FLEXIBLE ALGORITHMS - AN INTRODUCTION

R.B. Yehezkael (formerly Haskell)

Revised: July 2016 - סיון תשע"ו Minor changes: January 2017 - טבת תשע"ז

Department of Computer Sciences and Flexible Computation Research Laboratory
Jerusalem College of Technology - Machon Lev,
Havaad Haleumi 21, P.O.B. 16031, Jerusalem 91160

E-mail - rafi@g.jct.ac.il Home Page - http://homedir.jct.ac.il/~rafi

Contents

Preface

These course notes have been written for beginning undergraduate students of computing science and for suitable students in their final year of secondary school. It is based on a course in Hebrew "מבוא לאלגוריתמים גמישים", given at Jerusalem College of Technology, available at http://homedir.jct.ac.il/~rafi/algoflex.pdf.

First courses in computer science usually deal with sequential algorithms and programs. This has unfortunate consequences in that the student's mind becomes accustomed to a sequential way of thinking. This makes it hard for him to later understand and utilize parallelism effectively. Parallel programming on the other hand is very hard and so it is not practical to introduce these concepts at an early stage.

Our approach is to use flexible algorithms, which have a simple notation and are multifaceted. These flexible algorithms may be executed in a variety of orders, sequential or parallel, producing results independent of the execution order. They may be converted to sequential algorithms and to hardware block diagrams. They may also be used for giving an overall description of systems.

The educational approach of these notes is as follows.

Reading and executing flexible algorithms - understanding their behaviours.
Acquiring an early awareness of parallelism.
Converting flexible algorithms to hardware block diagrams (arouse curiosity).
Converting flexible algorithms to sequential algorithms (tail recursion).
Deeper understanding of flexible algorithms - computational induction.
Changes to flexible algorithms.
Writing flexible algorithms.
Overall description of systems using flexible algorithms (arouse curiosity).

This material can be given as the first part of an introductory course on computer science, and can be followed by material on sequential programming. A flexible algorithm would be used as a design for the program and would be converted (via a sequential algorithm) to a sequential program. Alternatively, the material can be taught in parallel to an introductory programming course.

Let me take this opportunity to thank those who have helped me in clarifying my ideas and preparing these course notes. Thanks to E. Dashtt and E. Gensburger who worked with me when developing the Hebrew course, which is the basis of these notes. Thanks and apologies to the students who had a hard time during the first year when this course was given - subsequently the course was well received. Thanks to H. G. Mendelbaum for the joint work on various papers we wrote together, which provided certain ideas for this work. Thanks too to G. Vidal Naquet and T. Hirst for various discussions on these and related ideas. Thanks to M. Goldstein and C. Levey for their helpful suggestions and encouraging comments. Thanks to S. Mizrahi for his clarifications regarding different kinds of processors and cores.

1. Introduction

So you do not know what an algorithm is, or the difference between a sequential and a flexible algorithm. Well read on.
Mr. Sequential Algorithm went to the supermarket equipped with the following shopping list, carefully written for him by his devoted wife.

12 eggs
2 jars of jam
4 potatoes
6 apples

Being true to his name, he was obliged to shop for all the items on the list, in the order that they were written. Strangely, that night Mr. Sequential Algorithm slept on the floor of the supermarket and was taken home by his weeping wife the next morning. What went wrong? Well, there were no eggs and so he waited and waited besides his empty trolley for the eggs which of course never arrived, and he was fortunate indeed that his noble wife saved him from starvation.
His brother Mr. Flexible Algorithm was also given the same shopping list by his devoted wife. Being true to his name, he was obliged to shop for all the items on the list, but he was not obliged to purchase them in the order that they were written. He himself decided in what order he would pick up the items, and he was sufficiently smart to pick up two items simultaneously.
Alas, even though he was somewhat smarter than his brother, he too slept on the floor of the supermarket that night. Again there were no eggs, and he just had to get everything on that list. So though he got everything else on the list, he too just waited and waited for the eggs which never arrived. The next morning, he too was saved from starvation by his noble wife.
And what about my wife you may ask? Well, as she has great wisdom and likes getting things done fast, fast, fast, she sorts the shopping list according to the order in which the food is to be found in the supermarket. She also instructs me to come back quickly and not to worry if an item of food is unavailable. (Didn't I tell you she likes getting things done fast, fast, fast.) Later, she asks me what I was unable to get.
Well, because of my wife's wisdom I do the shopping quickly and I am happy to tell you that I have never been locked in a supermarket. Thanks to my good wife, I do not worry about starvation.

Seriously now
Historically, programming has it roots based on sequential algorithms, and a sequential algorithm will wait doing nothing, if an item of data is unavailable. A flexible algorithm may do other things in the mean while, but would in the end also wait indefinitely if an item of data were unavailable. In such circumstances, programs based on both of these kinds algorithms would have to be stopped by the user when waiting indefinitely. Typically, they would also be stopped when the computer is turned off.
Though my wife's instructions to me are simple enough for us to perform, incorporating these instructions into algorithms is no easy task, and is not dealt with here. (Indeed they are not dealt with in first courses in algorithms and programming.) So for the sake of simplicity, we shall assume that indefinite waiting never occurs when executing sequential or flexible algorithms.

Sequential and flexible algorithms

From the previous story we learn the following.

A <u>sequential algorithm</u> is as a process that is executed sequentially in the order in which the steps are written.

A <u>flexible algorithm</u> is a process whose final results are uniquely defined but there may be several orders in which the steps may be executed, all of which give the same final results.

We mainly deal with flexible algorithms.

Marking exams as an example of a flexible algorithm

There are three questions on an exam. If there is no one to help, the teacher will have to mark all the exams, but he may do this in a variety of orders. He may mark an answer book at a time, or he may mark all the answers to question 1 then all the answers to question 2 then all the answers to questions 3. If he has two assistants, he may mark all the answers to question 1, one assistant may mark all the answers to question 2, and the other assistant may mark all the answers to questions 3.

The shopping list as an example of a flexible algorithm

There are many ways to buy the items on a shopping list at a supermarket. You may pick up the items in the order they are written on the list, or in the order you find them in the shop, etc.

Flexibility in the order of evaluating simple expressions

Simple arithmetic expressions may be evaluated in a variety of orders, all of which give the same final result. For example here are three of many orders for evaluating the expression (((9-7)/(6-5)) / ((5-3)/(2-1))).

Parallel evaluation of all innermost sub-expressions

(((9-7)/(6-5)) / ((5-3)/(2-1)))
= ((2/1) / (2/1))
= (2 / 2)
= 1

Sequential evaluation of the leftmost innermost sub-expression

(((9-7)/(6-5)) / ((5-3)/(2-1)))
= (((2/(6-5)) / ((5-3)/(2-1)))
= (((2/1) / ((5-3)/(2-1)))
= ((2 / ((5-3)/(2-1)))
= ((2 / (2/(2-1)))
= ((2 / (2/1))
= (2 / 2)
= 1

Sequential evaluation of the rightmost innermost sub-expression
(((9-7)/(6-5)) / ((5-3)/(2-1)))
= (((9-7)/(6-5)) / ((5-3)/1))
= (((9-7)/(6-5)) / (2/1))
= (((9-7)/(6-5)) / 2)
= (((9-7)/1) / 2)
= ((2/1) / 2)
= (2 / 2)
= 1

Note
The final results are the same for all three evaluations.

2. Flexible algorithms in a functional style and execution methods in sets/values form

We now explain via simple examples how flexible algorithms are written as functions, and how sets of statements are executed.
Three execution methods are presented:
1) Parallel execution.
2) Sequential execution with *immediate* execution of function calls or activations.
3) Sequential execution with *delayed* execution of function calls or activations.
All three execution methods will be presented in *sets/values form*, that is, as a sequence of "<u>sets of statements - values of results</u>".
Now here are some simple examples.
Note that in the following, comments start with "//".

Flexible algorithm for swapping two values

```
function a', b' •= swap (a, b);   // This is the function header which states that
                                  // the function receives values a, b
                                  // and returns results a', b'.

// SPECIFICATION:
// IN - a, b are any values..
// OUT - a' will get the value of b and b' will get the value of a.

// The set of statements to be executed follows.
{   a' •= b;    // this means assign (or copy) the value of b into a’
    b' •= a;    // this means assign (or copy) the value of a into b’
} // end swap
```

Notes
1) The "IN" part of the specification describes the values received by the "IN variables" of the function and the "OUT" part of the specification describes the results returned via the "OUT variables" of the function.
2) For the sake of clarity, OUT variables are tagged and IN variables are not tagged.
3) A statement such as a’ •= b; is called an assignment statement, whose purpose is to copy the value of the right hand side into the variable on the left hand side.

IMPORTANT
In a flexible algorithm, a variable may be assigned a value only once. This is a key requirement which enables the execution to be performed in a variety of orders with uniquely defined final results (flexible execution).

The three execution methods presented in sets/values form
Suppose that we wish to execute a', b' •= swap (1, 2);

Parallel execution

set of statements	a', b'
{ a', b' •= swap (1, 2); }	_, _
{ a' •= 2; b' •= 1; }	_, _
{ }	2, 1

Sequential execution left to right with immediate execution of function call at left

set of statements	a', b'
{ a', b' •= swap (1, 2); }	_, _
{ a' •= 2; b' •= 1; }	_, _
{ b' •= 1; }	2, _
{ }	2, 1

Sequential execution left to right with delayed execution of function call at left
In this case, this is identical to the previous sequential immediate execution method.

Note
The final results are the same for all three execution methods.

Flexible algorithm for reversing five values

```
function a', b', c', d', e' •= rev5(a, b, c, d, e);

// SPECIFICATION:
// IN - a, b, c, d, e are any values.
// OUT - values a', b', c', d', e' like a, b, c, d, e but in reverse order

{ a'•=e;
   b', c', d'•=rev3(b, c, d);
   e'•=a;
} // end rev5

function a', b', c' •= rev3(a, b, c);

// SPECIFICATION:
// IN - values a, b, c
// OUT - values a', b', c' like a, b, c but in reverse order

{ a'•=c;
   b' •=rev1(b);
   c'•=a;
} // end rev3

function a' •= rev1(a);

// SPECIFICATION:
// IN - value a
// OUT - value a' like a but in reverse order - that is a' is a

{ a'•=a;
} // end rev1
```

<u>The three execution methods presented in sets/values form</u>
Suppose we wish to execute the statement a', b', c', d', e' •= rev5 (1, 2, 3, 4, 5). We start the execution by writing this as a set of one element
{ a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }.
At the start, all the output variables a', b', c', d', e' have not received values, which is denoted by the symbol "_".

Parallel execution

set of statements	a', b', c', d', e'
{ a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }	_, _, _, _, _
{ a' •= 5 ; b', c', d' •= rev3(2, 3, 4) ; e' •= 1 }	_, _, _, _, _
{ b' •= 4, c' •= rev1(3) ; d' •= 2 }	5, _, _, _, 1
{ c' •= 3 }	5, 4, _, 2, 1
{ }	5, 4, 3, 2, 1

Sequential execution left to right with immediate execution of function call at left

set of statements	a', b', c', d', e'
{ a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }	_, _, _, _, _
{ a' •= 5 ; b', c', d' •= rev3(2, 3, 4) ; e' •= 1 }	_, _, _, _, _
{ b', c', d' •= rev3(2, 3, 4) ; e' •= 1 }	5, _, _, _, _
{ b' •= 4, c' •= rev1(3) ; d' •= 2 ; e' •= 1 }	5, _, _, _, _
{ c' •= rev1(3) ; d' •= 2 ; e' •= 1 }	5, 4, _, _, _
{ c' •= 3 ; d' •= 2 ; e' •= 1 }	5, 4, _, _, _
{ d' •= 2 ; e' •= 1 }	5, 4, 3, _, _
{ e' •= 1 }	5, 4, 3, 2, _
{ }	5, 4, 3, 2, 1

Sequential execution left to right with delayed execution of function call at left
So far we have not seen an example where the execution of the function calls at the left are delayed. Here this occurs in the two lines marked "#" below where there is a basic statement which can be immediately executed before the function call. In this case the basic statement is the assignment statement e'•=1 in the first line marked "#", and is the assignment statement d'•=2 in the second line marked "#".

set of statements	a', b', c', d', e'	
{ a', b', c', d', e' •= rev5(1, 2, 3, 4, 5) }	_, _, _, _, _	
{ a' •= 5 ; b', c', d' •= rev3(2, 3, 4) ; e' •= 1 }	_, _, _, _, _	
{ b', c', d' •= rev3(2, 3, 4); e' •= 1 }	5, _, _, _, _	#
{ b', c', d' •= rev3(2, 3, 4) }	5, _, _, _, 1	
{ b' •= 4, c' •= rev1(3) ; d' •= 2 }	5, _, _, _, 1	
{ c' •= rev1(3) ; d' •= 2 }	5, 4, _, _, 1	#
{ c' •= rev1(3) }	5, 4, _, 2, 1	
{ c' •= 3 }	5, 4, _, 2, 1	
{ }	5, 4, 3, 2, 1	

Note
The final results are the same for all three execution methods.

Exercises

1) Define functions rev2, rev4 including specifications, similar to the previous functions, for handling two and four values.
Present the execution of a', b', c', d' •= rev4 (2, 3, 4, 5) according to the three execution methods described previously.

2) Suppose that we change the definitions of the functions rev1, rev2, rev3 as follows.

```
function a', b', c', d', e' •= new_rev5(a, b, c, d, e);
{
  b', c', d'•= new_rev3(b, c, d);
  a'•=e;
  e'•=a;    // note that the two assignments are written together
}

function a', b', c' •= new_rev3(a, b, c);
{
  b' •= new_rev1(b);
  a'•=c;
  c'•=a;    // note that the two assignments are written together
}

function a' •= new_rev1(a);
{
  a'•=a;
}
```

Present the execution of a', b', c', d', e' •= new_rev5(1, 2, 3, 4, 5) according to the three execution methods described previously.

Flexible algorithm for reversing (part of) a vector

```
function v' •= reverse(v, low, high);

// SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.

{
if (low<high)
  {$v'_{high}$ •= $v_{low}$;
   v' •= reverse (v, low+1, high-1);
   $v'_{low}$ •= $v_{high}$;}
else if (low=high)
       {$v'_{high}$ •= $v_{high}$;};
} // end reverse
```

Note
The elements of vectors are numbered from zero, that is, v_0 is the first element of the vector v.

The three execution methods presented in sets/values form

Suppose that v=(1,2,3,4,5) and we wish to execute r' •= reverse (v,0,4). Here are the three executions. Note that the parallel execution is presented briefly and in full. The two sequential executions are presented only in brief.
In the brief form, we have not written the "if" statements, and have directly written the branch which applies.

Parallel execution (in brief)

set of statements	r'
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)
{ r'_4 •= v_0; r' •= reverse (v, 0+1, 4-1); r'_0 •= v_4; }	(_, _, _, _, _)
{ r' •= reverse (v, 1, 3); }	(5, _, _, _, 1)
{ r'_3 •= v_1; r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; }	(5, _, _, _, 1)
{ r' •= reverse (v, 2, 2); }	(5, 4, _, 2, 1)
{ r'_2 •= v_2; }	(5, 4, _, 2, 1)
{ }	(5, 4, 3, 2, 1)

Parallel execution (in full)

```
set of statements                                                r'
{ r' •= reverse (v, 0, 4); }                                     ( _, _, _, _, _ )
{
if (0<4)
   {v'_4 •= v_0;
     v' •= reverse (v, 0+1, 4-1);
     v'_0 •= v_4;}
else if (0=4)
        {v'_4 •= v_4;};
}                                                                ( _, _, _, _, _ )
{ r'_4 •= v_0; r' •= reverse (v, 0+1, 4-1); r'_0 •= v_4; }       ( _, _, _, _, _ )
{ r' •= reverse (v, 1, 3); }                                     ( 5, _, _, _, 1 )
{
if (1<3)
   {v'_3 •= v_1;
     v' •= reverse (v, 1+1, 3-1);
     v'_1 •= v_3;}
else if (1=3)
        {v'_3 •= v_3;};
}                                                                ( 5, _, _, _, 1 )
{ r'_3 •= v_1; r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; }       ( 5, _, _, _, 1 )
{ r' •= reverse (v, 2, 2); }                                     ( 5, 4, _, 2, 1 )
{
if (2<2)
   {v'_2 •= v_2;
     v' •= reverse (v, 2+1, 2-1);
     v'_2 •= v_2;}
else if (2=2)
        {v'_2 •= v_2;};
}                                                                ( 5, 4, _, 2, 2 )
{ r'_2 •= v_2; }                                                 ( 5, 4, _, 2, 1 )
{ }                                                              ( 5, 4, 3, 2, 1 )
```

Sequential execution left to right with immediate execution of function call at left (in brief)

set of statements	r'
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)
{ r'_4 •= v_0; r' •= reverse (v, 0+1, 4-1); r'_0 •= v_4; }	(_, _, _, _, _)
{ r' •= reverse (v, 0+1, 4-1); r'_0 •= v_4; }	(_, _, _, _, 1)
{ r' •= reverse (v, 1, 4-1); r'_0 •= v_4; }	(_, _, _, _, 1)
{ r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, 1)
{ r'_3 •= v_1; r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, _, 1)
{ r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r' •= reverse (v, 2, 3-1); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r' •= reverse (v, 2, 2); r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r'_2 •= v_2; r'_1 •= v_3; r'_0 •= v_4; }	(_, _, _, 2, 1)
{ r'_1 •= v_3; r'_0 •= v_4; }	(_, _, 3, 2, 1)
{ r'_0 •= v_4; }	(_, 4, 3, 2, 1)
{ }	(5, 4, 3, 2, 1)

Sequential execution left to right with delayed execution of function call at left (in brief)
Here again the execution of the function calls at the left are delayed in the two lines marked "#" below where there is a basic statement which can be immediately executed before the function call. (In this case the basic statement is an assignment statement)

set of statements	r'	
{ r' •= reverse (v, 0, 4); }	(_, _, _, _, _)	
{ r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }	(_, _, _, _, _)	
{ r' •= reverse (v, 0+1, 4-1); r'_0 •= v_4; }	(_, _, _, _, 1)	#
{ r' •= reverse (v, 0+1, 4-1); }	(5, _, _, _, 1)	
{ r' •= reverse (v, 1, 4-1); }	(5, _, _, _, 1)	
{ r' •= reverse (v, 1, 3); }	(5, _, _, _, 1)	
{ r'_3 •= v_1; r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; }	(5, _, _, _, 1)	
{ r' •= reverse (v, 1+1, 3-1); r'_1 •= v_3; }	(5, _, _, 2, 1)	#
{ r' •= reverse (v, 1+1, 3-1); }	(5, 4, _, 2, 1)	
{ r' •= reverse (v, 2, 3-1); }	(5, 4, _, 2, 1)	
{ r' •= reverse (v, 2, 2); }	(5, 4, _, 2, 1)	
{ r'_2 •= v_2;}	(5, 4, _, 2, 1)	
{ }	(5, 4, 3, 2, 1)	

Notes
1) The final results are the same for all three execution methods.
2) Even though the value of a variable may not be changed, one can create a new variable (having the same name) and give it a new value reflecting the change. This happens when the function is reactivated by the statement v' •= reverse (v, low+1, high-1). This causes a new variable low to get the value of the existing variable low plus one. Similarly a new variable high gets the value of the existing variable high minus one.
3) The requirement that a variable may be assigned a value only once is a key requirement which enables the execution to be performed in a variety of orders with uniquely defined final results (flexible execution).

Exercises

1) Suppose that v=(1,2,3,4,5,6,7) and we wish to execute the set of statements { v' •= reverse (v,0,3); v' •= reverse (v,4,6); }.
Present the execution according to the three execution methods, using the previous definition of "reverse".
(You should find that both the left and right halves of the vector have been reversed.)

2) Suppose the definition of "reverse" is changed as follows.

```
function v' •= reverse1(v, low, high);
{
if (low<high)
 {  $v'_{low}$ •= $v_{high}$;
    v' •= reverse1 (v, low+1, high-1);
    $v'_{high}$ •= $v_{low}$;}
else if (low=high)
        {$v'_{high}$ •= $v_{high}$;};
} // end reverse1
```

Suppose that v=(1,2,3,4,5). Present the execution of r'•=reverse1(v,0,4) according to the three execution methods.
Compare your executions with the executions of a', b', c', d', e' •= rev5 (1, 2, 3, 4, 5) given previously.

3) Suppose the definition of "reverse" is changed as follows.

```
function v' •= reverse2(v, low, high);
{
if (low<high)
   {
   v' •= reverse2 (v, low+1, high-1);
   $v'_{high}$ •= $v_{low}$; $v'_{low}$ •= $v_{high}$;    // Note that the two assignments are together.
   }
else if (low=high)
        {$v'_{high}$ •= $v_{high}$;};
} // end reverse2
```

Suppose that v=(1,2,3,4). Present the execution of r'•=reverse2(v,0,3) according to the three execution methods.

4) A palindrome is a word or character string which has the same reading forward and back. Define a function which receives a character string as a vector of characters and determines if the string is a palindrome, and returns "true" or "false" accordingly.

Flexible algorithm for the greatest common divisor

Here is Euclid's algorithm for determining the greatest common divisor of two positive integers. (Zero is not positive.)

```
function r' •= gcd(m,n)

// SPECIFICATION:
// IN - m, n are positive integers.
// OUT - the function returns in r' the greatest positive integer dividing both m and n.

{  if (m=n)
      {r'•=m};
   if (m>n)
      {r' •= gcd(m-n, n);};
   if (n>m)
      {r' •= gcd(m, n-m);};
} // end gcd
```

The three execution methods presented in sets/values form
Suppose that we wish to execute r' •= gcd(6,4).

Parallel execution

```
set of statements                                   r'
{ r' •= gcd(6,4) }                                  _
{  if (6=4)
      {r'•=6};
   if (6>4)
      {r' •= gcd(6-4, 4);};
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{ gcd(6-4, 4); }                                    _
{ r' •= gcd(2, 4 }                                  _
{  if (2=4)
      {r'•=2};
   if (2>4)
      {r' •= gcd(2-4, 4);};
   if (4>2)
      {r' •= gcd(2,4-2);};
}                                                   _
{ r' •= gcd(4-2, 2) }                               _
{ r' •= gcd(2, 2) }                                 _
{  if (2=2)
      {r'•=2};
   if (2>2)
      {r' •= gcd(2-2, 2);};
   if (2>2)
      {r' •= gcd(2, 2-2);};
}                                                   _
{ r' •= 2 }                                         _
{  }                                                2
```

Sequential execution left to right with immediate execution of function call at left

```
set of statements                                   r'
{ r' •= gcd(6,4) }                                  _
{  if (6=4)
      {r'•=6};
   if (6>4)
      {r' •= gcd(6-4, 4);};
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{  if (6>4)
      {r' •= gcd(6-4, 4);};
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{  r' •= gcd(6-4, 4);
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{  r' •= gcd(2, 4);
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{  if (2=4)
      {r'•=2};
   if (2>4)
      {r' •= gcd(2-4, 4);};
   if (4>2)
      {r' •= gcd(2,4-2);};
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
{  if (2>4)
      {r' •= gcd(2-4, 4);};
   if (4>2)
      {r' •= gcd(2,4-2);};
   if (4>6)
      {r' •= gcd(6,4-6);};
}                                                   _
```

```
{  if (4>2)
     {r' •= gcd(2,4-2);};
   if (4>6)
     {r' •= gcd(6,4-6);};
}                                                    –
{  r' •= gcd(2,4-2);
   if (4>6)
     {r' •= gcd(6,4-6);};
}
                                                     –
{  r' •= gcd(2,2);
   if (4>6)
     {r' •= gcd(6,4-6);};
}                                                    –
{  r' •= 2
   if (4>6)
     {r' •= gcd(6,4-6);};
}                                                    –
{ if (4>6)
     {r' •= gcd(6,4-6);};
}                                                    2
{  }                                                 2
```

Sequential execution left to right with delayed execution of function call at left
In this case, this is identical to the previous sequential immediate execution method.

Note
The final results are the same for all three execution methods.

Exercises

1) Improve the efficiency of the above algorithm by using the division "/" and remainder "%" operators. Here division mean the quotient or integer part of the division.
For example 4/3 is 1, 2/3 is 0, 8%3 is 2, 9%3 is 0.

2) The efficiency can be further improved by using an auxiliary function. Rewrite the algorithm using two functions only.

3) Here is a flexible algorithm for computing the squares of the elements of a vector v, giving the result in the vector v'.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
// SPECIFICATION: Like squares, but puts values only at places i onwards.
{ if (i< length of v)
     { loop(i•=i+1); $v'_i$•=$v_i \times v_i$; };
};   // end loop
```

Suppose that v=(-1, -2, 3, 4) and we wish to execute of v'•=square(v) by the parallel method. Suppose that a team consisting of a manager and four assistants performs the parallel execution. It takes the manager ten seconds to activate each stage (or line) of the execution. It takes an assistant one minute to calculate the square of a number and put it in the appropriate place of v'.
Present the parallel execution indicating the times of starting and completing each stage. Also indicate the total execution time.

4) Here is a flexible algorithm for copying (part of) a vector v into another vector v'.

```
function v'•=copy(v, low, high, index);

// SPECIFICATION:
// IN - "v" is a vector and "low" and "high" are places within the vector v
// where we assume low ≤ high.
// "index" is a place within the vector v'.
// OUT - From the place "index", v' is a copy of v in the range "low" to "high".
// Other elements of v' are not given values by the function.

{
  $v'_{index}$•=$v_{low}$;
   if( low < high )
     { copy(low•=low+1; index•=index+1); }
} // end copy
```

a) What happens if by mistake we execute this algorithm when initially low>high? (The algorithm is not designed to handle this case.)
b) Rewrite the algorithm so that it will carry out the copy even when low>high. (This means for example that executing v'•=copy(v, 4, 6, 2) produces the same results as executing v'•=copy(v, 6, 4, 2) and v_4, v_5, v_6 will be copied to v'_2, v'_3, v'_4 in both cases. Try writing this using one function only, and then more efficiently using two functions.)

Errors in flexible algorithms

There are two kinds of errors.
Syntax errors are errors in the form of the flexible algorithm and it is meaningless to execute such a flexible algorithm. You do not need to execute the flexible algorithm to see there is an error.
On the other hand there may be no error in the form or syntax of a flexible algorithm but an error may occur when it is executed. This is an execution time or run time error.

Syntax error (error in the form)

```
function x', y' •= bug1(x, y);
{
  x’•=3;          // No error here.
  x•=y+1;         // x may not be assigned here as it is an INPUT variable.
  y'•=x'+1;       // May not use the value of x’ here as it is an OUTPUT variable.
}
```

Other kinds of syntax errors are brackets which do not balance, errors in punctuation, etc.

Execution time error (run time error)

```
function x' •= bug2(x);    // No error in the form or syntax of the flexible algorithm.
{
  if (x ≤ 3)    x'•=x+1;
  if (x ≥ 3)    x'•=x+2;
}
```

If the value of x is 3 an error occurs during execution, as two assignments are made to x' (a conflict).

3. Different styles for passing parameters

The above definition of gcd, included within it two calls or activations of itself, which we wrote in style (1) below. We can equivalently write such calls in other styles.

1) Functional style (as before):	*r' •= gcd(m-n, n) ...* *r' •= gcd(m, n-m)*
2) Procedural style:	gcd(m-n, n, r') ... gcd(m, n-m, r')
3) Assignment style:	gcd(m•=m-n; n•=n; r'•=r') ... gcd(m•=m; n•=n-m; r'•=r')
4) Abbreviated assignment style: *(Only write the changes.)*	*gcd(m•=m-n) ...* *gcd(n•=n-m)*

In the following the functional and abbreviated assignment styles will be used.

In writing gcd(m•=m-n) ... gcd(n•=n-m) in the abbreviated assignment style, the values of m and n are not changed by the statements m•=m-n ... n•=n-m. This is because m on the right hand side denotes the current variable m, and m on the left hand side denotes the new variable m which will be used by the new activation of the function gcd. Similarly for the variable n. So there are separate variables for each call or activation of the function gcd.

Here is the definition of gcd in the abbreviated assignment style.

```
function r' •= gcd(m,n)

// SPECIFICATION:
// IN - m, n are positive integers.
// OUT - the function returns in r' the greatest positive integer dividing both m and n.

{  if (m=n)
      {r'•=m};
   if (m>n)
      {gcd(m•=m-n};
   if (n>m)
      {gcd(n•=n-m);};
} // end gcd
```

Exercises

1) Rewrite the function reverse given in chapter 2 in the abbreviated assignment style.

2) Rewrite the definition of rev1, rev3, rev5 given in chapter 2 in the abbreviated assignment style.

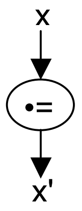
4. Converting flexible algorithms to hardware block diagrams

Reversing a fixed size vector

Here again is the flexible algorithm for reversing (part of) a vector.

```
function v' •= reverse(v, low, high);

// SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.

{
if (low<high)
  {$v'_{high}$ •= $v_{low}$;
    v' •= reverse (v, low+1, high-1);
    $v'_{low}$ •= $v_{high}$;}
else if (low=high)
        {$v'_{high}$ •= $v_{high}$;};
} // end reverse
```

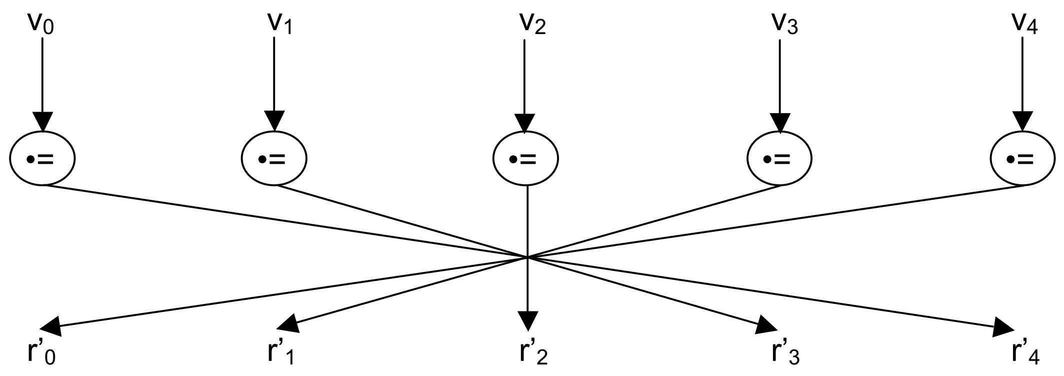
Let us say we have a circuit for performing an assignment x' •= x which is represented by the following diagram.



We can produce a block diagram of a circuit for reversing a vector of five elements from the set { r' •= reverse (v, 0, 4); } by symbolically executing the algorithm, that is by substituting using function definitions as in mathematics. The execution is shown as a sequence of equivalent sets of statements as follows:

{ r' •= reverse (v, 0, 4); }
≡ { r'_4 •= v_0; r' •= reverse (v, 1, 3); r'_0 •= v_4; }
≡ { r'_4 •= v_0; r'_3 •= v_1; reverse (v, 2, 2); r'_1 •= v_3; r'_0 •= v_4; }
≡ { r'_4 •= v_0; r'_3 •= v_1; r'_2 •= v_2; r'_1 •= v_3; r'_0 •= v_4; }

As the last line contains only operations which have circuits, it essentially describes the following block diagram for carrying out the reverse. (As no output is being generated, there is no need give the state of the output variables, which we gave in previous examples of actual computations.)



Of course this technique can be used for any constant size vector.

Adding two numbers and a carry

We assume that there is a function a3b for adding three digits producing two results a sum digit and a carry digit. For example, executing c' , s' •= a3b(9,3,1) would make s'•=3 and c'•=1. (Here s' is the sum and c' is the carry, each being a single digit.) Here is a specification of the function a3b.

```
function c', s' •= a3b(u, v, c)
// SPECIFICATION:
// IN - u, v, c, are the digits to be added.
// OUT - c' is the one digit carry and s' is the one digit sum.
```

Here is a function "add" for adding two numbers consisting of several digits and an existing carry (a single digit). The function add gives two results: a sum consisting of several digits and a carry consisting of one digit. It is assumed that both numbers being added and the sum produced consist of n+1 digits, where n denotes the position of the least significant digit. The most significant digit has position zero.

```
function c', s' •= add(u,v,c,n);

// SPECIFICATION:
// IN - "n" denotes the position of the least significant digits of u,v, to be added,
// where n≥0. The most significant digit has position zero.
// Digit "c" is also added at the position of the least significant digits of u, v.
// OUT - the carry of the addition is produced in c' and the sum itself in s'.

 {
  if (n>0)
  add( c, s'_n•= a3b(u_n, v_n, c); n•=n-1);
  else c', s'_0•=a3b(u_0, v_0, c);
 } // add
```

For example executing c',s'•=add(123,987,6,2) would make s'•=116 and c'•=1 .

Notes

1) In the above function, the occurrences of c on opposite sides of the "•=" denote different variables and similarly for n.
2) Though we have written this function based on decimal digits, the same definition would be used if we used binary digits (or bits). In the binary number system, executing c' , s' •= a3b(1,1,1) would make s'•=1 and c'•=1.
   Similarly, executing c',s'•=add(101,010,1,10) would make s'•=000 and c'•=1.

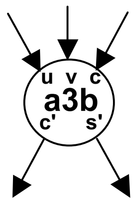
Here is an example of parallel execution of c',s' •= add(123, 987, 6, 2).

set of statements	c' s'
u v c n	
{ c',s' •= add(123, 987, 6, 2) }	_, _, _, _
{ c',s' •= add(123, 987, 1, 1) }	_, _, _, $\underline{6}$
{ c',s' •= add(123, 987, 1, 0) }	_, _, $\underline{1}$, $\underline{6}$
{ c', s'_0•=a3b(1, 9, 1) }	_, _, $\underline{1}$, $\underline{6}$
{ }	$\underline{1}$, $\underline{1}$, $\underline{1}$, $\underline{6}$

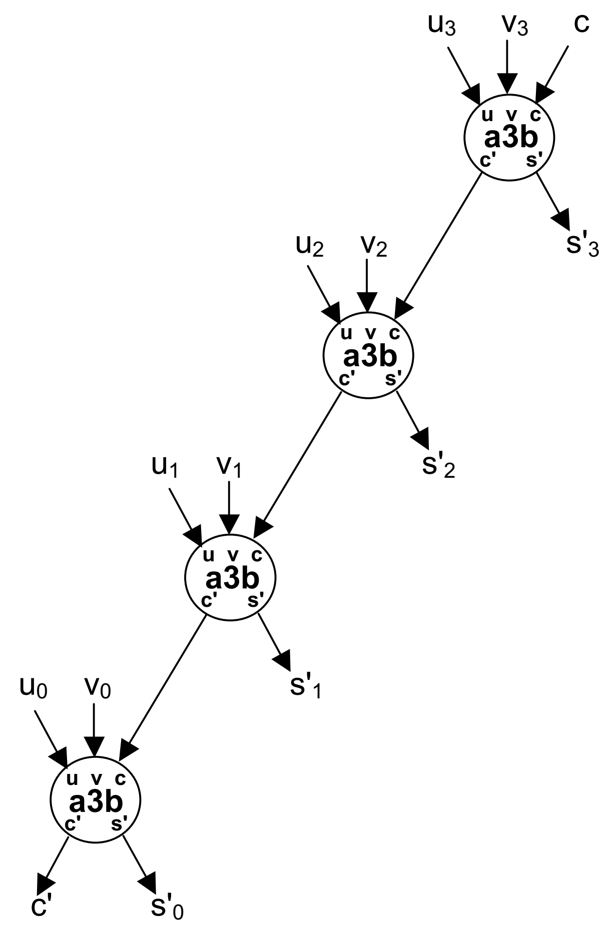
Let us now develop a block diagram of a circuit for a 4 digit serial adder.

{c',s'•=add(u,v,c,3);}
≡ {add(c,s'_3•=a3b(u_3,v_3,c);n•=2);}
≡ {add(c,s'_2•=a3b(u_2,v_2;c,s'_3•=a3b(u_3,v_3,c));n•=1);}
≡ {add(c,s'_1•=a3b(u_1,v_1;c,s'_2•=a3b(u_2,v_2;c,s'_3•=a3b(u_3,v_3,c)));n•=0);}
≡ {c',s'_0•=a3b(u_0,v_0;c,s'_1•=a3b(u_1,v_1;c,s'_2•=a3b(u_2,v_2;c,s'_3•=a3b(u_3,v_3,c))));}

The last line contains only primitive operations a3b. Let us represent a3b diagrammatically by:



The last line can now be represented by the diagram:

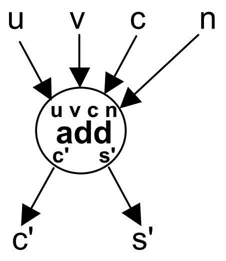


Note that this diagram represents a circuit for a serial adder, and this was obtained from the flexible algorithm by a process called symbolic execution, which is similar to the usual execution. Similar diagrams may be found in introductory books on digital logic and systems.

The above development was hard to follow.

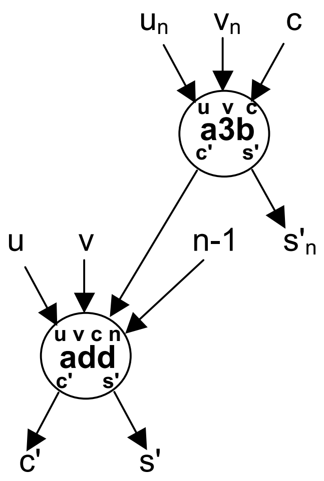
Here is a diagrammatic development that is much easier to understand.

A function call c', s'•=add(u, v, c, n) can be represented by:

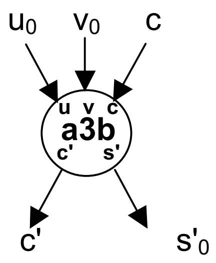


This diagram is equivalent to one of the following depending whether or not n>0.

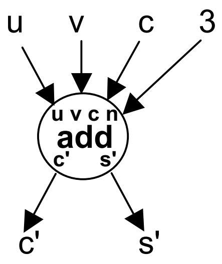
<u>Diagram for n>0</u>



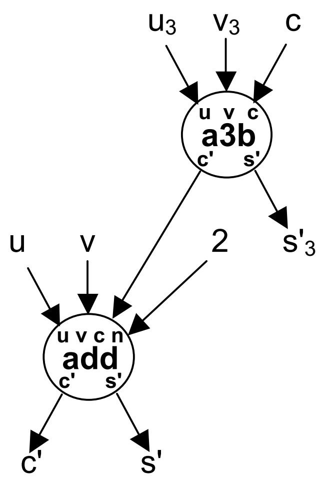
<u>Diagram for n=0</u>



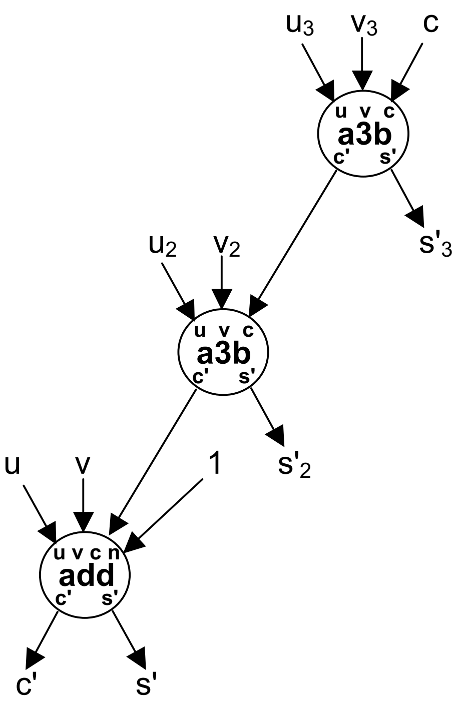
The call for a 4 digit adder is c', s'•=add(u, v, c, 3) and is represented by:



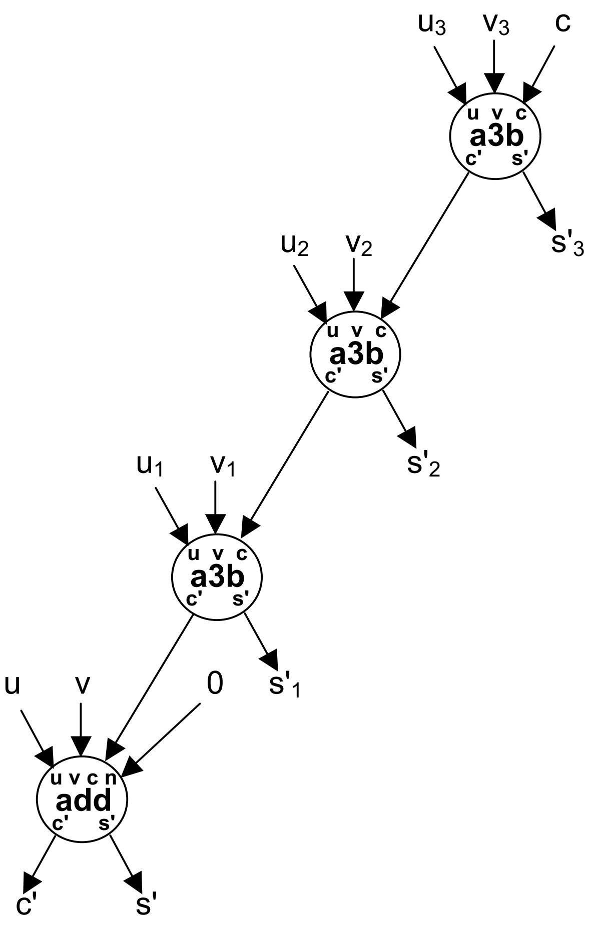
Equivalent to:



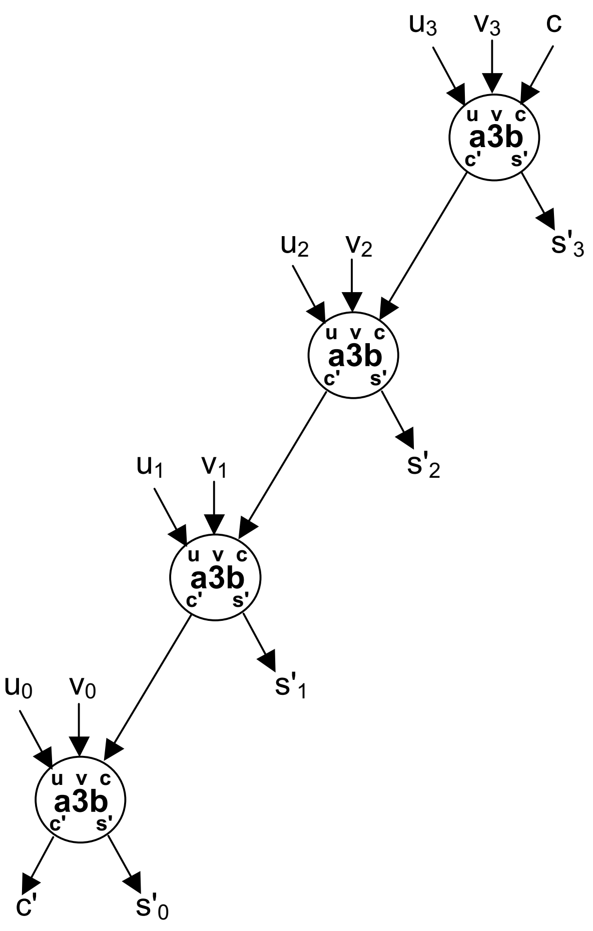
Equivalent to:



Equivalent to:



Finally giving



Here we have obtained diagrammatically, the same diagram for a serial adder.

5. Converting a flexible algorithm into a sequential algorithm

Unlike flexible algorithms, sequential algorithms allow changing the value of a variable. This has the advantage that less memory is required for executing such an algorithm. On the other hand, given a sequential algorithm, it is not easy to see what operations may be executed in parallel; as such an algorithm is optimized for sequential execution. A flexible algorithm should be viewed as a non specific description for carrying out the computation, and may be converted into a sequential algorithm. We describe the conversion of a flexible algorithm to a sequential one in three stages. (Strictly speaking the third stage is unnecessary but highly desirable, as it makes the algorithm easier to understand.)

Converting "reverse" to a sequential algorithm

The flexible algorithm should first be written in abbreviated assignment style, where in the function calls we only write the values changed. So here is the flexible algorithm "reverse" written in abbreviated assignment style.

```
function v' •= reverse(v, low, high);

// SPECIFICATION: As before.

{
if (low<high)
  { $v'_{high}$ •= $v_{low}$;
    reverse (low•=low+1, high•=high-1);
    $v'_{low}$ •= $v_{high}$;}
else if (low=high)
       {$v'_{high}$ •= $v_{high}$;};
} // end reverse
```

Notes

1) In the above, the assignment "low•=low+1", does not change the value of the variable "low". This is because "low" on the right hand side denotes the current variable "low", and "low" on the left hand side denotes the new variable "low" which will be used by the new activation of the function reverse. (Similarly for "high".)
2) The statement reverse (low•=low+1, high•=high-1) is equivalent to v' •= reverse (v, low+1, high-1).

Here is how this flexible algorithm may be converted into a sequential algorithm.

<u>*Stage 1*</u>

1) Delete the function header function v' •= reverse(v, low, high) and in its place write a label "reverse:".
2) Delete the tags from the output variables.
3) In place of reactivating the function by a function call, use a "goto" statement to the label "reverse".
4) Replace •= by :=. Note that •= denotes assignment used in flexible algorithms and does not allow the value of a variable to be changed. However := denotes

assignment used in sequential algorithms, and allows many assignments to the same variable and the value of a variable to be changed.

After performing these actions, the sequential algorithm, which is not yet correct, becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// (Additional temporary variables are used as will be seen later.)

reverse: { // Need temporary variable(s).
            if low<high
             {
               $v_{high}$:=$v_{low}$;
              low:=low+1; high:=high-1;
              goto reverse;
               $v_{low}$:=$v_{high}$    // Statement unreachable.
             }
            else    // Unnecessary and inefficient.
             if low=high
                 $v_{high}$:=$v_{high}$
          }
```

Stage 2
1) Change the order of the statements suitably.
2) Use temporary variables as needed.
3) Delete unnecessary parts of the algorithm.

After performing these actions, the sequential algorithm becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// Additional temporary variables new_vlow, new_vhigh are used below.

reverse: { if low<high
            {
              new_vlow:=$v_{high}$;    new_vhigh:=$v_{low}$;
              $v_{low}$:=new_vlow;    $v_{high}$:=new_vhigh;
              low:=low+1;    high:=high-1;
              goto reverse;
            }
          }
```

Stage 3
Where possible, use a while loop in place of "goto".

After doing this, the sequential algorithm becomes:

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// Additional temporary variables new_vlow, new_vhigh are used below.

while low<high
{
  new_vlow:=$v_{high}$;    new_vhigh:=$v_{low}$;
  $v_{low}$:=new_vlow;    $v_{high}$:=new_vhigh;
  low:=low+1;    high:=high-1;
}
```

Notes

1) The above may be executed sequentially, statement after statement. However, the statements on the same line could be automatically executed in parallel by a multi-scalar processor or core.
2) In general, suppose that we write a a flexible algorithm and convert it to a sequential algorithm as above. This will give more opportunities for multi-scalar processors or cores to automatically use parallelism on the derived sequential algorithm when compared to writing a sequential algorithm from the very start. This is because the new variables introduced as in the examples above, give more opportunities for parallel execution.
3) The assignment "low:=low+1", changes the value of the variable "low", it is increased by one. Here "low" on the both sides denote the same variable. (Similarly for "high" etc.)
4) The variables new_vlow, new_vhigh are temporary variables which enable the swap to be performed correctly. We can rewrite the previous sequential algorithm so that it uses only one temporary variable "temp" as follows.

```
// SPECIFICATION:
// As before, but changes are made to the vector v and the result is given in v itself.
// An additional temporary variable "temp" is used below.

while low<high
{
  temp:=$v_{low}$;
  $v_{low}$:=$v_{high}$;
  $v_{high}$:=temp;
  low:=low+1;  high:=high-1;
}
```

In the above, the three assignments "temp:=v_{low}; v_{low}:=v_{high} v_{high}:=temp;" must be performed sequentially as written. Though fewer variables are used, this may be slower in execution than the previous version of the *while* loop when executed on a multi-scalar processor or core. The use of more variables can enable more to be done in parallel. Thus for a <u>*non*</u> multi-scalar processor or core, this *while* loop is the better choice. For a multi-scalar processor or core, the previous *while* loop is the better choice.

Flexible and sequential algorithms for summation
Here is a flexible algorithm for summing the elements of a vector of numbers.

```
function s’ •= sum(v);
// SPECIFICATION:
// Input:    v is a vector of numbers.
// Output: s’ is the sum of the elements of the vector v.
//             s’ = $v_0 + v_1 + ... + v_{(\text{length of v-1})}$

{   sum1(i•=0;s•=0);    // equivalent to s'•=sum1(v, 0, 0)   } // end sum

function s’ •= sum1(v, i, s);
// SPECIFICATION:    s’ = $s + v_i + v_{i+1} ... + v_{(\text{length of v - 1})}$
// Note that when i ≥ length of v, this means that  s’•= s.
{
   if (i<length of v)
     {sum1(i•=i+1; s•=s+$v_i$)};    // equivalent to s'•=sum1(v, i+1, s+$v_i$)
   else
       { s’•=s; }
} // end sum1
```

If as explained earlier this is converted to a sequential algorithm where the result is given in s itself, we obtain:

```
i:=0; s:=0;
while i<length of v
{
   new_i:=i+1; new_s:=s+$v_i$
   i:=new_i; s:=new_s;
}
```

By being careful with the order in which we write the statements, there is no need for the extra variables new_i, new_s and we can write:

```
i:=0;s:=0;
while i<length of v
{
   s:=s+$v_i$;       // Note that reversing the order of these two statements
   i:=i+1;         // will cause an error in the computed value.
}
```

Exercises

1) Here is a flexible algorithm for computing the squares of the elements in a vector v, giving the result in the vector v'. Rewrite it as a sequential algorithm where the vector v is changed to hold the result.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
// SPECIFICATION: Like squares, but puts values only at places i onwards.
{ if (i< length of v)
     { loop(i•=i+1); $v'_i$•=$v_i \times v_i$; };
};    // end loop
```

2) Suppose that n is a positive integer. Here is a flexible algorithm for computing the sum of the integers from 0 to n. (If n is negative this sum is zero.)

```
function s'•=sigma(n);
{ loop(i•=0; s•=0);}    // end sigma

function s'•=loop(n, i, s);
{
  if i>n
     {s'•=s}
   else
      {loop(i•=i+1; s•=s+i);}
} // end loop
```

a) Write specifications for the functions sigma and loop.
b) Convert this flexible algorithm to a sequential algorithm.

3) Write a flexible algorithm, including specifications, for computing the alternating sum of the integers from 1 to n, that is, 1-2+3-4+...

Convert your flexible algorithm to a sequential algorithm.

4) Here is a flexible algorithm for copying (part of) a vector v into another vector v'.

```
function v'•=copy(v, low, high, index);

// SPECIFICATION:
// IN - "v" is a vector and "low" and "high" are places within the vector v
// where we assume low ≤ high.
// "index" is a place within the vector v'.
// OUT - From the place "index", v' is a copy of v in the range "low" to "high".
// Other elements of v' are not given values by the function.

{
   $v'_{index}$•=$v_{low}$;
    if( low < high )
      { copy(low•=low+1; index•=index+1); }
} // end copy
```

Convert this flexible algorithm into a sequential algorithm, where the result is given in the same vector v. Make sure your solution works properly in all the following cases: (i) index < low (ii) low ≤ index ≤ high (iii) high < index

An example using read and write statements
Here is a flexible algorithm for computing the squares of the elements of a vector v, giving the result in the vector v'.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
// SPECIFICATION: Like squares, but puts values only at places i onwards.
{ if (i < length of v)
    { loop(i•=i+1); $v'_i$•=$v_i \times v_i$; };
};    // end loop
```

Let us present several versions of a sequential algorithm taking into account the location of the values on the user's computer.

a) If we convert this to a sequential algorithm where the result is given in v' we obtain:

```
i:=0;
while i < length of v
    {$v'_i$:= $v_i \times v_i$; i:=i+1}
```

b) If we convert this to a sequential algorithm where the result is given in v itself we obtain:

```
i:=0;
while i < length of v
    {$v_i$:= $v_i \times v_i$; i:=i+1}
```

c) If we convert this to a sequential algorithm where the result is only to be displayed on the computer screen, we do not need the vector v'. We can therefore write:

```
i:=0;
while i < length of v
    {write($v_i \times v_i$); i:=i+1}
```

d) If we convert this to a sequential algorithm where the values of v are to be obtained from the keyboard, and the vector v is not needed thereafter, we do not need the vector v. We can therefore write:

```
i:=0;
while i < length of v'
      {read(value); $v'_i$:= value × value; i:=i+1}
```

e) If we convert this to a sequential algorithm where the values of v are to be obtained from the keyboard, and the vector v is not needed thereafter, and where the result is only to be displayed on the computer screen, we do not need the vectors v, v'. We do however need to know the number of values to be read. We can therefore write:

```
i:=0;
while i < number of values to be read
      {read(value); $v'_i$:= value × value; i:=i+1}
```

f) If we convert this to a sequential algorithm where the values of v are to be obtained from the keyboard, and the vector v is not needed thereafter, and where the result is only to be displayed on the computer screen, we do not need the vectors v, v'. If we do not know how many values are to be read but we do know when the data values are finished, we do not need the variable i. We can therefore write:

```
while data values are not finished
      {read(value); write(value × value)}
```

Notes

1) We have seen that the use of a sequential algorithm may result in the use of a simple variable instead of a vector, particularly with the use of read and write statements. (Also called input and output statements respectively.) This is an important advantage of sequential algorithms as less memory is needed for execution. However, this also restricts possibilities of parallel computation.
2) Flexible algorithms do not have read and write statements. They may require more memory for execution, but more can be done in parallel.
3) We do not recommend writing sequential algorithms as the first step in program development, as this will restrict the possibility of using parallelism. It is better to start with a flexible algorithm and then decide whether things should be done sequentially or in parallel.

6. The correctness of (flexible) algorithms

Executing an algorithm on some examples of input data gives some confidence that the algorithm works correctly, but this does not prove that the algorithm will work correctly on all possible input values. A proof technique is needed to prove that the algorithm works correctly in all cases. We would like to be able to prove that the algorithm always halts and gives correct results. Surprisingly, proof of halting is hard, and it has been shown by Turing that this can not be done by an algorithm, that is, no algorithmic methods for checking halting can exist.

We present a technique for proving partial correctness, meaning that if the execution halts, the results obtained agree with the specifications. (Note that the execution may not halt.) The technique is called computational induction and is based on the principle of complete induction below.

Induction (revision)
In the following the variables k, m, n denote non negative integers.
Suppose that p(n) is some property of n.
Here are two techniques for proof by induction.

Principle of simple induction
1) Prove that p(0).
2) Assume p(k) and prove that p(k+1) holds.
Conclusion: p(n) is true for every non negative integer n.

This is the usual technique which is taught in schools. However, it is not sufficiently powerful to be used as a basis for computational induction. The following more powerful technique can be used as the mathematical basis of computational induction.

Principle of complete induction
1) Prove that p(0).
2) Assume that p(0), p(1)…p(n-1) are true and prove that p(n) is true. (Equivalently assume that p(m) is true for m<n and prove that p(n) is true.)
Conclusion: p(n) is true for every non negative integer n.

Example of complete induction
Suppose that $S_0=2$ $S_1=5$ and,$S_n=5S_{n-1}-6S_{n-2}$ when n>1.
We claim that $S_n=2^n+3^n$ and this is the meaning we give to p(n).
When n=0, $2^0+3^0=2=S_0$ so p(0) is true.
When n=1, $2^1+3^1=5=S_1$ so p(1) is true.
Let us now deal with the case n>1.
We may assume that p(0), p(1)…p(n-1) are true, and so in particular we may assume that p(n-2), p(n-1) are true.
Therefore $S_{n-1}=2^{n-1}+3^{n-1}$ and $S_{n-2}=2^{n-2}+3^{n-2}$.
And so $S_n = 5S_{n-1}-6S_{n-2} = 5(2^{n-1}+3^{n-1})-6(2^{n-2}+3^{n-2})$
$= 10\times2^{n-2}-6\times2^{n-2}+15\times3^{n-2}-6\times3^{n-2}$
$= 4\times2^{n-2}+9\times3^{n-2} = 2^2\times2^{n-2}+3^2\times3^{n-2} = 2^n+3^n$.
Conclusion: p(n) is true for every non negative integer n, that is $S_n=2^n+3^n$.

Note that it is hard to use simple induction to prove this result in view of the need to assume that both p(n-1) and p(n-2) are true.

Exercise
Use simple induction to prove that $S_n=2^n+3^n$.
Hint: Use simple induction on p'(n) where p'(n) means (p(n) and p(n+1)), and, p(n) is as above.

Computational induction
This is a proof technique for proving the partial correctness of an algorithm, that is, if the execution halts then the results obtained are in agreement with the specifications. A specification is a precise definition of what the algorithm receives (IN values) and the results it gives (OUT values).

The structure of computational induction
1) Check that every result given explicitly agrees with the specifications.
2) Assume that every internal function call works according to specification and check that the final results the function gives agree with the specifications.
Conclusion: If the execution halts the results obtained are correct, that is, the algorithm is partially correct.

Partial correctness of reverse
Here again is the original definition of the function reverse.

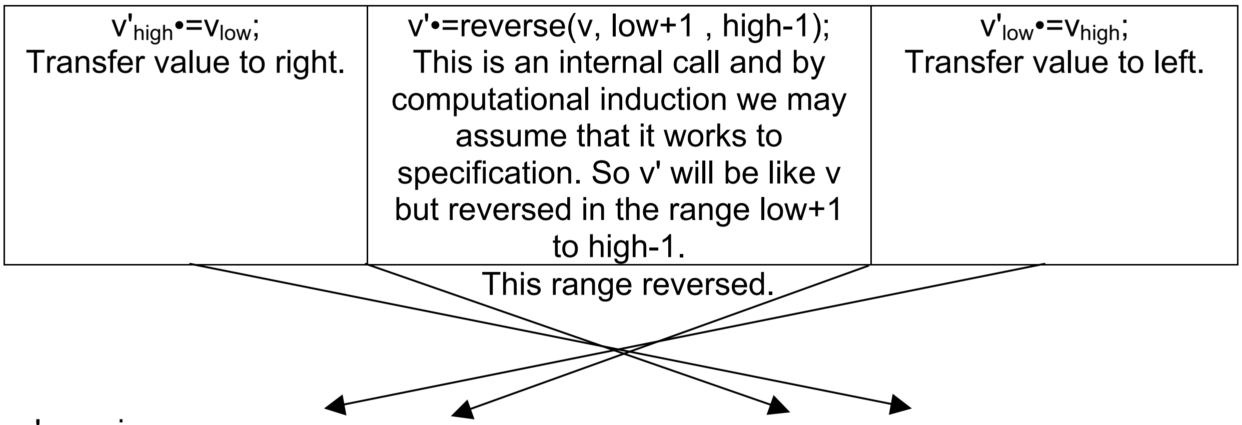
```
function v' •= reverse(v, low, high);

// SPECIFICATION:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - If low≤high, then within the range "low" to "high", v' is like v but reversed.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.

{
if (low<high)
  {v'_high •= v_low;
   v' •= reverse (v, low+1, high-1);
   v'_low •= v_high;}
else if (low=high)
       {v'_high •= v_high;};
} // end reverse
```

Let us use computational induction to show that it is partially correct. This allows us to assume that v' •= reverse(v, low+1, high-1) works to specification when proving that v' •= reverse(v, low, high) is partially correct.

Initially the vector v' holds no values.
_ , ··· , _ , _ , ··· , _ , ··· , _

Three cases need to be considered.
(a) low<high. (b) low=high. c) low>high.

a) low<high.
Here we need to execute:



v'_{high}•=v_{low};
Transfer value to right.

v'•=reverse(v, low+1 , high-1);
This is an internal call and by computational induction we may assume that it works to specification. So v' will be like v but reversed in the range low+1 to high-1.

v'_{low}•=v_{high};
Transfer value to left.

v' now is: .... v_{high}, v_{high-1}, ...v_{low+1}, v_{low} ....

This final value is in agreement with the specification of v'•=reverse(v,low,high).

b) low=high.
Here v_{high} is transferred to v'_{high} and v' becomes:
_ , ... v_{high} , _ , _ , _

This final value is also in agreement with the specification of v'•=reverse(v,low,high).

c) low>high.
Here no values are transferred to v' and this is in agreement with the specification of v'•=reverse(v,low,high).

So in all cases the final value of v' is in agreement with its specification, and "reverse" is partially correct. Q.E.D.

Notes
1) We assume that the internal call v'•=reverse(v,low+1,high-1) works correctly and we check that v'•=reverse(v,low,high) works correctly. We must not assume that v'•=reverse(v,low,high) works correctly.
2) Here is the connection with complete induction. Assuming halting, the number of steps m in v'•=reverse(v,low+1,high-1) is smaller the the number of steps n in v'•=reverse(v,low,high). This is because the execution of v'•=reverse(v,low+1,high-1) is part of the execution of v'•=reverse(v,low,high). So we may assume that v'•=reverse(v,low+1,high-1) works correctly according to its specification.
3) It must be remembered that the execution may not halt as this is not checked by computational induction. In this example however, the execution halts for all input values permitted by the specification, as the difference of the third parameter minus the second parameter is reduced in the internal call to reverse. Initially this difference is (high-low) and in the internal call it is (high-1)-(low+1)=high-low-2. Eventually this difference will become zero (low=high) or negative (low>high) where halting is immediate.

Showing that no variable is assigned more than once.
Here we use computational induction to show that no variable or component of a vector etc. is assigned more than once. Though this is not written explicitly in the specification, this is a basic requirement of all flexible algorithms. In effect this means we are checking that there are no write/write conflicts, which is important in parallel computing. To do this we use a stripped down specification, in which we do not describe the result produced but only which elements of v' (the OUT parameter) receive values.

```
// STRIPPED DOWN SPECIFICATION of the function reverse:
// IN - "v" is a vector and "low", "high" are places within the vector v.
// OUT - No element of v' is assigned more than once.
// If low≤high, then within the range "low" to "high", v' receives values.
// Other elements of v' are not given values by this function.
// If low>high, then the function does nothing to v'.
```

This will show that if the execution halts, no element of v' is assigned more than once; i.e. no write/write conflicts.

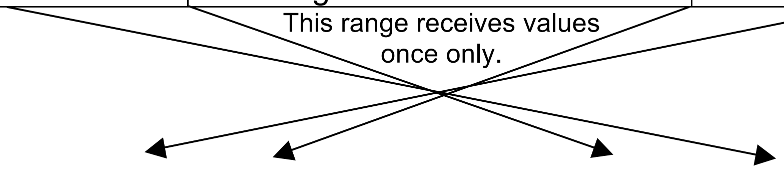
Initially the vector v' holds no values.
_ , ··· , _ , _ , ··· , _ , ··· , _

Three cases need to be considered.
(a) low<high. (b) low=high. c) low>high.

a) low<high.

Here we need to execute:

v'_{high}•=v_{low}; Transfer value to right.	v'•=reverse(v, low+1 , high-1); This is an internal call and by computational induction we may assume that it works to the stripped down specification. So v' will receive values in the range low+1 to high-1. Also, no element of v' in this range will be assigned more than once.	v'_{low}•=v_{high}; Transfer value to left.



v' now is: .... s_v, s_v, ... s_v, s_v ....
Here s_v denotes some value.
Clearly the entire range from low to high receives values. Also since low < high, the assignments in the left and right columns above do not cause assignments to an element of v' element more than once.
Thus this final value is in agreement with the stripped down specification of v'•=reverse(v,low,high).

b) low=high.
Here v_{high} is transferred to v'_{high} and v' becomes:
 _ , ... s_v , _ , _ , _
Clearly no element of v' in this range will be assigned more than once.
Thus this final value is also in agreement with the specification of
v'•=reverse(v,low,high).

c) low>high.
Here no values are transferred to v' and no element of v' in this range is assigned more than once. This is in agreement with the stripped down specification of v'•=reverse(v,low,high).

So in all cases the final value of v' is in agreement with the stripped down specification of v'•=reverse(v,low,high), and reverse is partially correct. This means that if the execution halts, no element of v' is assigned more than once; i.e. no write/write conflicts. Q.E.D.

Note
For educational reasons we have presented separate proofs for partial correctness of reverse with respect to its specification, and used a stripped down specification to show that no element of v' is assigned more than once. These proofs can of course be combined.

Exercise
Combine the two previous proofs.

Analyzing a function with an error - bugreverse
Consider now the following version of the reverse program, which includes an error.

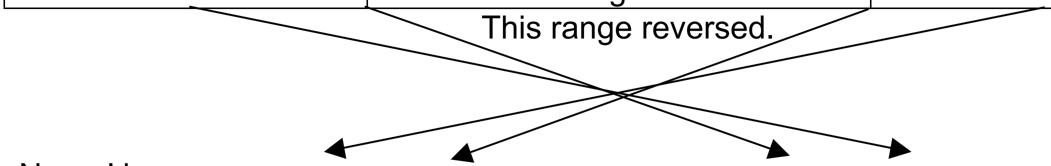
```
function v'•=bugreverse(v,low,high);

// SPECIFICATION: Same as reverse.

if (low<high)
{
  v'high•=vlow;
  v'•=bugreverse(v,low+2,high-2);    // ERROR HERE.
  v'low•=vhigh;
}
else
  if (low=high)
   { v'high•=vhigh; }
} // end bugreverse
```

Let us now use computational induction to show that there is an error.
Initially the vector v' is empty.

As before three cases need to be considered.
(a) low<high.   (b) low=high.   c) low>high.

a) low<high.
Here we need to execute:

v'_{high}•=v_{low}; Transfer value to right.	v'•=reverse(v, low+2 , high-2); This is an internal call and by computational induction we may assume that it works to specification. So the v' will be like v reversed in the range low+1 to high.	v'_{low}•=v_{high}; Transfer value to left.



Now v' is: .... v_{high}, _, v_{high-2}, ...v_{low+2}, _, v_{low} ....
and we see the range of values from low to high is not reversed.
There is an error.

Note
The value v' gets in the analysis by computational induction may not be the value the function gives v'. This is because in executing the function, the error may be repeated many times. However, in the computational induction analysis the error occurs only once, as we assumed that the internal function call works correctly.
So by computational induction v' gets the value:
.... v_{high}, _, v_{high-2}, v_{high-3}, v_{high-4}, ... v_{low+4}, v_{low+3}, v_{low+2}, _, v_{low} ....
However by execution v' gets the value:
.... v_{high}, _, v_{high-2}, _, v_{high-4}, ... v_{low+4}, _, v_{low+2}, _, v_{low} ....
More generally we can say the following:
1) If the function is correct the value v' gets by computational induction assuming halting and by execution are identical to that defined in the specification.
2) If the function is incorrect, then in some execution, the values v' gets by execution and by computational induction will be different from that defined in the specification.
More precisely, the values v' get by execution and by computational induction may or may not be equal, but both will differ from that defined in the specification.

Exercise
Suppose we execute v'•=bugreverse(v, low, high).

a) Find values for v, low, high for which the values v' gets by computational induction assuming halting and by execution are identical.
b) Find values for v, low, high for which the values v' gets by computational induction assuming halting and by execution are different.

Partial correctness of sum and sum1
Here again are the functions sum and sum1.

```
function s' •= sum(v);
// SPECIFICATION:
// Input:   v is a vector of numbers.
// Output: s' is the sum of the elements of the vector v.
//            s' = v0 + v1 +... + v (length of v-1)
```

```
{
  sum1(i•=0; s•=0);    // equivalent to s'•=sum1(v, 0, 0)
} // end sum

function s' •= sum1(v, i, s);
// SPECIFICATION:    s' = s + v_i + v_{i+1} ... + v_(length of v - 1)
// Note that when i ≥ length of v, this means that  s'•= s.
{
  if (i<length of v)
    {sum1(i•=i+1; s•=s+v_i)};    // equivalent to s'•=sum1(v, i+1, s+v_i)
  else
     { s'•=s; }
} // end sum1
```

Let us now show by computational induction that the functions sum and sum1 are partially correct. This allows us to assume that s'•=sum1(v, 0, 0) works to specification when proving that s' •= sum(v) is partially correct. This also allows us to assume that s'•=sum1(v, i+1, s+v_i) works to specification when proving that s' •= sum1(v, i, s) is partially correct.

The function sum

Here we execute sum1(i•=0; s•=0) or equivalently s'•=sum1(v, 0, 0).
By computational induction we may assume that this internal function call works to specification and so by substituting 0 for s and 0 for i in the specification of sum1 we get $s' = 0 + v_0 + \ldots v_{\text{length of v-1}} = v_0 + v_1 + \ldots + v_{(\text{length of v-1})}$.
Clearly this agrees with the specification of of s'•=sum(v).
So assuming that the execution halts the results will agree with the specification. In other words, the function sum is partially correct.

The function sum1

Two cases need to be considered.   (a) i < length of v.   (b) i ≥ length of v.

(a) i < length of v.
Here we execute sum1(i•=i+1;s•=s+v_i) or equivalently s'•=sum1(v, i+1, s+v_i).
By computational induction we may assume that this internal function call works to specification and so by substituting (s + v_i) for s and i+1 for i in the specification of sum1 we get $s' = (s + v_i) + v_{i+1} + \ldots\ldots\ldots + v_{\text{length of v-1}}$.
Clearly this agrees with the specification of s'•=sum1(v, i, s).

(b) i ≥ length of v.
Here s'=s and clearly this agrees with the specification of s'•=sum1(v, i, s).

So assuming that the execution halts the results will agree with the specification. In other words, the function sum1 is partially correct.

Analyzing a function with an error - bugsum1
Consider the following version of sum1 in which there is an error.

```
function s'•=bugsum1(v,i,s);
// SPECIFICATION: Same as sum1.
{
 if i < length of v
   { bugsum1 (i•=i+1; s•=i+$v_i$) }   // ERROR – should be s•=s+$v_i$ not s•=i+$v_i$.
 else
   { s'•=s; }
}
```

Two cases need to be considered. (a) i < length of v. (b) i ≥ length of v.

(a) i < length of v.
Here we execute bugsum1(i•=i+1;i•=s+v_i) or equivalently s'•=bugsum1(v, i+1, i+v_i). By computational induction we may assume that this internal function call works to specification and so by substituting (i + v_i) for s and i+1 for i in the specification of bugsum1 we get $s' = \underbrace{(i + v_i)}_{\boxed{\text{Error, this should be s.}}} + v_{i+1} + \ldots\ldots\ldots + v_{\text{length of v-1}}$.

Some general comments
1) Computational induction is based on complete induction where n is the number of steps in the execution, and p(n) means that the algorithm works according to specification if there are n steps in the execution. Intuitively speaking, assuming halting, the number of steps m in executing an internal function call is less than the number of steps n in obtaining the final results. So m<n and by complete induction we can assume p(m), that is, the internal function calls works to specification.
2) The strength of computational induction is that it handles any type of data as induction is not performed on the data but on the number of steps of the execution. It also checks all possible input values. The weakness of this technique is that it does not check halting.
3) We will not present formal mathematical techniques for proving halting.

Exercises

1) Consider the function:

```
function s'•=sq1(n,s);
// SPECIFICATION: Here n is an integer and s' = $n^2$+s when n>0
//                otherwise s' = s.
{
if n > 0
  {sq1(n•=n-1; s•=s+2×n+1);}
else
  {s'•=s};
}
```

a) Use computational induction to check this function for partial correctness. (You should find there is an error.)
b) Correct the error.
c) Check the corrected function for partial correctness.

2) Here again is the flexible algorithm for reversing five values. Use computational induction to check the function rev5, rev3, rev1 for partial correctness.

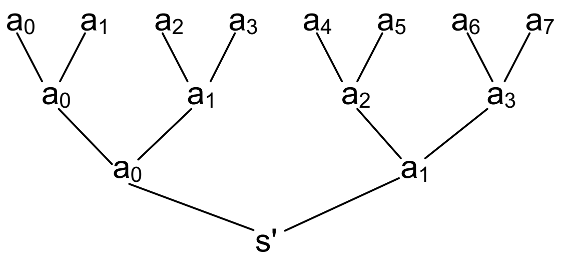
```
function a', b', c', d', e' •= rev5(a, b, c, d, e);

// SPECIFICATION:
// IN - a, b, c, d, e are any values.
// OUT - values a', b', c', d', e' like a, b, c, d, e but in reverse order

{  a'•=e;
   b', c', d' •= rev3(b, c, d);
   e'•=a;
} // end rev5

function a', b', c' •= rev3(a, b, c);

// SPECIFICATION:
// IN - values a, b, c
// OUT - values a', b', c' like a, b, c but in reverse order

{  a'•=c;
   b' •= rev1(b);
   c'•=a;
} // end rev3

function a' •= rev1(a);

// SPECIFICATION:
// IN - value a
// OUT - value a' like a but in reverse order - that is a' is a

{ a'•=a;
} // end rev1
```

7. Various flexible algorithms

Summing eight numbers

The statement $s' \bullet= a_0 + a_1 + \ldots\ a_7$ is equivalent to $s' \bullet= ((\ldots(a_0 + a_1) + a_2) + a_3) + \ldots\ a_7)$ when fully bracketed. When bracketed in this way, sequential evaluation is forced.
Here is a different bracketing of the expression which enables parallel evaluation.
$s' \bullet= (((a_0 + a_1) + (a_2 + a_3)) + ((a_4 + a_5) + (a_6 + a_7)))$

Intuitively, here is how the calculation performed.



In the above, It is useful to view each occurrence of a_i as a separate variable. This view simplifies writing this calculation as a flexible algorithm, in which the computation of sub-expressions may be performed in parallel. So for example, there are three variables named a_0 etc. Here now are three functions add8, add4, add2 for carrying out the computation in this way.

```
function s' •= add8 ($a_0$, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$, $a_6$, $a_7$)
// SPECIFICATION: $s' = a_0 + a_1 + \ldots\ a_7$
{ add4 ($a_0$•=$a_0$+$a_1$; $a_1$•=$a_2$+$a_3$; $a_2$•=$a_4$+$a_5$; $a_3$•=$a_6$+$a_7$);}    // end add8

function s' •= add4($a_0$,$a_1$,$a_2$,$a_3$);
// SPECIFICATION: $s' = a_0 + a_1 + a_2 + a_3$
{add2($a_0$•=$a_0$+$a_1$; $a_1$•=$a_2$+$a_3$);} // end add4

function s' •= add2($a_0$, $a_1$);
// SPECIFICATION: $s' = a_0 + a_1$
{s'•=$a_0$+$a_1$}; // end add2
```

Here is the parallel execution of s'•=add8(1,2,3,4,-1,-2,-3,-4).

set of statements	*s'*
{s' •= add8(1,2,3,4,-1,-2,-3,-4)}	_
{s' •= add4(3,7,-3,-7)}	_
{s' •= add2(10,-10)}	_
{ }	0

Exercise
Show by computational induction that the functions add8, add4, add2 are partially correct.

Summing the elements of a vector of size n where n is a power of 2
Here is how the previous specific example can be generalized to handle a vector of n elements, where n is a power of 2.

```
function s'•=addn(v,n)
// SPECIFICATION: v is a vector and n is its size which must be a power of 2.
//                The function computes s' = $v_0 + v_1 ... + v_{n-1}$
{
if    (n=1)
      {s' •= $v_0$}
else addn (
              n•=n/2;
              $v_0$•=$v_0+v_1$;
              $v_1$•=$v_2+v_3$
              .
              .
              .
              $v_{n/2-1}$ •= $v_{n-2} + v_{n-1}$
          );
} // end addn
```

Here is the parallel execution of s' •= addn((1,2,3,4,-1,-2,-3,-4), 8).

set of statements	*s'*
{s' •= addn((1,2,3,4,-1,-2,-3,-4), 8)}	_
{s' •= addn((3,7,-3,-7), 4)}	_
{s' •= addn((10,-10), 2)}	_
{s' •= addn((0), 1)}	_
{ }	0

Exercises

1) Show by computational induction that the function addn is partially correct.

2) Write the following fragment as a function addpairs.

```
$v'_0$•=$v_0+v_1$;
$v'_1$•=$v_2+v_3$;
.
.                              v'•=addpairs(v, n)
.
.
$v'_{n/2-1}$•=$v_{n-2} + v_{n-1}$;
```

This allows us to rewrite "else addn (...)" as "else addn(n•=n-1; v•=addpairs(v, n))".

3) Rewrite the function addn, including the specification, to handle any vector size n where n≥1.

4) Define a function to determine the minimum value in a vector of n elements where n is a power of 2. Use the structure of addn in your solution.

Circular shift

Here is a flexible algorithm (two functions) for a left circular shift by one place.

```
function v’ •= lcs(v);      // Left Circular Shift

// SPECIFICATION:
// v, v' are vectors having the same size.
// The elements of v' are like v but moved one place to the left.
// However the element in place zero of v appears
// as the last element of v'

{
  $v'_{(\text{length of } v - 1)}$ •= $v_0$;
  moveleft(n•=1);
}

function v’ •= moveleft(v,n)

// SPECIFICATION:
// v, v' are vectors having the same size.
// The elements of v' are like v but moved one place to the left.
// Nothing is transferred to place zero of v'.

{
   if (n < length of v)
      {
           $v'_{n-1}$ •= $v_n$;
            moveleft(n•=n+1);
      }
}
```

Exercises

1) Convert the above to a sequential algorithm where the result is to be given in the vector v itself.

2) Rewrite the above functions so that zero does not need to treated as a special case. Hint: Use the remainder operator "%".

3) Rewrite the function lcs, including the specification, so that it receives an additional parameter indicating the number of places to be shifted left.

4) Write a function rcs, including the specification, for a Right Circular Shift by one place.

Merging two ordered vectors

Suppose we are given two ordered vectors, which are arranged in non-decreasing order. We wish to merge these two vectors into a similarly ordered vector holding the elements of both these given vectors.
For example merging (1,1,3,4,9) and (-1,3,5,7)) produces (-1,1,1,3,3,4,5,7,9).

```
function v’•=merge(v1,v2);

// SPECIFICATION:
// v1, v2 are vectors ordered in non decreased order.
// v' will be a similarly ordered vector holding the elements of v1 and v2.
// The length of v' must equal the sum of the lengths of v1 and v2.

{ loop (i•=0; j•=0; k•=0) }    // end merge

function v’•=loop(v1, v2, i, j, k)

// SPECIFICATION: As above, but the merge from place i in v1
// and from place j in v2 producing the result in v' from place k onwards.
// It is required that (length of v1 - i) + (length of v2 - j) = (length of v' - k).

{
  if (i<length of v1) and (j < length of v2)
     if ($v1_i \le v2_j$)
        {$v'_k$•=$v1_i$; loop(i•=i+1; k•=k+1);}
      else
        {$v'_k$•=$v2_j$; loop(j•=j+1; k•=k+1);}
  if (i = length of v1 ) and (j < length of v2)
     { $v'_k$•=$v2_j$; loop(j•=j+1; k•=k+1);}
  if (i < length of v1 ) and (j = length of v2)
     { $v'_k$•=$v1_i$; loop(i•=i+1; k•=k+1);}
} // end loop
```

Exercises

1) Present the parallel execution of v'•=merge((1,2,4),(3,5)).

2) Improve the efficiency of the above flexible algorithm, and state if you have improved it for parallel execution or for sequential execution or for both.

3) Convert the above to a sequential algorithm in which the result is produced in v'.

4) Rewrite the function "loop" above using the function "copy" with specification as follows.

```
function v'•=copy(v,low,high,index);
// SPECIFICATION:
// IN - "v" is a vector and "low" and "high" are places within the vector v
// where we assume low ≤ high.
// "index" is a place within the vector v'.
// OUT - From the place "index", v' is a copy of v in the range "low" to "high".
// Other elements of v' are not given values by the function.
```

5) Write a function merge1 (similar to merge) having specification as follows:

```
function v’ •= merge1(v,low1 ,high1, low2, high2, index);
// SPECIFICATION:
// v, v' are vectors.
// low1 ,high1, low2, high2 are places in v
// where low1 ≤ high1 and low2 ≤ high2.
// The ranges low1 to high1 and low2 to high2 do not overlap, that is,
// either high1 < low2 or high2 < low1.
// In the ranges low1 to high1 and low2 to high2 the elements
// of v are arranged in non decreasing order.
// index is a place in v'.
// v' will be a similarly ordered from the place "index" onwards and will
// only hold the values from v from the ranges low1 to high1 and low2 to high2.
// Other elements of v' are not given values by this function.
```

6) Convert the function merge1 you wrote in the previous question into a sequential algorithm.

Even-Odd Sort
We describe a sorting technique called "even-odd sort" which enables parallel execution. Pairs of values in even positions are compared and copied interchanged if necessary to a new vector. Then, pairs of values in odd positions are compared and copied interchanged if necessary to a new vector. This is repeated until a sorted vector is obtained. The sort is carried out with auxiliary functions epcs, opcs, is_sorted which are given later.

```
function v’•=sort(v)

// SPECIFICATION:
// v, v' are vectors having the same length.
// v' will be like v but sorted in non-decreasing order.

{
  if  (is_sorted(v))
     {v’•=v}
     else sort(v•=opcs(epcs(v)));
}; // end sort

function  b’•=is_sorted(v)

// SPECIFICATION:
// If v is sorted in non-decreasing order then b' will be true
// otherwise b' will be false.

{loop1(i•=0);};

function b’•=loop1(v,i)’
{
  if (i ≤ length of v-2)
    if ($v_i$ > $v_{i+1}$)
      { b’•=false }
    else
      { loop1 (i•=i+1) }
  else
    { b’•=true };
} // end loop1

function v’•=epcs(v);    // Even Pair Compare Swap

// SPECIFICATION:
// v' will be like v but when i is even and $v_i>v_{i+1}$ then $v'_{i+1}=v_i$ and $v'_i=v_{i+1}$.

{
 loop2(i•=0)
 if length of v is odd    // When length of v is odd, copy last element.
    {$v'_{\text{length of v} -1}$ •= $v_{\text{length of v-1}}$};
}; // end epcs
```

```
function v’•=loop2(v, i)
//SPECIFICATION:
// Like epcs but works form position i onwards where i is even.

{
 if (i ≤ length of v-2)
   {
     if ($v_i > v_{i+1}$)
       { $v'_i$•=$v_{i+1}$; $v'_{i+1}$•=$v_i$ }
     else
       { $v'_i$•=$v_i$; $v'_{i+1}$•=$v_{i+1}$ };
     loop2 (i•=i+2);
   }
} // end loop2
```

Here are the main steps of an example execution.

```
v'•=sort ( (5,4,3,2,1) )
                              epcs
                                     (4,5,2,3,1)
                              opcs
•=sort ( (4,2,5,1,3) )
                              epcs
                                     (2,4,1,5,3)
                              opcs
•=sort ( (2,1,4,3,5) )
                              epcs
                                     (1,2,3,4,5)
                              opcs
•=sort ( (1,2,3,4,5) )

•=(1,2,3,4,5)
```

Exercise
Write a function opcs "Odd Pair Compare Swap" for handling the interchanges in the odd positions. Take care to include the specification.

Improving the efficiency
The functions epcs and opcs can be modified so that they also return a boolean value (true/false value) indicating whether or not a swap has occured. If this is done, the efficiency can be significantly improved as follows.

```
function v’•=newsort(v)
{ aux1(  v1,b1•=newepcs(v) ) };
```

Note that in the following b1, b2 are boolean variables, and v1, v2 are auxiliary vectors.

```
function v’•=aux1(v1, b1);
{ aux2( v2,b2•=newopcs(v1) ) };
```

```
function v'•=aux2(b1, v2, b2);
{
   if(b1 or b2)
     {newsort(v•=v2)}
   else
     {v'•=v2};
}
```

Exercises

1) The above solution can be improved by only testing the value of b2. Why is this possible?

2) Write function definitions and specifications of newepcs and newopcs, so that they also return a boolean value indicating if an interchange has occurred.

Merge Sort

This technique sorts a vector by repeated merging. For simplicity we assume that the length of the vector is a power of 2. For example suppose we wish to sort a vector of 8 elements. We view this as 8 single elements, and of course each single element is sorted.
(8, 1 , 7, 2 , 6, 3 , 5, 4)
We merge pairs of single elements to obtain:
(1,8 , 2,7 , 3,6 , 4,5)
Now we have four sorted pairs, so we merge two and two pairs to obtain:
(1,2,7,8   ,   3,4,5,6)
Now we have two sorted runs of four elements, so we merge them to obtain:
(1,2,3,4,5,6,7,8)
Now we have a sorted vector.

Note that at each stage the size of the sorted run in the vector is doubled with respect to the previous stage. Also the number of sorted runs in the vector is halved with respect to the previous stage. Here is what happens to these values.

Number of sorted runs	*Size of sorted run*
8	1
4	2
2	4
1	8

Exercise
Write a function m2 with specification as follows.

```
function v'•=m2 (v, size, place);
// SPECIFICATION:
// v is a vector having two ranges which are sorted in non-decreasing order.
// These ranges are of length "size" and at position "place" onwards in v.
// These two ranges are merged into a single range of length "2×size"
// and are put at position "place" onwards in v'.
```

Let us now use the function m2 to define the function mergesort. The structure of mergesort is similar to the function addn we defined earlier.

```
function v’•=mergesort(v)
// SPECIFICATION:
// v, v' are vectors having the same length which must be a power of 2.
// v' will be like v but sorted in non-decreasing order.

{loop(size•=1);}

function v’•=loop(v, size)
// SPECIFICATION:
// v, v' are vectors having the same length which must be a power of 2.
// The vector v consists of consecutive sorted ranges of length "size"
// which must be a power of 2.
// v' will be like v but but sorted.
{
  if (size=length of v)
    {v’•=v}
   else
     loop (
              size•=size×2;
              v•=m2( v, size, 0 );
              v•=m2( v, size, 2×size );
              v•=m2( v, size, 4×size );
              v•=m2( v, size, 6×size );
              .
              .
              .
              v•=m2( v, size, length of v – (2×size) );
            );
 }
```

Exercises

1) Convert the above functions to a sequential algorithm.

2) Rewrite the function mergesort so that it will work with a vector of any size.
Hint: Use the function merge1 with specification as follows:

```
function v’ •= merge1(v,low1 ,high1, low2, high2, index);
// SPECIFICATION:
// v, v' are vectors.
// low1 ,high1, low2, high2 are places in v
// where low1 ≤ high1 and low2 ≤ high2.
// In the ranges low1 to high1 and low2 to high2 the elements
// of v are arranged in non decreasing order.
// index is a place in v'.
// v' will be a similarly ordered from the place "index" onwards and will
// only hold the values from v from the ranges low1 to high1 and low2 to high2.
// Other elements of v' are not given values by this function.
```

Binary Search

Here is a function for searching for a value in a sorted vector, arranged in non-decreasing order.

```
function place'•= bsearch(v, low, high, value)
// SPECIFICATION:
// v is a vector sorted in non-decreasing order in the range "low" to "high".
// If "value" is to be found in v in the range // "low" to "high", then
// place' will receive the position of the first occurrence of "value" in this range.
// If "value" is not to be found in this range or if low>high,
// then place' is given the value -1.
{
  if ( low>high )
    { place'•= -1; }    // code for not found
  else
    {   if ( value = $v_{(low + high)/2}$ )
          { place' •= (low + high)/2 }
        else
          if ( value < $v_{(low + high)/2}$ )
            { bsearch( high•=((low + high)/2)-1) ); }   // search in left half.
          else
            { bsearch( low•=((low + high)/2+1) ); }         // search in right half.
    }
  }
```

Exercises

1) Show using computational induction that the function bsearch is partially correct.

2) The function bsearch is not written efficiently as the value (low + high)/2) may be calculated several times. Use an auxiliary function for the part of the function following the first "else", so that this value is computed at most once.

3) Rewrite function bsearch without the last two "else"s but with an additional "if". Are there more possibilities for parallel execution in this version of the function? If so, what may be performed in parallel?

4) *[Hard]* Change the definition of the function bsearch so that it will work even if the range of values in v is unsorted. It is suggested you use an auxiliary function with two additional variables for holding the values of the two calls or activations of bsearch above.

8. Convenient abbreviations for writing functions

Unlabeled "let blocks"

Consider the function:

```
function r' •= f(x, y, z);
// SPECIFICATION: …
{
// statements of f
… x+y-z…
…
… x+y-z…… x+y-z…
…
}
```

The above is inefficient, as the value of x+y-z may be evaluated several times. It is more efficient (but less clear) if we write two functions as follows to avoid the repeated evaluation of x+y-z.

```
function r' •= new_f(x, y, z);
{
r'•=block(x, y, z, x+y-z);    // equivalent to   r'•=block(xyz•=x+y-z);
}

function r' •= block(x, y, z, xyz);
// The purpose of this function is to provide an extra variable xyz which
// receives the value x+y-z when activated by the previous function.

{ // statements of f with xyz in place of x+y-z
… xyz…
…
… xyz…… xyz…
…
}
```

It is shorter and more convenient to write the previous two functions in the following form, where the "let" is used to indicate the introduction of a local or internal variable and a hidden auxiliary function like the function "block" above.

```
function r' •= new_f(x,y,z);
{let xyz• = x+y-z;     // Give local or internal variable xyz its value.

// statements of f with xyz in place of x+y-z
… xyz…
…
… xyz…… xyz…
…
}
```

Labeled "let blocks"

Consider the following functions.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{ loop(i•=0); };    // end squares

function v'•=loop(v, i);
// SPECIFICATION: Like squares, but puts values only at places i onwards.
{ if (i< length of v)
    { loop(i•=i+1); v'_i•=v_i×v_i; };
};    // end loop
```

It is shorter and more convenient to write the previous function in the following form.

```
function v' •= new_squares (v);
{ let i•=0;  // Initial value of local variable i.
   loop:     // This is the label.
   if (i< length of v)
      {
         loop(i•=i+1);     // This activates or calls "loop" again.
         v'_i•=v_i×v_i;
      };
};    // new_squares
```

Note
The use of a label allows us to activate or call the "let block" again from within the "let block". It is a convenient abbreviation for introducing (hidden) internal functions and auxiliary variables.

Notes
1) "Let blocks" are briefer and clearer than auxiliary functions, and this an advantage when writing flexible algorithms (function definitions).
2) The simplest way of executing a function incorporating a "let block", is to rewrite it using auxiliary functions and then use any of the execution techniques described earlier.

Computational induction with "let blocks"
The simplest way of proving partial correctness of a function incorporating "let blocks", is to rewrite it using auxiliary functions and add specifications for the auxiliary functions. Then use computational induction to prove partial correctness. However, if we write a specification for the "let block", we can avoid making this conversion. It is important to write a general specification for the "let block" for any value of its local or internal variables as done in a function specification, and not just a specification for the initial values of these variables.

Here again are the functions sum, sum1 as we originally wrote them.

```
function s' •= sum(v);
// SPECIFICATION:
// Input:    v is a vector of numbers.
// Output: s' is the sum of the elements of the vector v.
//             s' = $v_0 + v_1 + ... + v_{(length\ of\ v-1)}$
{
  sum1(i•=0;s•=0);    // equivalent to s'•=sum1(v, 0, 0)
} // end sum

function s' •= sum1(v, i, s);
// SPECIFICATION:   s' = $s + v_i + v_{i+1} ... + v_{(length\ of\ v - 1)}$
// Note that when i ≥ length of v, this means that  s'•= s.
{
  if (i<length of v)
    {sum1(i•=i+1; s•=s+$v_i$)};    // equivalent to s'•=sum1(v, i+1, s+$v_i$)
  else
     { s'•=s; }
} // end sum1
```

Here now are functions sum, sum1, where we have modified sum1 so that it is a labelled "let block". Note that the specification of the labeled "let block" sum1 below is identical to the specification of the function sum1 above. These specifications define what will be obtained for any values of i, s, <u>and not just for their initial zero values</u>.

```
function s' •= sum(v);
// SPECIFICATION:
// Input:    v is a vector of numbers.
// Output: s' is the sum of the elements of the vector v.
//             s' = $v_0 + v_1 + ... + v_{(length\ of\ v-1)}$
{
let i•=0; let s•=0;    // initial values of i and s are zero.
sum1:
  // SPECIFICATION:   s'•= $s + v_i + v_{i+1} ... + v_{(length\ of\ v - 1)}$
  // Note that when i ≥ length of v, this means that s'•= s.
  if (i<length of v)
    {sum1(i•=i+1;s•=s+$v_i$)};    // equivalent to s'•=sum1(v, i+1, s+$v_i$)
  else
     { s'•=s; }
} // end sum
```

Let us now show by computational induction that the functions sum and the labeled "let block" sum1 are partially correct.

The function sum
Here we execute the labeled "let block" sum1 with i, s having values 0.
By computational induction we may assume that this labeled "let block" works to specification in this case.

So by substituting 0 for i and s in the specification of the labeled "let block" sum1 we get s' = 0 + v_0 + … $v_{\text{length of v-1}}$.
Clearly this agrees with the specification of of s'•=sum(v).
So assuming that the execution halts the results will agree with the specification. In other words, the function sum is partially correct.
(When sum1 is written as a function, this corresponds to assuming that the internal function call sum1(i•=0;s•=0) or equivalently s'•=sum1(v, 0, 0) works to specification.)

The labeled "let block" sum1
Two cases need to be considered. (a) i < length of v. (b) i ≥ length of v.

(a) i < length of v.
Here we execute sum1(i•=i+1;s•=s+v_i) or equivalently s'•=sum1(v, i+1, s+v_i).
By computational induction we may assume that this internal call to the labeled "let block" sum1 works to specification and so by substituting (s + v_i) for s and i+1 for i in the specification of sum1 we get s' = (s + v_i) + v_{i+1} + …….. + $v_{\text{length of v-1}}$.
Clearly this agrees with the specification of s'•=sum1(v, i, s).

(b) i ≥ length of v.
Here s'=s and clearly this agrees with the specification of s'•=sum1(v, i, s).

So assuming that the execution halts the results will agree with the specification. In other words, the function sum1 is partially correct.

Notes
1) The proof of sum1 here should be compared with the proof given in the chapter on computational induction. Not surprisingly they are very similar since "let blocks" are convenient abbreviations for functions.
2) Remember to take care to write a general specification for the "let block" for any value of its local or internal variables as done in a function specification, and not just a specification for the initial values of these variables.

A modified version of binary search
There is a hard exercise in the previous chapter to rewrite the function bsearch for binary search so that it would work even if the vector is not sorted. It is much easier to solve this problem using blocks and here is a solution.
Also, the value (low + high)/2 will be computed just once in this solution.

```
function place'•= new_bsearch(v, low, high, value)
// SPECIFICATION:
// v is a vector sorted in non-decreasing order in the range "low" to "high".
// If "value" is to be found in v in the range // "low" to "high", then
// place' will receive the position of the first occurrence of "value" in this range.
// If "value" is not to be found in this range or if low>high,
// then place' is given the value -1.
```

```
{
  if ( low>high )
    { place'•= -1; }    // code for not found
  else
    {  let middle•=(low + high)/2;
       if  ( value=v_middle )
          { place'•=middle }
        else
          {  let place1 •= new_bsearch (v,low, middle-1,value);      // search in left half
             let place2 •= new_bsearch (v, middle+1,high,value );  // search in right half
             if place1 ≠ -1
               { place'•=place1; }    // search in left half found something.
             else
               { place'•=place2; }    // use value from search in right half.
          }
    }
}
```

Exercises

1) In the function new_bsearch above, when value is found in the vector v, place' will hold the position of the first occurrence of value in the vector v. Modify the function so that place' will hold the position of the last occurrence of value in the vector v.

2) Rewrite the function new_bsearch above using two auxiliary functions instead of the two "let blocks".

"**Forall loops**"
This has the form: forall i•=m, n; { statements }
where m, n are integers. This is viewed equivalent to any of the following forms. Typically, m ≤ n. If m > n, nothing is executed.

// Form 1a - count up	// Form 1b - count down

```
{
  let i•=m;
  for:
  if i ≤ n
    { for(i•=i+1); statements };
}
```
	```
	{
	  let i•=n;
	  for:
	  if i ≥ m
	    { for(i•=i-1); statements };
	}
	```

`// Form 2 - parallel "let blocks"`
`//              assuming m ≤ n.`
 	
`{let i•=m; statements }`
`{let i•=m+1; statements }`
`   ...`
`   ...`
`{let i•=n-1; statements }`
`{let i•=n; statements }`
	Notes
	1) The "let blocks" may be written in any order.
	2) If m > n there are no "let blocks" as nothing is executed.

Regarding parallel "let blocks" in each "let block" there is a separate variable i holding a value in the range m to n.

Since "let blocks" are convenient abbreviations for functions, the "forall loop" can also be written as a set of calls to such a function as follows.

```
// Form 3 – use of an auxiliary function called "block".

{...•= block(...,m); ...•= block(...,m+1); ... ; ...•= block(...,n-1);  ...•= block(...,n); }

The function "block" has the following form.

function ...•= block(..., i); {statements};
```

Notes
Do not worry if you do not understand the following two notes, they have been written to make you a little aware of things you may meet in books and courses on parallel computers.
1) Form 2 is useful for execution on n computers with separate memories.
2) Form 3 is useful for execution on n computers with a shared memory.

Computational induction with "forall loops"
Computational induction may also be used to prove the correctness of a "forall loop", but in a much simpler way from which the correctness of a "let block" is proved. There is no need to write a specification for the "forall loop". Just prove what you can for a typical value of the index of the "forall loop", and then deduce that this will be true for all values of the index variable, in the range specified by the "forall loop".

Now for an example.

Here again is a flexible algorithm using a "forall loop" for computing the squares of the elements of a vector v, giving the result in the vector v'.

```
function v' •= squares (v);
// SPECIFICATION:
// v, v' are vectors having the same length.
// Each element of v' is the square of the corresponding element of v.
{
  forall i•=0, (length of v) - 1; { v'ᵢ•=vᵢ×vᵢ; };
};    // end squares
```

Here the proof of partial correctness is trivial. For a typical value of i, $v'_i=v_i\times v_i$. Therefore this holds for all values of i in the range 0 to (length of v) – 1. Therefore each element of v' is the square of the corresponding element of v.

Exercise
Here is a flexible algorithm for computing the sum of the squares of the elements of a vector v. A "let block" is used.

```
function s' •= sum_of_squares (v);
// SPECIFICATION:
// v, is a vector.
// s’ is the sum of the squares of the elements of v.
{
let i•=0; let s•=0;
// SPECIFICATION:  Please write this yourself.
//
//
//
//
loop:
   if (i< length of v)
      { loop(i•=i+1; s•=s+vᵢ×vᵢ) }
   else s'•=s ;
};    // end sum_of_squares
```

a) Please write the specification of the labeled "let block" loop.

b) Prove by computational induction that the function sum_of_squares and the labeled "let block" loop are partially correct.

Note
In the above exercise, it should not be possible to replace the labeled "let block" by a "forall loop". This is because the iterations (repetitions) of the "forall loop" are independent of each other and may be carried out in any order. This is not the case with a labeled "let block". You should try writing sum_of_squares with a "forall loop" instead of a labeled "let block" to understand why this is not possible.

Adding corresponding values in two tables (matrices)
Here is a flexible algorithm with two nested "forall loops" for adding corresponding values in two tables (matrices) a, b and giving the result in a third table (matrix) c'. All tables (matrices) have the same number of rows and the same number of columns. The position of the first element in a row or column is zero.

```
function c' •= add_tables(a, b);
SPECIFICATION: See explanation above.
{
  forall i•= 0, (number of rows) -1;
        { forall j•=0, (number of columns) -1; { $c'_{ij}$ •= $a_{ij} + b_{ij}$ }; };
};    // end add_tables
```

The function addn revisited
Here is again is the original function addn for summing the elements a vector of n elements, where n is a power of 2.

```
function s'•=addn(v,n)
// SPECIFICATION: v is a vector and n is its size which must be a power of 2.
//                The function computes s' = $v_0 + v_1 ... + v_{n-1}$
{
if   (n=1)
     {s' •= $v_0$}
else addn (
                n•=n/2;
                $v_0$•=$v_0+v_1$;
                $v_1$•=$v_2+v_3$
                .
                .
                .
                $v_{n/2-1}$ •= $v_{n-2} + v_{n-1}$
            );
} // end addn
```

Here it has been rewritten with a "forall loop".

```
function s'•=addn(v,n)
// SPECIFICATION: v is a vector and n is its size which must be a power of 2.
//                The function computes s' = $v_0 + v_1 ... + v_{n-1}$
{
if    (n=1)
       {s' •= $v_0$}
else addn (
                n•=n/2;
                forall i•=0, n/2-1; { $v_i$•=$v_{2i} + v_{2i+1}$; };
            );
} // end addn
```

Exercises

1) Use computational induction to prove that the version of the function addn with the "forall loop" is partially correct.

2) Here is a flexible algorithm using both "let blocks" and "forall loops" for calculating the sums of all the columns in a table (matrix).

```
function v' •= sum_columns(a);
// SPECIFICATION:
// a is a table (matrix) with m rows and n columns.
// v' is a vector of n values which will hold the sums of the n columns,
// that is $v'_j = a_{0,j} + a_{1,j} + … + a_{m-2,j} + a_{m-1,j}$.
// Note that the position of the first element in a row or column is zero.
{
 forall j•=0, n-1;
 { let i•=0; let sum•=0;
   loop:
   // SPECIFICATION:  Please write this yourself.
   //
   //
   //
   //
      if j < m
        { loop(i•=i+1; cs•=cs+$a_{ij}$ ); }
      else $v'_i$ •= cs;
 }
}    // end sum_columns
```

a) Please write the specification of the labeled "let block" loop.

b) Prove by computational induction that the function sum_columns and the labeled "let block" loop, are partially correct.

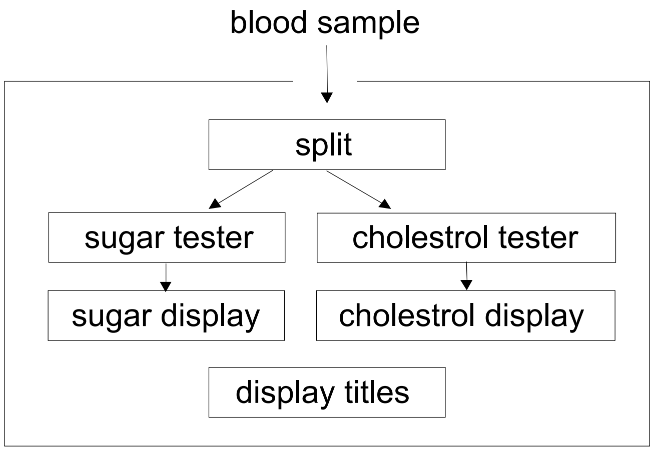
3) Here again are the definitions of the functions mergesort and loop where we have omitted the specifications. Rewrite them as a single function using a labelled "let block" and a "forall loop". To do the rewriting, you do not need to know anything about the function m2 used inside the function loop.

```
function v’•=mergesort(v)
{loop(size•=1);}

function v’•=loop(v, size)
{
  if (size=length of v)
    {v’•=v}
   else
      loop (
             size•=size×2,
             v•=m2(v,size,0);
             v•=m2(v,size,2×size);
             v•=m2(v,size,4×size);
             v•=m2(v,size,6×size);
             .
             .
             .
             v•=m2(v,size,length of v – (2×size)
           );
 }
```

9. Overall description of systems using flexible algorithms

A system is a collection of components which work together. Some of these components may work independently of each other, in parallel. Other components may need to be synchronized for them to work correctly. In view of this, sequential algorithms are restrictive for describing systems as they hide possibilities for doing things in parallel, and flexible algorithms are a better choice.
The task of designing (large) systems is complex and it is usual to give an overall description of the system and its components. This makes the task of designing easier. Let us now give an overall description of a blood test machine/system using a diagram and then using a flexible algorithm.

Diagrammatic description of a blood test machine
This simple blood test machine receives a sample of blood, measures the amount of sugar and cholesterol in the blood by two independent tests and displays the results. To do this it splits the blood sample into two smaller samples.

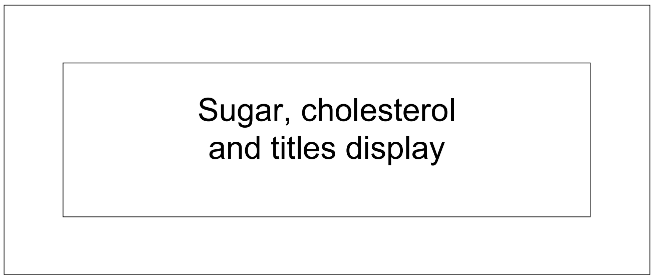


Flexible algorithm
The flexible algorithm approach allows several designs for building such a machine. Here then is the flexible algorithm for the blood tester.

```
function   title1', sugar', title2', cholesterol' •=
bloodtester(blood_sample);
{ let sample1, sample2 •= split(blood_sample);
  title1'•="SUGAR";
  sugar' •= sugartester(sample1);
  title2'•="CHOLESTEROL";
  cholesterol' •= cholesteroltester(sample2); }
}; // end bloodtester
```

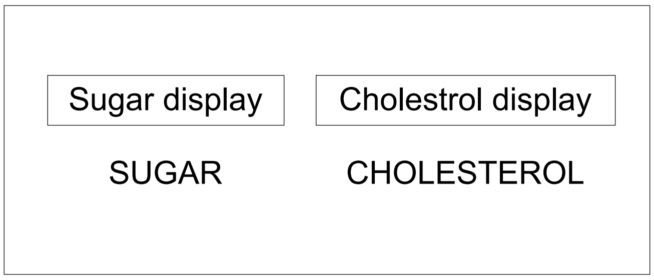
Here are three designs of the front panel of such a machine which are compatible with flexible execution. (Only the first design is compatible with sequential execution.)

First design of the front panel of such a machine



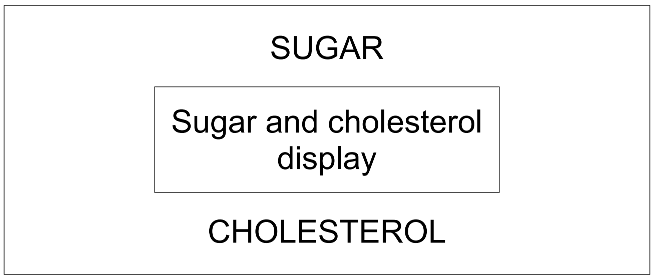
Here titles would be displayed when the algorithm is executed, and a larger (more expensive) display is needed. However, it is easy to adapt this design so that it can display the titles in more than one language.

Second design of the front panel of such a machine



Here the titles are "displayed" when the machine is built. Smaller (less expensive) displays are needed and perhaps less electricity is needed, which is important, particularly for a portable battery powered machine. However, it is not possible to adapt this design so that it can display the titles in more than one language.

Third design of the front panel of such a machine



This is a variation on the second design. It may be preferable to the previous design if the cost of a larger display is cheaper than the cost of two smaller displays, or if the larger display uses less electricity than two smaller displays.

Further reading

Here are details of books and articles on various topics in computer science.

Algorithms in a different functional style
"*Algorithms: a functional programming approach*", F. Rabhi and G. Lapalme, Addison-Wesley, 1999

Broad coverage of various topics in computing science
"*Algorithmics: The Spirit of Computing*", D. Harel (with Y. Feldman), Addison-Wesley, 3rd edition 2004

Hardware
"*Digital Design*", M. M. Mano, Prentice Hall, 3rd edition 2002

"*Digital Design Principles and Practices*", J. F. Wakerly, Prentice Hall, 4th edition 2005

Comprehensive coverage of sequential algorithms
"*Introduction to Algorithms*", T H. Cormen, C E. Leiserson, R L. Rivest and C Stein. MIT press and McGraw Hill, 2nd Edition 2001 and 2002

Advanced coverage of sequential and parallel algorithms
"*Algorithms sequential and parallel: a unified approach*", R. Miller and L. Boxer, Prentice Hall, 2000

Proving correctness
"*Program Verification*", N. Francez, Addison Wesley and Pearson Higher Education 1992

Relevant articles by the author on the web at http://homedir.jct.ac.il/~rafi
"Reasoning about Programs using Specifications and Induction", Internal Paper, Jerusalem College of Technology, revised 2004

"SIMPLE FLEXIBLE LANGUAGE - SFL", Presented to the Compiler Group Meeting, IBM Research Laboratories, Haifa, 3rd May 2000. Most recent version revised 2004