffices to establish that termination can be achieved in a finite number of iterations and that each iteration reduces this number by at least one.

As for the execution of a program, the execution of a repetitive statement may also be viewed as a passage through state space where each iteration achieves at least one new state of the computation. On termination of the repetitive statement the state of the computation is such that all guards are \textit{false}. The guards are Boolean expressions and their condition (namely all \textit{false}) is characteristic of the state of the computation at the termination of the loop. We therefore say that the Boolean expression

\[
\text{all guards} = \text{false}
\]

guids to select that state of the computation in which the termination of the loop is possible. We say that a Boolean expression \textit{selects} those states of the computation for which the expression has the value \textit{true} and we often call this expression a \textit{predicate that selects a set of states}.

A remarkable discovery of Robert Floyd and E.W. Dijkstra was that every loop is characterised by a predicate which they called the loop invariant \textit{P}. This predicate selects the states of state space which are available to the
iterations of the loop. In Dijkstra's words the loop invariant "captures the essence of the loop".

Traditional program design proceeds as follows: first the purpose of a loop is described in English. Then suitable statements are invented or discovered or just guessed to express the loop in some programming language. Finally the loop is tested and retested to see whether the statements perform as expected. Unfortunately there is a major flaw in debugging. Again quoting Dijkstra: "debugging can only prove the presence of bugs, not their absence". One can never be certain that debugging has removed all design omissions and programming errors. Once a program is written in this way it is remarkably difficult and often impossible to determine a loop invariant for its loops.

This made some computing scientists suspicious. Why is it so difficult or even impossible to determine the loop invariant which defines the set of states in state space which is available to the iterations of the loop? Perhaps the specification of the loop is unclear or the program is incorrect or its complexity exceeds our comprehension?

Fortunately there is another way pioneered by E.W. Dijkstra to construct loops with complexity under control so that the correctness of the program can be determined by logical reasoning rather than by machine testing of program code. This method starts with the loop invariant and uses it and the need to make at least one step towards termination to derive the statements of the loop program.

4.6 Summary

Our mini-language, which will be our notation for the derivation of programs will permit the following statements

\[
<\text{statement}> ::= <\text{assignment statement}> \\
| <\text{IF-statement}> \\
| <\text{repetitive statement}> \\
| \text{skip} \\
| <\text{statement}> ; <\text{statement}>
\]
5.0 STATES AND STATE SPACE

A mathematical variable may represent an infinite number of values in the integer domain and an uncountably infinite number of values in the decimal number or as we sometimes say "real number" domain.

A programming variable always represents a finite number of values simply because a finite number of bits is used to represent an integer or a real number in a computer. This number may be large as for integer or real number representations or just equal to two for Boolean values.

It is sometimes useful to construct a model of a computation, a description of what happens as a program is executed. One such model borrows most of its concepts from physics and is defined as follows:

Each variable of a program defines one dimension of a state space of the computation. Each distinct value of a variable defines a separate state of the computation for that dimension. For example, a Boolean variable defines one dimension with two separate states, an integer variable defines one dimension with $2^n$ separate states where $n$ is the number of bits used in the representation of integer numbers.

At any one instant in time, the set of current values of all variables of a program defines the current state of the computation. An assignment statement usually changes the value of at least one variable and so it changes the current state of the computation. In our mini-language and model, the assignment statement is the only statement which can change the state of the computation.

We say that the dimensions associated with the variables of a program define the state space of the program. The execution of a program follows a path in state space from a state corresponding to the initial values of the variables to a state that is acceptable as a solution state of the problem which the program was supposed to solve.

A problem may therefore be specified as follows:
"starting in one state of a set of initial states, the program should terminate when the values of the variables form a state which is one of a set of acceptable final states."

At this stage the reader might think that we are trying to confuse rather than clarify the process of programming but it will become apparent in subsequent sections that it is indeed quite easy to express problem statements in terms of sets of states. This is because a well defined and long established branch of mathematics called mathematical logic can be used to guide our arguments and discussions.

5.1 Predicates as Selections of Sets of States

Any Boolean expression containing variables used by a computation is said to select those states from the state space of a computation for which the value of the Boolean expression is true. For example, if the integer variable \( i \) represents the one-dimensional state space of a computation then

\[
i = 25
\]

selects one state, the expression

\[
i = -5 \text{ or } i = 10
\]

selects two states and

\[
i > 37
\]

selects many states, namely all states in the range 38..\text{largestint}. The Boolean constant

\[
true
\]

selects all states and the Boolean constant

\[
false
\]

selects no states (the empty set). The concepts of state space and predicate can be used to sharpen the precision of problem specification as shown in the next section.
THE SPECIFICATION OF PROBLEMS

Problems may be described in many different ways. For example let us state two simple problems

Problem 1:  *Find the largest element in a sequence of numbers*

Problem 2:  *Find the sum of a sequence of numbers*

This formulation gives no hint on how the problem should be solved or how the sequence of numbers should be represented in the computer. It maximizes the freedom of the programmer. Let us state a more specific description of the same problems

Problem 1:  *Given a fixed array of integers* \(B[0..N-1]\) *and* \(N \geq 0\)  
*Determine*  
*the value of* \(\text{max}\) *such that* \(\text{max} = \text{largest element of } B.\)

Problem 2:  *Given a fixed array of integers* \(B[0..N-1]\) *and* \(N \geq 0\)  
*find*  
\[\text{sum} = \sum_{i=0}^{N-1} B[i]\]

This description of our two problems suggests that the sequence of numbers should be stored as an array which is a data structure that is available in most programming languages. It states that the array values are fixed, that the program is not allowed to alter or permute the values of \(B[0..N-1]\). It also specifies that the array may be empty so that we have to find an acceptable definition for the value of the largest element of an empty array and for the sum of the values of the elements of an empty array. The value \(\text{sum} = 0\) is acceptable for the sum of no elements but the maximum value of no elements presents a problem.

The value 'undefined' would be acceptable but although in most programming languages we can define an empty set or an empty string, we have no value 'undefined' for integer or real numbers. In such a case we can respecify the problems to consider non-empty arrays only \((N > 0)\) or introduce
by definition some artificial value such as max = smallest available number. In the subsequent development of Problem 1 we will restrict ourselves to non-empty arrays (N > 0).

The specification of the problem may be sharpened further by replacing as many words as possible by predicates that describe acceptable states or sets of states. The given initial conditions are referred to as the pre-condition Q and the expected final state is defined by the postcondition R. In terms of Q and R our problem specification becomes

Problem 1:  
Q : B[0..N-1] is a fixed array of integers and N > 0  
R : max = largest element of B

Problem 2:  
Q : B[0..N-1] is a fixed array of integers and N ≥ 0  
\[ R : \text{sum} = \sum_{i=0}^{N-1} B[i] \]

Our aim is to reduce the number of words used in the problem specification and to rely more and more on predicates to specify sets of states precisely and concisely. To describe properties of whole arrays we borrow the concepts of universal and existential quantification from predicate calculus.

6.1 An Application of the Universal Quantifier

The statement  
max is at least as large as the value of the largest element of B

can be expressed in the following way:

for all values of the subscript i such that i is greater than or equal to zero and less than N we have max ≥ B[i].

This sentence is more concisely expressed by the predicate (see Section 1.8)

\[ (A_i : 0 \leq i < N : \text{max} \geq B[i]) \]

"for all i" "such that" "we have"
This is not yet a complete postcondition for Problem 1. The condition does not state that \texttt{max} must be equal to one of the array elements. This condition will be discussed in the next section.

The postcondition of our second problem can be expressed using a slight modification of this form as

\[
R: \text{sum} = ( \Sigma i : 0 \leq i < N : \text{B}[i])
\]

"sum over i" "such that" "we have summed"

where \texttt{sum} is a variable whose value is equal to the sum of the array elements so that this is an equivalent expression of the conventional mathematical formula

\[
\text{sum} = \sum_{i=0}^{N-1} \text{B}[i]
\]

From time to time we will express different counting or summing concepts in a form that resembles a universal quantifier in order to simplify subsequent algebraic manipulations.

6.2 An Application of the Existential Quantifier

The statement

\textit{the value max occurs at least once in the array B[0.. N-1] where N > 0}

may be rewritten as

\textit{There exists at least one value of the subscript \(i\) in the range \(0 \leq i < N\) such that \(\text{max} = \text{B}[i]\).}

Once again we use the notation of predicate calculus to say the same more concisely

\[
( Ei : 0 \leq i < N : \text{max} = \text{B}[i])
\]

"there exists at least one \(i\)" "such that" "we have"

6.3 Preconditions and Postconditions as Predicates

Our problems can now be specified using predicates and very few if any explanatory sentences or words

Problem 1: Given \(Q : \text{B[0..N-1] is a fixed array of integers and } N > 0\).

\textit{Write a program that establishes}

\[
R : (Ai : 0 \leq i < N : \text{max} \geq \text{B}[i]) \land (Ei : 0 \leq i < N : \text{Max} = \text{B}[i])
\]
Problem 2: Given \( Q: B[0..N-1] \) is a fixed array of integers and \( N \geq 0 \).

Write a program that establishes

\[
R : \text{sum} = ( \sum_{i=0}^{N} B[i] )
\]

7.0 THE SHAPE AND FORM OF ONE-LOOP PROGRAMS

Our two problems have almost identical preconditions. Their postconditions are also somewhat similar. They are representative of a whole class of problems which manipulate a one-subscript array in some way. Such problems are solved with one loop which in most cases contains only one guarded statement. In many cases array elements are examined in sequence with an increasing or decreasing index. The form of such a program that processes the array elements one by one in increasing index order is given by the template

\[
k := 0; \quad \text{statements to establish the truth of the loop invariant } P
\]

\[
do \quad \text{if } k \neq N \rightarrow \text{statements to make progress towards the termination of the loop.}
\]

\[
k := k + 1 \quad \text{od}
\]

Floyd and Dijkstra discovered that a predicate is the most concise way to describe the state of a computation after each iteration of the loop. If the predicate is also true before the loop starts and remains true after the loop terminates then it fully describes the sequence of states which the computation takes as the loop progresses from start to termination. Such a predicate is called the loop invariant \( P \) and as Dijkstra observed the loop invariant "defines the essence of what the loop does". Dijkstra was first to realize that the loop invariant can be used to derive the unspecified program statements in the above template in such a way that the correctness of the program is established by logical arguments or by algebraic manipulation of symbols rather than by extensive computer testing. The loop invariant \( P \) has remarkably simple properties.
(i) \( P \) is a predicate that selects (is true for) the sequence of states which the computation finds itself in at the end of each iteration during the execution of a loop.

(ii) \( P \) also selects the state in which loop execution is allowed to commence.

(iii) \( P \) is true for the state in which the loop execution terminates

Hence the purpose of the first group of unspecified statements in the program template is to bring the computation into a state for which \( P \) is true. We describe this part of the program by the words "establish the truth of \( P \)" and we denote the establishment of the truth of a predicate by curly brackets around the predicate. Hence the first part of the template is written as

\[
k := 0
\]

\[
\{P\}
\]

where the statements that establish \( P \) are still unspecified because \( P \) itself is not specified yet. During the execution of the loop \( P \) is true at the completion of each iteration. Hence together with the problem specification of the previous section the program template becomes

\[
\{Q\}
\]

\[
k := 0;
\]

\[
\{P\}
\]

\[
do \ k \neq N \rightarrow 
\]

\[
k := k+1 \quad \{P\}
\]

\[
\od \quad \{R\}
\]

If there are no program statements after the end of the loop then \( R \) is a conjunction of \( P \) and the negation of the guard

\[
R \equiv \text{not (guard) and } P
\]

In the above template this relation becomes
\[ R \equiv (k = N) \land P \]

It took some time to realize that this conjunction can be used to derive \( P \) from \( R \). Again Dijkstra showed us the way aided by the work of David Gries and the Programming Science Group at the University of Eindhoven in Holland.

\section*{8.0 THE DERIVATION OF PROGRAMS}

This section explores the idea that for a one-loop program in which the loop is the last statement, the postcondition conceals the loop invariant as well as the complement of the guard because

\[ R \equiv \text{not (guard)} \quad \text{and} \quad P \]

Therefore the postcondition which is given by the problem statement can be used to discover a suitable loop invariant \( P \) and loop guard which we will for historical reasons call BB. Once the postcondition has been formulated, the derivation of predicates which are suitable loop invariants is a non-unique algebraic activity that follows a set pattern of steps and rules and requires practically no creative effort. By contrast as we all know, the invention of programs or the guessing of suitable loop invariants is very difficult and requires a great creative effort even for relatively simple programs.

\subsection*{8.1 Rules for the Derivation of \( P \) from \( R \)}

The idea is to split \( R \) into two parts, one which is the loop invariant \( P \) and another which is the negation of the guard BB. If the original formulation of the postcondition does not have an easily separable form, it is manipulated by using Boolean and predicate calculus into an equivalent form which permits separation. The most common rules for the derivation of \( P \) from \( R \) are given below.

(i) replace a constant or an expression by a variable  
(ii) omit a conjunct (\textbf{and} clause)  
(iii) introduce a disjunct (\textbf{or} clause)  
(iv) introduce a new variable
For example, the postcondition for Problem 1 is

\[ R : ( \forall i : 0 \leq i < N : \max \geq B[i]) \land (\exists i : 0 < i < N : \max = B[i]) \]

where \( N \) is a constant. Replacing \( N \) by a new variable \( k \) gives a possible loop invariant

\[ P : ( \forall i : 0 \leq i < k : \max \geq B[i]) \land (\exists i : 0 < i < k : \max = B[i]) \]

so that

\[ R \equiv (k = N) \land P \]

and a loop guard for this choice of \( P \) is the negation of \( k = N \) which is

\[ BB : k \neq N \]

What statements are required to make \( P \) true initially? At \( k = 0 \) the range \( 0 \leq i < k \) is empty because there is no value of \( i \) that satisfies \( 0 \leq i < 0 \). By definition the universal quantifier is true for an empty range. There is however some difficulty in defining a meaningful value for \( \max \) when the range is empty so that \( Q \) specifies \( N > 0 \) which means that there is always at least one element in the array so that \( P \) can be established by the assignment statements

\[
\begin{align*}
  k &:= 1; \\
  \text{max} &:= B[0]; \quad \{P\}
\end{align*}
\]

or writing this as a multiple assignment

\[
\begin{align*}
  k, \text{max} &:= 1, B[0] \quad \{P\}
\end{align*}
\]

Note that we say \( P \) is established if the values of the program variables contained in \( P \) are such that the predicate \( P \) yields the value true.

The program template for this problem therefore becomes
\[ k, \text{max} := 1, B[0]; \{Q\} \]
\[ \text{do } k \neq N \rightarrow \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \text{od } \{P\} \]
\[ k := k+1 \{P\} \]

where

\[ Q : B[0..N-1] \text{ is a fixed array of integers and } N > 0 \]

\[ P : (\forall i : 0 \leq i < k : \text{max B[i]} \Lambda (\exists i : 0 \leq i < k : \text{max} = B[i]) \]

\[ R : (\forall i : 0 \leq i < N : \text{max B[i]} \Lambda (\exists i : 0 \leq i < N : \text{max} = B[i]) \]

To complete the program we have to derive the unspecified statements of the guarded statement inside the loop.

### 8.2 The Derivation of the Loop Body from P

In this section we will discuss variables and values of variables. In computing science a variable denotes a memory location which stores a value. For example an integer variable stores exactly one integer value. The variable is represented by an identifier so that in Pascal

\[ \text{var} \]
\[ k : \text{integer}; \]
\[ \text{begin} \]
\[ k := 5; \]
\[ \ldots \]

defines a memory location named \( k \) into which the value 5 is stored. It is customary to refer to the value of a variable by its name so that in the Pascal statement

\[ \text{writeln ('}k'=, k) \]

the variable name \( k \) refers to the current value of the variable \( k \) and not to the memory location where it is stored. Normally there is no confusion in the double duty of the variable name. The context in which it is used determines whether it refers to the memory location where the value is stored or to the value itself.
Confusion arises however when we have to discuss two different general values of the same variable. In such cases we must clearly distinguish between the value of a variable and the name of the location where the variable is stored. Let us introduce the following notational conventions (using the identifier k as an example)

(i) the identifier \( k \) denotes the location of the value of \( k \) or in the mathematical sense we can say that \( k \) denotes any one of the possible values of \( k \). If the predicate \( P \) depends on \( k \) we can express this as
\[
P(k)
\]

(ii) a particular value stored at the location \( k \) may be denoted by \( k' \) and we will use a special notation to denote this:
\[
P(k'/k)
\]
means that we are considering the predicate \( P \) with the specific value \( k' \) substituted for the variable \( k \).

(iii) a different value may be denoted by \( k'' \) and so on by attaching more and more primes to the name of the identifier.

The loop of our problem can now be described as follows: upon entry of the loop \( k \) has a certain value \( k' \). Upon completion of the next iteration \( k \) receives a new value \( k'' = k' + 1 \). Using our notation we have

\[
\{ P(k'/k) \}
do \ k \neq N \rightarrow
\]

\[
\begin{array}{c}
\vdots \\
k := k + 1 \\
\{ P(k''/k) \text{ and } k'' = k' + 1 \}
\end{array}
\od
\]

so that before the loop starts execution of the loop body for the current iteration \( P(k) \) is \text{true} with the value \( k = k' \) while after the iteration is completed \( P(k) \) is again \text{true} with the new value \( k = k'' = k' + 1 \). The purpose of the so far unspecified statements of the loop body is to change the state of computation so that \( P(k' + 1/k) \) becomes \text{true} so that the new value \( k'' = k' + 1 \) can be stored in \( k \).
by the last assignment statement of the loop body to establish the truth of 
P(k''/k) for the new value k = k'' of k.

There is a simple way to determine what statements are required to establish the truth of \( P(k''/k) \). We form \( P(k''/k) \) by substituting for every occurrence of k in P the expression \( k' + 1 \) which expresses the new value \( k'' \) in terms of the old value \( k' \). Since \( P(k'/k) \) is true before the start of the iteration, a large part of the new predicate \( P(k'+1/k) \) will be true already. In Problem 1 the first part of the loop invariant is

\[
P : (\forall i : 0 \leq i < k : \text{max} \geq B[i])
\]

so that

\[
P(k'/k) : (\forall i : 0 \leq i < k' : \text{max} \geq B[i])
\]

and

\[
P(k'+1/k) : (\forall i : 0 \leq i < k'+1 : \text{max} \geq B[i])
\]

By splitting off the last element we get

\[
P(k'+1/k) \equiv (\forall i : 0 \leq i < k' : \text{max} \geq B[i]) \quad \text{and} \quad (\text{max} \geq B[k'])
\]

\[
\equiv P(k'/k) \quad \text{and} \quad (\text{max} \geq B[k'])
\]

since \( P(k'/k) \) already yields true before the iteration starts, the unspecified statements of the iteration have to establish

\[
\text{max} \geq B[k']
\]

This is accomplished by the rather obvious IF-Statement

\[
\text{if}
\begin{align*}
\text{max} & \geq B[k'] \quad \rightarrow \quad \text{skip} \\
\lbrack[] \text{max} < B[k'] \quad \rightarrow \quad \text{max} := B[k']
\end{align*}
\text{fi}
\]

The truth of the second part of the loop invariant is obviously established by the IF-statement above. After the execution of the IF-statement, the truth of the invariant \( P(k) \) has been extended to include the value \( k'' = k'+1 \). Therefore \( P(k''/k) \) is true and after the assignment statement

\[
k := k'+1
\]
which changes the value of \( k \) itself from \( k' \) to \( k'' \) we can say that

\[
P(k)\]

is true for the new range of values of \( i \). This argument may be repeated for the next iteration. Our program derivation is now completed. The program of the loop body at the start of any one iteration is as follows

\[
\{P(k'/k) \quad \text{do} \quad k' \neq N \rightarrow \text{if} \quad \text{max} \geq B[k'] \rightarrow \text{skip} \\
\quad \text{[] max} < B[k'] \rightarrow \text{max} := B[k'] \\
\quad \text{fi; } k := k' + 1 \quad \{P(k''/k) \text{ and } k'' = k'+1\} \quad \text{od} \]

where we have explicitly shown the current value of \( k \) as it will be at each step of the execution of the current iteration of the loop. The whole program becomes

\[
\{Q\} \\
\text{k, max := 1, B[0];} \\
\{P\} \\
\text{do } k \neq N \rightarrow \text{if} \quad \text{max} \geq B[k] \rightarrow \text{skip} \\
\quad \text{[] max} < B[k] \rightarrow \text{max} := B[k] \\
\quad \text{fi; } k := k + 1 \quad \{P\} \quad \text{od} \\
\{R \equiv k = N \text{ and } P\}
\]

This program is easily translated into a Pascal procedure where the fixed array \( B \) is assumed to be global to the procedure

\[
\text{procedure } \text{FindMax (n : integer);} \\
\quad \text{var } \\
\quad k : \text{integer}; \\
\quad \text{begin} \\
\quad k := 1; \\
\quad \text{max} := B[0]; \\
\quad \text{while } (k <\ n) \text{ do} \\
\quad \text{begin} \\
\quad \quad \text{if } \text{max} < B[k] \text{ then } \text{max} := B[k]; \\
\quad \quad k := k + 1 \\
\quad \text{end;}
\]

end;
8.3 Solution of Problem 2

Consider the problem specification of Problem 2 as given in Section 5.3

Problem 2: Given $Q$: $B[0..N-1]$ is a fixed array of integers and $N \geq 0$
Write a program that establishes

$R$: \( \sum = (\Sigma : 0 \leq i < N : B[i]) \)

Let us derive a possible loop invariant by the replacement of the constant $N$ by the variable $k$ so that

$P$: \( \sum = (\Sigma : 0 \leq i < k : B[i]) \)

and the postcondition $R$ is

$R \equiv P \text{ and } (k = N)$

and the loop guard $BB$ is the negation of $(k = N)$

$BB: k \neq N$

The sum of the elements of an empty sequence (no elements) is equal to zero. Hence $P$ may be established by the multiple assignment

$k, \text{ sum} := 0, 0; \{P\}$

The template program becomes

\[
\begin{align*}
\text{k, sum} & := 0, 0; \{P\} \\
\text{do k} & \neq N \rightarrow \\
& \text{----------} \\
& \text{----------} \\
& \text{k} := k + 1 \{P\} \\
\text{od} \\
\{R \equiv P \text{ and } (k = N)\}
\end{align*}
\]

the statements of the loop body are determined by finding what needs to be done to establish $P(k'+1/k)$ when $P(k'/k)$ is already true.

$P(k'+1/k) \equiv \sum = (\Sigma : 0 \leq i < k'+1 : B[i])$

$\equiv \sum = (\Sigma : 0 \leq i < k' : B[i]) + B[k']$

Upon entry to the current iteration of the loop the variable $\text{sum}$ already contains a value equal to the sum of the first $k'$ terms. $P(k'+1/k)$ becomes true if the value of the $(k'+1)$th element of the array $B[k']$ is added to the sum. Therefore the statement that establishes $P(k'+1/k)$ is

$\text{sum} := \text{sum} + B[k]$
and the program becomes

\[
\begin{align*}
& k, \text{sum} := 0, 0; \quad \{P\} \\
& \text{do } k \neq N \rightarrow \text{sum} := \text{sum} + B[k]; \\
& \quad k := k + 1 \quad \{P\} \\
& \text{od} \quad \{R \equiv P \text{ and } (k = N)\}
\end{align*}
\]

This can be translated into a Pascal procedure as follows:

```
procedure Sum Array;
var
  k: integer;
begin
  k := 0;
  sum := 0;
  while (k < N) do
    begin
      sum := sum + B[k];
      k := k + 1
    end
end;
```

**9.0 ANOTHER SIMPLE PROBLEM**

Problem 3: Given \(Q:\) \(N\) is a fixed integer and \(N \geq 0\)

_Write a program to establish_

\[
R : \quad 0 \leq a^2 \leq N < (a + 1)^2 \text{ and } a \geq 0
\]

This problem requires us to write a program which finds the largest integer \(a\) that is at most equal to the square root of \(N\). Rewrite \(R\) as several conjuncts

\[
R : \quad 0 \leq a^2 \text{ and } a^2 \leq N \text{ and } N < (a+1)^2 \text{ and } a \geq 0
\]

**9.1 Deleting a Conjunct**

Obtain a possible \(P\) by the deletion of the third conjunct of \(R\) to give

\[
P : \quad 0 \leq a^2 \text{ and } a^2 \leq N \text{ and } a \geq 0
\]

so that

\[
R \equiv P \text{ and } N < (a+1)^2
\]

Because \(Q\) states that \(N \geq 0\), \(P\) can be established by the assignment

\[
a := 0
\]
We now use our creative and intuitive powers to determine that the most reasonable progress towards termination of the loop is made by incrementing the value of a by one so that the program becomes

\[
a := 0; \{ P \}
\]

\[
do \quad N \geq (a+1)^2 \rightarrow \quad a := a+1 \quad \{ P \}
\]

\[
od \quad \{ R \equiv P \text{ and } N < (a+1)^2 \}
\]

Progress towards termination of the loop is due to \( a := a+1 \). For any particular iteration if \( a \) has the value \( a' \) initially then \( a \) has the value \( a'' = a'+1 \) after its completion. The missing statements are derived from

\[
P(a'/a) \equiv 0 \leq (a')^2 \text{ and } (a')^2 \leq N
\]

\[
P(a'+1/a) \equiv 0 \leq (a'+1)^2 \text{ and } (a'+1)^2 \leq N
\]

The first conjunct is always true. The second conjunct of \( P(a'+1) \) is true if the guard of the loop is true! Hence the loop body requires no further action and the program is

\[
a := 0; \{ P \}
\]

\[
do \quad N \geq (a+1)(a+1) \rightarrow a := a+1 \quad \{ P \}
\]

\[
od \quad \{ R \equiv P \text{ and } N < (a+1)^2 \}
\]

This program starts with zero and linearly searches through increasing integers until it finds the required value. This is not a very efficient algorithm and the question arises whether there is a way to derive a better solution. The derivation of a solution is not a unique process therefore another method of deriving \( P \) from \( R \) will give another solution provided the new \( P \) can be initially established and maintained.

**9.2 Replace an Expression by a Variable**

This time let us derive a \( P \) by replacing an expression with a variable so that from

\[
R : 0 \leq a^2 \leq N < (a+1)^2 \quad \land \quad a \geq 0
\]
we obtain the invariant
\[ P : a^2 \leq N < b^2 \land a \geq 0 \land b \geq 0 \]
by replacing \((a+1)^2\) by \(b^2\). Since the problem states that \(a > 0\) we choose \(b\) to be positive also. We now have

\[ R \equiv P \text{ and } b^2 = (a+1)^2 \]

so that the loop guard is \(b^2 \neq (a+1)^2\). For efficiency reasons we can replace this by \(b \neq (a+1)\) because both \(a\) and \(b\) are positive or zero. To establish \(P\) initially, let us look at each inequality in turn:

\[ a^2 \leq N \]

is initialized by \(a := 0\) because \(Q\) states that \(N \geq 0\). The second inequality

\[ N < b^2 \]

requires some value of \(b\) for which the inequality obviously holds. Since \((N + 1)^2 > N\) even for \(N = 0\), we can satisfy the second inequality by choosing \(b = N + 1\). Therefore \(P\) is established by the assignment statement

\[ a, b := 0, N+1 \quad \{P\} \]

In this solution there is no index variable that steps through a range. Progress towards termination is achieved by an increase of \(a\) or a decrease of \(b\) until the values of \(b\) and \(a\) differ by one \((b = (a+1))\).

Increments like \(a := a+1\) or decrements like \(b := b-1\) are certainly possible but a more efficient algorithm will be obtained if the largest possible portion of the range between \(a\) and \(b\) is eliminated at each step. If nothing is known about the location of the desired value, the halfway point between \(a\) and \(b\) is the best choice. The program of the loop body is derived by the same analysis as in the previous sections. Suppose that the values of \((a,b)\) at the beginning of an iteration are \((a',b')\) and at the end of the iteration we wish to leave \(b'' = b'\) but increase the value of \(a\) to

\[ a'' = (a'+b') \div 2 \]
then the invariance of \( P \) requires that \( P(a''/a,b''/b) \) yields \textit{true} where

\[
P(a''/a,b''/b) \equiv ((a'+b') \div 2)^2 \leq N < (b')^2
\]

The condition \( N < (b')^2 \) is satisfied because \( P(a',b') \) is \textit{true} but the value of \( a \) can only be increased from \( a' \) to \( a'' \) if the left condition is satisfied. This is achieved by the guarded statement

\[
((a'+b') \div 2)^2 \leq N \rightarrow a := (a'+b') \div 2
\]

The value of \( b \) can be decreased from \( b' \) to

\[b'' = (a'+b') \div 2\]

provided that we can establish the truth of

\[
P(a''/a,b''/b) \equiv (a')^2 \leq N < ((a'+b') \div 2)^2
\]

In this case the first inequality is satisfied because \( P(a'/a,b'/b) \) is \textit{true} but \( b' \) can be increased to \( b'' \) only if the second condition is satisfied. This is achieved by the guarded statement

\[
N < ((a'+b') \div 2)^2 \rightarrow b := (a'+b') \div 2
\]

Combining these two guarded statements into one IF-statement we obtain the program

\[
a, b := 0, N + 1; \quad \{ P \} \\
do b \neq (a+1) \rightarrow \quad \text{if} \\
\quad ((a+b) \div 2)^2 \leq N \rightarrow a := (a+b) \\
\quad \text{div } 2 \\
\quad \{ P \} \\
\quad [N < ((a+b) \div 2)^2 \rightarrow b := (a+b) \\
\quad \text{div } 2 \\
\quad \text{fi} \\
\quad \text{od} \\
\quad \{ R \equiv P \text{ and } (b = (a+1)) \}
\]

For efficiency reasons introduce a new variable \( d \) so that the midpoint of the range is calculated only once. The introduction of a new variable into the loop body invariably changes \( P \). Otherwise this variable has no role to play in the derivation of the program for the loop body. In this case however the scope of \( d \) is restricted to the IF-statement where it represents the value of an expression that occurs several times. The variable \( d \) is therefore local to the loop body and we can omit it from \( P \). The program becomes
This program like binary search takes only about log base two of \(N\) steps to find the required value where the previous algorithm took of the order of \(N\) steps. The program is once again easily translated into a Pascal procedure.

```pascal
procedure FastFindSqrt;
var
   b, d: integer;
begin
   a := 0;
   b := N+1;
   while b <> (a+l) do
      begin
         d := (a+b) div 2;
         if d*d <= N then a := d
         else b := d
      end;
end;
```

10.0 BINARY SEARCH

This section presents Dijkstra's solution to the well known binary search algorithm which shows that even for well known algorithms this method yields a surprisingly simple and elegant program.

Problem 4: Q: \(A[0..N]\) is an ordered sequence of integers and \(X\) is an integer and \(A[0] \leq X < A[N]\)

Determine if there exists at least one \(i\) such that \(0 \leq i < N\) and \(X = A[i]\).

This problem is different from the previous problems in that the postcondition \(R\) is not given. The specification of \(R\) requires an intuitive insight, a creative idea.
Idea: Since $X$ is already contained in the half-open interval between $A[0]$ and $A[N]$, derive the loop invariant $P$ from the precondition $Q$ by replacing the two constants 0 and $N$ by variables $i$ and $j$ so that

$$P : A[i] \leq X < A[j]$$

If we increase the value of $i$ or decrease the value of $j$ under invariance of $P$ until there are no interior points in the interval between $i$ and $j$ so that

$$j = i+1$$

and $P$ still holds true then we have

$$A[i] \leq X < A[i+1]$$

and the truth value of the boolean expression

$$A[i] = X$$

determines whether the value of $X$ is present in the array or not. The postcondition is therefore

$$R : A[i] \leq X < A[j] \quad \text{and} \quad j = (i+1)$$

so that the guard of the loop is

$$BB : j \neq i+1$$

$P$ is established by the assignment statement

$$i, j := 0, N$$

so that the program becomes

$$i, j := 0, N; \quad \{P\}$$

$$\text{do} \quad j \neq i+1 \quad \rightarrow \quad \text{ }$$

$$\text{ }$$

$$\text{ }$$

$$\text{ }$$

$$\text{ }$$

$$\text{ }$$

$$\text{od}$$

$$\{ R \equiv P \quad \text{and} \quad j = (i+1) \}$$

As in Section 8.2 progress towards completion of the loop is made by increasing $i$ or decreasing $j$. If we introduce a new variable $k$ such that
\[ k := (i+j) \div 2 \]

then the value \( i \) can be increased to \( (i'' = k' = (i'+j') \div 2) \) provided that we can establish the truth of

\[ \mathbf{P}(k'/i,j'/j) = A[k'] \leq X < A[j'] \]

and this requires the guarded statement

\[ A[k'] \leq X \rightarrow i := k' \]

while the truth of

\[ \mathbf{P}(i'/i,k'/j) = A[i'] \leq X < A[k'] \]

requires the guarded statement

\[ X < A[k'] \rightarrow j := k' \]

Gathering these guarded statements into an IF-statement yields the program

\[
\begin{array}{l}
i, j := 0, N; \{ \mathbf{P} \} \\
do j \neq i+1 \rightarrow k := (i+j) \div 2; \\
\quad \text{if} \\
\quad \quad A[k] \leq X \rightarrow i := k \\
\quad \quad [ \] X < A[k] \rightarrow j := k \\
\quad \fi \{ \mathbf{P} \} \\
\od \\
\{ \mathbf{R} = \mathbf{P} \quad \text{and} \quad j = i+1 \} \\
\end{array}
\]

This program easily translates into a Pascal function that returns the value \textbf{true} if the value \( X \) is present in the array and \textbf{false} if it is not present.

\[
\begin{align*}
\textbf{function} & \quad \text{BinSrch} : \text{boolean}; \\
\text{var} & \quad i,j,k : \text{integer}; \\
\text{begin} & \quad i := 0; \\
& \quad j := N; \\
\text{while} & \quad j \neq (i+1) \text{ do} \\
& \quad \text{begin} \\
& \quad \quad k := (i+j) \div 2; \\
& \quad \quad \text{if} \quad (A[k] \leq X) \\
& \quad \quad \quad \text{then} \quad i := k \\
& \quad \quad \quad \text{else} \quad j := k \\
& \quad \quad \text{end;} \\
& \quad \text{BinSrch} := (X = A[i]) \\
\text{end;}
\end{align*}
\]
11.0 A PROGRAM FOR FORWARD DIFFERENCES

With the notation and techniques established in the previous sections, let us derive a program which establishes a result which is not immediately obvious. David Gries [3] used this problem at the Newport Summer School but he did not emphasize the connection with the forward and backward difference formulas of numerical mathematics.

**Problem:** Given \( Q : N \geq 0 \), derive a program that establishes
\[
R : (A_i : 0 \leq i < N : f_i = i^3)
\]
without the use of multiplication, division or exponentation operations.

The first step is to derive the loop invariant and a predicate that can serve as the negation of the loop condition. In this case this can be done by splitting the postcondition into two conjuncts where we have introduced a subscript on \( P \) in anticipation of more components of \( P \) that will be derived later.

\[
R \equiv (A_j : 0 \leq j < i : f_j = j^3) \land i = N
\]
\[
\equiv P_o \land i = N
\]

The next step is to determine by some creative insight or by a simple guess how to make progress towards termination. In this case let us consider the usual assignment statement

\[
i := i + 1
\]

Let us start with the empty range by initializing

\[
i := 0 \quad \{ P_o \}
\]

and the program that has been derived so far may be written as

\[
i := 0 \quad \{ P_o \}
\]
\[
do \quad i \neq N \rightarrow \\
\quad \quad \\
\quad \quad \\
\quad \quad i := i + 1
\]
\[
\od;
\]
The rest of the loop body is derived from the requirement to establish the truth of $P_0(i'+1/i)$ so that the value of $i$ can be increased by one in the last statement of the loop body.

$$P_0(i'+1/i) = (\forall j: 0 \leq j < i'+1: f[j] = j^3)$$

$$= P_0(i'/i) \land f[i'] = (i')^3$$

Exponentiation or multiplication are not permitted as operations so that in this case the simple assignment statement

$$f[i] := i^3$$

cannot be used. Another alternative is to introduce an auxiliary variable $n_1$ such that its value is always equal to $i^3$. This requires a new component of the loop invariant

$$P_1 = n_1 = i^3$$

or explicitly in terms of the current value of $i$

$$P_1(i'/i) = n_1 = (i')^3$$

Because $i$ is initialized to zero, this component of $P$ can be established by

$$n_1 := 0$$

so that upon entry to the loop body $P_1(i'/i)$ yields true. $P_0(i'/i)$ is also true so that we can now establish the truth of $P_0(i'+1/i)$ by the assignment statement

$$f[i] := n_1 \{ P_0(i'+1/i) \}$$

The program therefore becomes

$$i, n_1 := 0, 0; \{ P_0 \land P_1 \}$$

$$\text{do } i \neq N \rightarrow f[i] := n_1; \{ P_0(i'+1/i) \}$$

$$\overline{i := i+1}$$

$$\text{od;}$$

The program is not complete because without further statements after $f[i] := n_1$ and before $i := i+1$ we do not necessarily have $P_1(i'+1/i)$ yielding true, so we have to investigate $P_1(i'+1/i)$.

$$P_1(i'+1/i) = n_1 = (i'+1)^3$$

$$= (i')^3 + 3(i')^2 + 3i' + 1$$
so that the truth of $P_1(i'+1/i)$ requires that

$$n_1 = (i')^3 + 3(i')^2 + 3i' + 1$$

but the current value of $n_1$ was established for $P_1(i'/i)$ so that it is

$$n_1 = (i')^3$$

and $P_1(i'+1/i)$ is established by the statement

$$n_1 := n_1 + (3i')^2 + 3i' + 1$$

Since multiplication is not allowed, introduce another variable $n_2$ and another component of the invariant $P^2(i)$ such that

$$P^2(i) \equiv n_2 = 3i^2 + 3i + 1$$

This is initially established by

$$n_2 := 1$$

because the initial value of $i$ was zero. Upon entry to the loop body we have

$$P^2(i'/i) \equiv n_2 = 3(i')^2 + 3(i') + 1$$

so that $P_1(i'+1/i)$ can be satisfied by the assignment

$$n_1 := n_1 + n_2$$

and our program becomes

\[
\begin{align*}
i, n_1, n_2 &:= 0, 0, 1; & \{P^0 \land P_1 \land P_2\} \\
\text{do } i \neq N & \rightarrow \text{ f[i] := n1; } & \{P^0(i'+1/i)\} \\
& \text{ n1 := n1+n2; } & \{P_1(i'+1/i)\} \\
& \text{ i := i+1 } \\
\end{align*}
\]

The statement $n_1 := n_1+n_2$ must come after $f[i] := n1$ because the first statement requires the old value of $n_1$, the one established by $P_1(i'/i)$.

We now have to investigate $P^2(i'+1/i)$ where we find

$$P^2(i'+1/i) \equiv n_2 = 3(i'+1)^2 + 3(i'+i) + 1$$

$$= 3(i')^2 + 6i' + 3 + 3i' + 3 + 1$$

$$= 3(i')^2 + 3i' + 1 + 6i' + 6$$
while

\[ P_2(i') = n2 = (3i')^2 + 3i' + 1 \]

so that since multiplication is not allowed we can once more introduce

\[ P_3(i) = n3 = 6i + 6 \]

so that

\[ P_3(i') = n3 = 6i' + 6 \]

and \( P_2(i'+1)/i \) can be established by the assignment

\[ n2 := n2 + n3 \]

and \( P_3(i) \) can be initially established by

\[ n3 := 6 \]

The assignment to \( n2 \) must be placed after the last use of the old value of \( n2 \) in the program so that we have

\[
\begin{align*}
i, n1, n2, n3 &:= 0, 0, 1, 6; \quad \{P_0 \land P_1 \land P_2 \land P_3\} \\
do i \neq N &\rightarrow f[i] := n1; \quad \{P_0(i'+1/i)\} \\
&\quad n1 := n1 + n2; \quad \{P_1(i'+1/i)\} \\
&\quad n2 := n2 + n3; \quad \{P_2(i'+1/i)\} \\
&\quad i := i + 1
\end{align*}
\]

\[ \text{od} \]

Finally we have to investigate \( P_3(i'+1/i) \) so that

\[ P_3(i'+1/i) \equiv n3 = 6(i'+1) + 6 = 6i' + 6 + 6 \]

Since this can be established by an assignment statement

\[ n3 := n3 + 6 \]

without multiplication or exponentiation, there is no need to introduce further components of \( P \) and our program is completed. The program is

\[
\begin{align*}
i, n1, n2, n3 &:= 0, 0, 1, 6; \quad \{P_0 \land P_1 \land P_2 \land P_3\} \\
do i \neq N &\rightarrow f[i] := n1; \quad \{P_0(i'+1/i)\} \\
&\quad n1 := n1 + n2; \quad \{P_1(i'+1/i)\} \\
&\quad n2 := n2 + n3; \quad \{P_2(i'+1/i)\} \\
&\quad n3 := n3 + 6; \quad \{P_3(i'+1/i)\} \\
&\quad i := i + 1
\end{align*}
\]

\[ \text{od} \quad \{R = P_0 \land P_1 \land P_2 \land P_3 \land i=N\} \]
This program establishes the forward difference formula for a polynomial which is a well known result of numerical mathematics. It does so without any reference to numerical mathematics. It is therefore interesting to note that this technique may be used to derive mathematical results provided that the program is properly interpreted in the wider context of mathematical theory. As an exercise the reader may wish to derive the backward difference formula for the same problem.

12.0 ACKNOWLEDGEMENTS

The author is grateful to Peter Strazdins for valuable comments and proofreading.

13.0 REFERENCES