Lecture notes on the science of programming for CSCI121

Juris Reinfelds

University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/compsciwp

Recommended Citation

https://ro.uow.edu.au/compsciwp/60

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
# Lecture Notes on the Science of Programming

**For CIC1121**

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.0 Basic Concepts, Terms and Definitions</td>
<td>1</td>
</tr>
<tr>
<td>3.0 Programming Concepts</td>
<td>2</td>
</tr>
<tr>
<td>4.0 Algorithms and Programs</td>
<td>3</td>
</tr>
<tr>
<td>4.1 Assignments &amp; Multiple Assignments</td>
<td>3</td>
</tr>
<tr>
<td>4.2 Guarded Statements</td>
<td>4</td>
</tr>
<tr>
<td>4.3 The IF Statement</td>
<td>4</td>
</tr>
<tr>
<td>4.4 A Statement That Does Nothing</td>
<td>5</td>
</tr>
<tr>
<td>4.5 The Loop</td>
<td>5</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>7</td>
</tr>
<tr>
<td>5.0 Predicates as Selectors of Sets of States</td>
<td>8</td>
</tr>
<tr>
<td>6.0 The Specification of Problems</td>
<td>8</td>
</tr>
<tr>
<td>6.1 The Universal Quantifier</td>
<td>9</td>
</tr>
<tr>
<td>6.2 The Existential Quantifier</td>
<td>10</td>
</tr>
<tr>
<td>6.3 Preconditions and Postconditions as Predicates</td>
<td>10</td>
</tr>
<tr>
<td>7.0 The Shape and Form of One-Loop Programs</td>
<td>10</td>
</tr>
<tr>
<td>8.0 The Derivation of Programs</td>
<td>12</td>
</tr>
<tr>
<td>8.1 Rules for the Derivation of P from R</td>
<td>13</td>
</tr>
<tr>
<td>8.2 The Derivation of the Loop Body from P</td>
<td>14</td>
</tr>
<tr>
<td>8.3 Solution of Problem 2</td>
<td>17</td>
</tr>
<tr>
<td>9.0 Another Simple Problem</td>
<td>19</td>
</tr>
<tr>
<td>9.1 Deleting a Conjunct</td>
<td>19</td>
</tr>
<tr>
<td>9.2 Replace an Expression by a Variable</td>
<td>20</td>
</tr>
<tr>
<td>10. Binary Search</td>
<td>22</td>
</tr>
</tbody>
</table>
## 1.0 INTRODUCTION

The first book on the derivation of correct programs with the help of predicate calculus was "A Discipline of Programming" by E.W. Dijkstra which was published by Prentice Hall in 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. It is called "The Science of Programming" and it was published by Springer in 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 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 a later course.

## 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 gramophone 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.
3. **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 by E.W. Dijkstra some years ago 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 section will introduce the most important concepts of this notation.

4.1 **Assignments & Multiple Assignments**

An assignment 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 set of expressions to a set 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*

\[ \text{temp} := x; \quad x := y; \quad y := \text{temp} \]

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

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

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 is true. If the value of the guard is false, the statement is not executed and the computation proceeds to the next statement in the program.

A statement in this context means either a single statement or a list of statements. Using the symbol "::=" to denote the words "is defined as" and the symbol "|" to denote "or" we can write

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

This notation is called BNF (Backus-Naur Form). It can be used to define a guarded statement as:

\[
<\text{guarded statement}> ::= <\text{guard} > \rightarrow <\text{statement}>
\]

where the symbol in the oval actually appears in the program and is not like ::= a symbol that belongs to the description of the definition. This definition requires a definition of the guard but at this point we will assume that the reader is familiar with Boolean Calculus so that we can define

\[
<\text{guard}> ::= \text{Boolean Expression}
\]

where the oval shape denotes that the sequence of definitions terminates with a well understood symbol or sequence of symbols. Instead of using guarded statement 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 <statement> which has the following form.

\[
\text{if}
\text{guard 1 } \rightarrow \text{statement 1}
\text{[ guard 2 } \rightarrow \text{statement 2}
\text{[ guard n } \rightarrow \text{statement n}
\text{fi}
\]
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 \textit{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

\begin{verbatim}
if x \geq 0 then y := sqrt(x) else y := sqrt(-x)
\end{verbatim}

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

\begin{verbatim}
if
  x \geq 0 \rightarrow y := sqrt(x)
[ ] x < 0 \rightarrow y := sqrt(-x)
fi
\end{verbatim}

\subsection*{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 \textit{skip} so that the if statement

\begin{verbatim}
if x < 0 then x := -x
\end{verbatim}

is expressed by the IF-statement

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

\subsection*{4.5 The Loop}

The \textit{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 complete definition of the repetitive statement permits any number of guarded statements.

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 of the statements belonging to a true guard is selected for execution. 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

\[
\text{do} \\
\quad x < N \rightarrow x := x + 1 \\
\text{od}
\]

In both cases the loop terminates when the guard becomes 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 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 assumes 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 false. The guards are Boolean expressions and their condition (namely all false) is characteristic of the state of the computation at the termination of the loop. We therefore say that the Boolean expression

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

helps to select the state of computation in which the termination of the loop is possible. We say that a Boolean expression \emph{selects} those states of the computation for which the expression has the value true and we often call this expression a \emph{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 \emph{loop invariant} \( P \). This predicate selects the states of state space which are available to the iterations of the loop. In Dijkstra's words the loop invariant "\emph{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 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 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.

\section*{4.6 Summary}

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

\[ \text{<statement>} ::= \text{<assignment statement>} \]
\[ \text{<IF-statement>} \]
\[ \text{<repetitive statement>} \]
\[ \text{skip} \]
\[ \text{<statement>} ; \text{<statement>} \]
5.0 PREDICATES AS SELECTORS OF SETS OF STATES

Any Boolean expression 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

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

selects two states

\[ i > 37 \]

selects many states, namely all states in the range 38.. \textit{largestint}, 

\textit{true}

selects all states and

\textit{false}

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

6.0 THE SPECIFICATION OF PROBLEMS

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

Problem 1: \textit{Find the largest element in a sequence of numbers}

Problem 2: \textit{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: \textit{Given a fixed array of integers} \( B[0..N-1] \) \textit{and} \( N \geq 0 \)
\textit{Determine the value of} \( \text{max} \) \textit{such that} \( \text{max} = \text{largest element of} \ B \).

Problem 2: \textit{Given a fixed array of integers} \( B[0..N-1] \) \textit{and} \( N \geq 0 \)
\textit{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 $B[0..N-1]$. Also 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 or the sum 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 in most programming languages we can define an empty set or an empty string but 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 $\text{max} = \text{smallest available number}$. In the subsequent development of Problem 1 we will consider non-empty arrays only ($N > 0$).

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 $N > 0$  
$R$: $\text{max} = \text{largest element of } B$

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

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 more concisely we borrow two concepts from predicate calculus. The notation used to express them is due to Dijkstra.

6.1 The Universal Quantifier

The statement

"max contains 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 $\text{max} \geq B[i]$".

This sentence is more concisely expressed by the predicate

$$( \forall i : 0 \leq i < N : \text{max} \geq B[i] )$$
Similarly the postcondition of our second problem can be expressed using a slight modification of this form as

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

so that this is an equivalent expression of the conventional mathematical formula

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

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 The Existential Quantifier

The statement

"the value \( x \) occurs at least once in the array \( B[0..N-1] \) where \( N \geq 0 \"

may be rewritten as

"There exists at least one value of the subscript \( i \) in the range \( 0 \leq i < N \) such that \( x = B[i] \)."

Once again we use the notation of predicate calculus slightly modified by Dijkstra to say the same more concisely as

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

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: \( B[0..N-1] \text{ is a fixed array of integers and } N \geq 0 \).
Write a program that establishes
R: \( (A_i: 0 \leq i < N: \text{max} \geq B[i]) \)

Problem 2: Given Q: \( B[0..N-1] \text{ is a fixed array of integers and } N \geq 0 \).
Write a program that establishes
R: \( \text{sum} = (\sum_{i: 0 \leq i < N} B[i]) \)
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

\[
\begin{align*}
    k &: = 0; \\
    & \text{statements to establish the truth of the loop invariant } P \\
    \text{do } & k \neq N \\
    & \text{statements to make progress towards the termination of the loop.} \\
    k &: = k + 1 \text{ od}
\end{align*}
\]

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 argument or 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

\[
\begin{align*}
    k &: = 0 \\
    \{P\}
\end{align*}
\]

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}
doo k = N  

______________

______________

k := k + 1  {P}

od  {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

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

In the above template this relation becomes

\[ R \equiv (k = N) \quad \text{and} \quad P \]

It took some time to realize that this relation can be used to derive P from R. The idea behind this was shown by the work of David Gries and the Programming Science Group at the University of Eindhoven in Holland.

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 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.

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 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 (and clause)
(iii) introduce a disjunct (or clause)
(iv) introduce a new variable

For example, the postcondition for Problem 1 is

\[ R : (A_i : 0 \leq i < N : \max \geq B[i]) \]

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

\[ P : (A_i : 0 \leq i < k : \max \geq [i]) \]

so that

\[ R \equiv (k = N) \quad \text{and} \quad P \]

and the guard is indeed

\[ BB : k \neq N \]

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

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

or writing this as a multiple assignment

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

The program template for this problem therefore becomes

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

\[
\begin{align*}
do \quad k \neq N & \quad \{P\} \\
\quad k &:= k+1 \quad \{P\}
\end{align*}
\]

\[
\begin{align*}
ad \quad \{R \equiv (k = N) \quad \text{and} \quad P\}
\end{align*}
\]

where

\[
\begin{align*}
Q & : B[0..N-1] \text{ is a fixed array of integers and } N > 0 \\
P & : (A_i : 0 \leq i < k : \max \geq B[i]) \\
R & : (A_i : 0 \leq i < N : \max \geq B[i])
\end{align*}
\]
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 general 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 Pacal

```pascal
var
  k : integer;
begin
  k := 5;
...```
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

```pascal
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

(ii) a particular value stored at the location k may be denoted by

(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

```pascal
{ P and (k = k') }

k := k+1
{ P and k = k'' = k'+1 }
```

od
Another way to say the same is to express the dependence of \( P \) upon \( k \) in functional notation to say that after each execution of the loop body we have

\[
\{ P(k) \equiv P(k') \}
\]
\[
do \ k \neq N \rightarrow \\
:\{ P(k'+1) \}
\]
\[
k := k + 1
\]
\[
\{ P(k) \equiv P(k'') \}
\]

where \( k'' = k' + 1 \) so that before the loop starts execution of the loop body for the current iteration \( P(k) \) is true with the value \( k = k' \) while after the iteration is completed \( P(k) \) is again 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 \( P(k' + 1) \) is \textit{true} so that the new value \( k'' = k' + 1 \) can be stored in \( k \) by the last assignment statement of the loop to establish the truth of \( P(k) \) for the new value \( k = k'' \) of \( k \).

There is a simple way to determine what statements are required to establish the truth of \( P \) for \( k'' \), the new value of \( k \). The method is to form \( P(k'') \equiv P(k' + 1) \) 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 \( P(k') \) is true before the start of the iteration, a large part of the new predicate \( P(k' + 1) \) will be \textit{true} already. In Problem 1

\[
P: (Ai : 0 \leq i < k : \max \geq B[i])
\]

so that

\[
P(k') : (Ai : 0 \leq i < k' : \max \geq B[i])
\]

and

\[
P(k' + 1) : (Ai : 0 \leq i < k' + 1 : \max \geq B[i])
\]

By splitting off the last element we get

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

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

since \( P(k') \) already is \textit{true} before the iteration starts, the unspecified statements of the iteration have to establish

\[
\max \geq B[k']
\]

This is accomplished by the IF-Statement

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

\[
\text{fi}
\]
After the execution of the above IF-Statement, the truth of invariant $P(k)$ has been extended to include the value $k'' = k' + 1$. Therefore $P(k'')$ is true and after the assignment statement

$$k := k' + 1$$

which changes the value of $k$ from $k'$ to $k''$ we can say that

$P(k)$

is true for the new range of values. 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

$$\{ P(k) \text{ and } k = k' \}$$

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

$$\{ Q \}$$

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

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

```
procedure FindMax (n : integer);
  var
    k : integer;
  begin
    k := 1;
    max := B[0];
    while (k <> n) do
      begin
        if max < B[k] then max := B[k];
        k := k + 1
      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 \(R:\ sum = (\sum_{i=0}^{N-1} B[i])\)

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

\[P: \ sum = (\sum_{i=0}^{k-1} B[i])\]

and the postcondition \(R\) is

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

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

\[k, sum := 0, 0; \{P\}\]

The template program becomes

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

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

\[
P(k'+1) \equiv \ sum = (\sum_{i=0}^{k'+1} B[i])
\equiv \ sum = (\sum_{i=0}^{k'} B[i]) + B[k']
\equiv \ sum(k') + B[k']
\]

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

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

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

This can be translated into a Pascal procedure as follows

```
procedure SumArray;
var 
  k: integer;
beg
  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 \( Q : N \text{ is a fixed integer and } N \geq 0 \text{ and } a \geq 0 \)
Write a program to establish
\[ R : 0 \leq a^2 \leq N < (a + 1)^2 \]

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

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

9.1 Deleting a Conject

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

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

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

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

The program becomes
Progress towards termination of the loop is due to \( a := a + 1 \) so that for any particular iteration \( a = a' \) initially and \( a = a'' = a' + 1 \) after its completion. The missing statements are derived from

\[
\begin{align*}
P(a') &= 0 \leq (a')^2 \quad \text{and} \quad (a')^2 \leq N \\
P(a'+1) &= 0 \leq (a'+1)^2 \quad \text{and} \quad (a'+1)^2 \leq N
\end{align*}
\]

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

\[
\begin{align*}
a := 0; \\
\{ P \}
\end{align*}
\]

\[
\begin{align*}
do \quad N \geq (a+1) \cdot (a+1) \rightarrow a := a + 1 \\
\{ R \equiv P \quad \text{and} \quad N < (a+1)^2 \}
\end{align*}
\]

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 if 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 \( P \) for Problem 3 by the replacement of an expression by a variable so that from

\[
R : 0 \leq a^2 \leq N < (a+1)^2
\]

we obtain the invariant

\[
P : a^2 \leq N < b^2
\]

by replacing \((a+1)^2\) by \(b^2\) where we choose \( b \) to be positive so that

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

and 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:
The inequality \( a^2 \leq N \) is initialized by \( a := 0 \) because \( Q \) states that \( N \geq 0 \). The second inequality \( N < b^2 \) is true for any \( N \) if \( b \geq (N+1) \) where \( N = 0 \) is the worst case. \( P \) is established by the assignment statement

\[
a, b := 0, N+1
\]

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

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

The right condition \( N < b' \) is satisfied because \( P(a',b') \) is true but \( a' \) can be increased to \( a'' \) only if the left condition is satisfied. This is achieved by the guarded statement

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

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

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

provided that we can establish the truth of

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

In this case the left condition is satisfied because \( P(a',b') \) is true but \( b' \) can be increased to \( b'' \) only if the right condition is satisfied. This is achieved by the guarded statement

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

Combining these two guarded statements into one IF-statement we obtain the program
For efficiency reasons introduce a new variable \( d \) so that the midpoint of the range is calculated only once. The program becomes

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

\[
do b \neq (a+1) \rightarrow \quad \text{if} \quad ((a+b) \div 2)^2 \leq N \rightarrow a := (a+b) \div 2 \quad \{P\}
\]

\[
\quad \text{fi} \quad \text{if} \quad N < ((a+b) \div 2)^2 \rightarrow b := (a+b) \div 2 \quad \{P\}
\]

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

This program like binary search takes only about \( \log_2 N \) 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.

\[
\text{procedure} \quad \text{FastFindSqrt}; \quad \text{var} \quad \text{b, d : integer;}
\]

\[
\begin{align*}
& a := 0; \\
& b := N+1; \\
& \text{while } b <> (a+1) \text{ do} \\
& \quad \text{begin} \\
& \quad \quad d := (a+b) \div 2; \\
& \quad \quad \text{if } d \times d \leq N \text{ then } a := d \\
& \quad \quad \text{else } b := d \\
& \quad \text{end}
\end{align*}
\]

\[
\text{end;}
\]

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

This problem is different from the previous problems in that we first have to
specify a postcondition R. 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 replacement of the two constants 0
and N by variables i and j so that


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 = j + 1

and P still holds true then we have

A[i] ≤ 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] ≤ X < A[j] and j = (i+1)

so that the guard of the loop is

BB : j ≠ i+1

P is established by the assignment statement

i, j := 0, N

so that the program becomes

i, j := 0, N; {P}
do j ≠ i+1 →
   {P}
   {R ≡ P and 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
23.

\[ k := (i+j) \div 2 \]

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

\[ P(i = k') \equiv A[k'] \leq X < A[j'] \]

so that this yields the guarded statement

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

and the truth of

\[ P(j = k') \equiv A[i'] \leq X < A[k'] \]

requires the guarded statement

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

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

\[
\begin{align*}
i, j &:= 0, N; \{P\} \\
d o j \neq i+1 & \rightarrow k := (i+j) \div 2; \\
& \quad \text{if} \\
& \quad \; A[k] \leq X \rightarrow i := k \\
& \quad \quad \{X < A[k] \rightarrow j := k\} \\
& \quad \text{fi} \{P\} \\
\end{align*}
\]

\[ \text{od} \{ \text{R} \equiv \text{P and } j = i+1 \} \]

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

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

J. Reinfelds
9.9.85