

USER'S GUIDE
BACI Pascal Compiler and
Concurrent PCODE Interpreter

Bill Bynum/Tracy Camp
College of William and Mary/Colorado School of Mines

July 14, 2003

Contents

1	Introduction	2
2	Pascal Compiler Syntax	2
3	Concurrency Constructs	4
3.1	COBEGIN—COEND	4
3.2	Semaphores	4
3.2.1	Initializing a Semaphore	4
3.2.2	P (or WAIT) and V (or SIGNAL) Procedures	5
3.2.3	Examples of Semaphore Usage	6
3.3	Monitors	7
3.3.1	Condition Variables	7
3.3.2	WAITC and SIGNALC Procedures	7
3.3.3	Immediate Resumption Requirement	8
3.3.4	An Example of a Monitor	8
3.4	Other Concurrency Constructs	8
3.4.1	ATOMIC Keyword	9
3.4.2	PROCEDURE suspend;	9
3.4.3	PROCEDURE revive(process_number : INTEGER);	9
3.4.4	FUNCTION which_proc : INTEGER;	9
3.4.5	FUNCTION random(range : INTEGER): INTEGER;	9
4	Built-in String Handling Functions	10
4.1	PROCEDURE stringCopy(dest : STRING; src : STRING);	10
4.2	PROCEDURE stringConcat(dest : STRING; src : STRING);	10
4.3	FUNCTION stringCompare(x : STRING; y : STRING): INTEGER;	10
4.4	FUNCTION stringLength(x : STRING): INTEGER;	10
4.5	FUNCTION sscanf(x : STRING; fmt : RAWSTRING ; ...): INTEGER;	10
4.6	PROCEDURE sprintf(x : STRING; x, fmt : RAWSTRING, ...);	11
5	Using the BACI Pascal Compiler and PCODE Interpreter	11
6	Sample program and output	12

1 Introduction

The purpose of this document is to provide a brief description of the BACI Pascal Compiler and Concurrent PCODE Interpreter programs and a description of how to use them. The compiler first compiles the user's program into an intermediate object code called PCODE, which the interpreter then executes. The Pascal compiler supports binary and counting semaphores and Hoare monitors. The interpreter simulates concurrent process execution.

Programs of the BACI System

program	function	described in
bacc	BACI C-- to PCODE Compiler	cmimi.ps
bapas	BACI Pascal to PCODE Compiler	this guide (guidepas.ps)
bainterp	command-line PCODE Interpreter	cmimi.ps, guidepas.ps
bagui	Graphical user interface to the PCODE Interpreter (UNIX systems only)	disasm.ps guiguide.ps
badis	PCODE de-compiler	disasm.ps
baar	PCODE archiver	sepcomp.ps
bald	PCODE linker	sepcomp.ps

The compiler and interpreter originally were procedures in a program written by M. Ben-Ari, based on the original Pascal compiler by Niklaus Wirth. The program source was included as an appendix in Ben-Ari's book, "Principles of Concurrent Programming". The original version of the BACI compiler and interpreter was created from that source code. Eventually, the compiler and interpreter were split into two separate programs, and a C-- compiler was developed to compile source programs written in a restricted dialect of C++ into PCODE object code executable by the interpreter.

The syntax for the Pascal compiler is explained below. This guide is applicable only to the BACI Pascal compiler and not to the BACI C-- compiler. Users interested in the C-- compiler should consult its user guide (see the file **cmimi.ps**).

2 Pascal Compiler Syntax

1. Comments can be delimited with `(* and *)` or `{ and }`, but the two styles cannot be mixed. C++ comments (beginning with `//`) are also accepted.
2. There are no files other than input and output. The program header must have the form:

```
PROGRAM program_name;
```

`READ`, `READLN`, and `EOLN` (all without a file parameter) behave as in standard Pascal. `WRITE` and `WRITELN`, also used without a file parameter, do not accept width specifiers for the output fields.

3. The only simple Pascal types are `INTEGER`, `BOOLEAN`, and `CHAR`. There are also other types related to the control of concurrency; these will be discussed below.
4. A `STRING` type is supported. To declare a string, the length of the string must be specified. The following declaration defines a string of length 20:

```
VAR
    string_name : STRING[20];
```

The length specifier should be the number of characters that the string should have and should not include space for the termination byte. The compiler takes care of reserving space for the termination byte. The length specifier must be either a literal constant or a program constant.

The `STRING` keyword is used in the declaration of a procedure or function for declaring a parameter of type `STRING`:

```
PROCEDURE proc(formal_parm : STRING);
```

This declaration asserts that `formal_parm` is of type `STRING[n]`, for some positive value of `n`. Parameters of type `STRING` are passed by reference. No check for string overrun is performed.

5. Arrays of any valid type are supported. Array declaration follows the usual Pascal syntax:

```
mytype = ARRAY[ LB..UB ] OF valid_type;
```

6. There are no subrange types and no `RECORD` types.
7. Constants of simple types are supported.
8. In the declaration of variables of `INTEGER` and `CHAR` types, initializers are supported. The value of the initializer must be a literal or program constant:

```
CONST
  m = 5;
VAR
  j : INTEGER := m;
  k : INTEGER := 3;
  c : CHAR := 'a';
```

9. Procedures and functions are supported with both `VAR` and non-`VAR` parameters. Standard scope rules apply. Recursion is supported.
10. The executable statements are `IF-THEN-ELSE`, `FOR`, `REPEAT-UNTIL`, and `WHILE-DO`. Statement bracketing with `BEGIN-END` is standard.
11. File inclusion, using the standard `C/C++` syntax, is supported:

```
#INCLUDE < ... >
#include " ... "
```

Both styles of `include` statement have the same semantics, because there is no “system” include directory.

12. The `EXTERNAL` keyword for defining external variables is supported, but its usage varies from that of Niklaus Wirth’s “standard” Pascal. In BACI Pascal, the `EXTERNAL` keyword precedes the definition of the external object, whereas in Wirth’s Pascal, the keyword trails the definition of the object.

Any an external variable can be of any valid BACI Pascal type. Initializers cannot be used with external variables. The `EXTERNAL` keyword can only occur at the global (“outer”) level, before the `PROGRAM` keyword has appeared, if the source file contains a main program block.

Typical examples:

```

EXTERNAL VAR
  i : INTEGER;
  a : ARRAY [1..20] OF CHAR;
  b : STRING[30];
// Initializers are not allowed ----> i : INTEGER := 30;
// (initialization, if present, must occur where i is defined)
EXTERNAL func(k : INTEGER) : INTEGER;
EXTERNAL MONITOR monSemaphore; // see Section 3. Only externally
  PROCEDURE monP(); // visible details of the monitor
  PROCEDURE monV(); // need be given here
END; // monSemaphore

```

The `-c` option must be used with `bapas` to compile source files that contain external references. See the *BACI System Separate Compilation Guide* for more information about the use of external variables.

3 Concurrency Constructs

3.1 COBEGIN—COEND

A process in BenAri Concurrent Pascal is a `PROCEDURE`. In the BACI system, the term “concurrent process” is synonymous with the term “concurrent thread.” A list of processes to be run concurrently is enclosed in a `COBEGIN—COEND` block. Such blocks cannot be nested and must appear in the main program.

```

COBEGIN
  proc1(...); proc2(...); ... ; procN(...);
COEND;

```

The `PCODE` statements belonging to the listed procedures are interleaved by the interpreter in an arbitrary, ‘random’ order, so that multiple executions of the same program containing a `COBEGIN—COEND` block can appear to be non-deterministic. The main program is suspended until all of the processes in the `COBEGIN—COEND` block terminate, at which time execution of the main program resumes at the statement following the `COEND` statement.

3.2 Semaphores

The interpreter has a predeclared `SEMAPHORE` type. That is, a `SEMAPHORE` in BACI Concurrent Pascal is a non-negative-valued `INTEGER` variable (see definition below) that can be accessed only in restricted ways. The binary semaphore, one that only assumes the values 0 and 1, is supported by the `BINARYSEM` subtype of the `SEMAPHORE` type. During compilation and execution, the compiler and interpreter enforce the restrictions that a `BINARYSEM` variable can only have the values 0 or 1 and that a `SEMAPHORE` variable can only be non-negative.

3.2.1 Initializing a Semaphore

Assignment to a `SEMAPHORE` or `BINARYSEM` variable is allowed only when the variable is declared. For example, both the following declarations are valid:

```

VAR
  s : SEMAPHORE := 17;
  b : BINARYSEM := 0;

```

The built-in procedure

```
INITIALSEM( semaphore, integer_expression );
```

is the only method available for initializing a semaphore of either type at runtime. In the call, the parameter `integer_expression` can be any expression that evaluates to an integer value is valid for the semaphore (non-negative for a `SEMAPHORE` type, or 0 or 1 for a `BINARYSEM` type). For example, the following two `INITIALSEM` calls show an alternative way to initialize the two semaphores declared above:

```
INITIALSEM( s, 17 );
INITIALSEM( b, 0 );
```

3.2.2 P (or WAIT) and V (or SIGNAL) Procedures

The P procedure (or synonymously, `WAIT`) and the V procedure (or synonymously, `SIGNAL`) are used by concurrently executing processes to synchronize their actions. These procedure provide the user with the only way to change a semaphore's value.

The prototypes of the two procedures is as follows:

```
PROCEDURE P(VAR s : SEMAPHORE);
```

or equivalently,

```
PROCEDURE WAIT(VAR s : SEMAPHORE);
```

and

```
PROCEDURE V(VAR s : SEMAPHORE);
```

or equivalently,

```
PROCEDURE SIGNAL(VAR s : SEMAPHORE);
```

The `SEMAPHORE` argument of each procedure is shown as a `VAR` parameter, because the procedure modifies the value of the `SEMAPHORE`.

The semantics of the P and V procedure calls are as follows:

```
P(sem);
```

If $sem > 0$, then decrement `sem` by 1 and return, allowing P's caller to continue.

If $sem = 0$, then put P's caller to sleep. These actions are **atomic**, in that they are non-interruptible and execute from start to finish.

```
V(sem);
```

If $sem = 0$ and one or more processes are sleeping on `sem`, then awake one of these processes. If no processes are waiting on `sem`, then increment `sem` by one. In any event, V's caller is allowed to continue. These actions are **atomic**, in that they are non-interruptible and execute from start to finish.

Some implementations of V require that processes waiting on a semaphore be awakened in FIFO order (queuing semaphores), but the BACI Interpreter conforms to Dijkstra's original proposal by randomly choosing which process to re-awaken when a signal arrives.

3.2.3 Examples of Semaphore Usage

To help to explain semaphore usage, we offer the following brief example:

```

BACI System: BenAri Pascal PCODE Compiler, 09:22  2 May 2002
Source file: semexample.pm  Thu May  9 09:47:54 2002
line  pc
  1  0  PROGRAM semexample;
  2  0      // example of C-- semaphore usage
  3  0  VAR
  4  0      count      : SEMAPHORE;  // a "general" semaphore
  5  0      output     : BINARYSEM;  // a binary (0 or 1) semaphore for unscrambling output
  6  0
  7  0  PROCEDURE increment;
  8  0  BEGIN
  9  0      P(output);      // obtain exclusive access to standard output
10  2      Writeln('before V(count) value of count is ',count);
11  6      V(output);
12  8      V(count);      // increment the semaphore
13 10  END; // increment
14 11
15 11  PROCEDURE decrement;
16 11  BEGIN
17 11      P(output);      // obtain exclusive access to standard output
18 13      Writeln('before P(count) value of count is ',count);
19 17      V(output);
20 19      P(count);      // decrement the semaphore (or stop -- see manual text)
21 21  END; // decrement
22 22
23 22  BEGIN // semexample
24 23      INITIALSEM(count,0);
25 26      INITIALSEM(output,1);
26 29      COBEGIN
27 30          decrement; increment;
28 34      COEND;
29 35  END. // semexample

```

The program uses two semaphores. One semaphore, `count`, is of `SEMAPHORE` type, which indicates to the BACI system that the semaphore will be allowed to have any non-negative value. The two concurrent procedures, `increment` and `decrement`, “signal” each other through the `count` semaphore. The other semaphore, `output`, is of `BINARYSEM` type, which indicates to the BACI system that the semaphore should always have the value zero or one; any other value causes a run-time exception. This semaphore is used to keep the output from the two concurrently executing procedures, `increment` and `decrement` from intermingling.

We produced the above compiler listing with the command

```

prompt% bapas semexample
Pcode and tables are stored in semexample.pco
Compilation listing is stored in semexample.lst

```

The `semexample.pco` file can then be executed with the BACI PCODE interpreter:

```

prompt% bainterpre semexample
Source file: semexample.pm  Thu May  9 09:47:54 2002
Executing PCODE ...
before V(count) value of count is 0
before P(count) value of count is 1

```

This execution of the PCODE file is one of the three possible outputs that the program can produce. The other two possible program outputs are

```

Source file: semexample.pm Thu May 9 09:47:54 2002
Executing PCODE ...
before P(count) value of count is 0
before V(count) value of count is 0

prompt% bainterp semexample
Source file: semexample.pm Thu May 9 09:47:54 2002
Executing PCODE ...
before V(count) value of count is 0
before P(count) value of count is 0

```

An interested reader might find it instructive to supply explanations for ways in which these three program outputs are generated and to show that these three outputs are the only outputs possible.

3.3 Monitors

The monitor concept, as proposed by Hoare, is supported with some restrictions. A MONITOR is a Pascal block, like a block defined by a procedure or function, with some additional properties. All procedures and functions in the monitor block are visible (that is, callable entry procedures) from the outside of the block, but the monitor variables are not accessible outside of the block and can only be accessed by the procedures and functions of the monitor. In BACI Concurrent Pascal, a monitor can be declared only at the outermost, global level. Monitors can not be nested. The code between the BEGIN—END at the end of the monitor block is run when the main program is started. Use this block to initialize the values of the monitor variables.

Only one procedure or function of the monitor block can execute at any given time. This feature makes it possible to use monitors to implement mutual exclusion. Use of monitors to control concurrency is advantageous because all of the code controlling concurrency is located in the monitor and not distributed widely across callers, as is the case when semaphores are used.

Three constructs are used by the procedures and functions of a monitor to control concurrency: CONDITION variables, WAITC (wait on a condition), and SIGNALC (signal a condition).

3.3.1 Condition Variables

A CONDITION variable can only be defined in a monitor, and thus, can only be accessed by the processes of the monitor. A CONDITION variable never actually *has* a value; it is somewhere to wait or something to signal. A monitor process can wait for a CONDITION to hold or signal that a given CONDITION now holds through the WAITC and SIGNALC calls.

3.3.2 WAITC and SIGNALC Procedures

WAITC and SIGNALC calls have the following syntax and semantics:

```
PROCEDURE WAITC(cond : CONDITION; prio : INTEGER);
```

The monitor process (and hence, also the outside process calling the monitor process) is blocked and assigned the priority `prio` for being re-awakened (see SIGNALC below). Note that this blocking action allows some other monitor process to execute, if one is ready.

```
PROCEDURE WAITC(cond : CONDITION);
```

This call has the same semantics as the WAITC above, but the wait is assigned a default priority of 10.

```
PROCEDURE SIGNALC(cond : CONDITION);
```

Wake some process waiting on condition `cond` with the smallest (highest) priority; otherwise, do nothing. Note that this is quite unlike the semaphore (V or SIGNAL), because SIGNALC is a no-op if no one is waiting, whereas V(`sem`) increments `sem` if no one is waiting, thus “remembering” the action when future P(`sem`)’s occur.

The priority scheme can be used to implement a FIFO discipline in re-awakening waiters. If each MONITOR process increments a monitor variable associated with the current priority assigned to a condition, then successive SIGNALC's to the condition will awaken the sleeping processes in FIFO order.

The compiler provides a BOOLEAN-valued function EMPTY(cond) that returns TRUE if there are no processes waiting in the queue of the condition cond and FALSE otherwise.

3.3.3 Immediate Resumption Requirement

This is the requirement that a process waiting on a condition that has just been signaled should have priority in re-entering the monitor over new calls to monitor processes (those wanting to enter "from the top"). The requirement rests on the assumption that the condition that has just been signaled has more "urgent" business to perform than a new entry into the monitor. The Immediate Resumption Requirement is implemented in the BACI Interpreter by suspending the signaler of a condition and picking (at random) one of the waiters on the condition with the appropriate priority to run. Because of this, monitor procedures that SIGNALC a condition typically do so as their last instruction.

When the process re-awakened by the SIGNALC leaves the monitor, a process executing in the monitor that has been suspended after issuing a SIGNALC call is allowed to resume execution in the monitor in preference to processes attempting to enter the monitor "from the top."

3.3.4 An Example of a Monitor

The following example of an implementation of a general semaphore with a monitor illustrates the monitor syntax:

```

MONITOR monSemaphore;
VAR
    semvalue : INTEGER;
    notbusy  : CONDITION;

PROCEDURE monP;
BEGIN
    IF (semvalue = 0) THEN
        WAITC(notbusy)
    ELSE
        semvalue := semvalue - 1;
END;

PROCEDURE monV;
BEGIN
    IF (EMPTY(notbusy)) THEN
        semvalue := semvalue + 1
    ELSE
        SIGNALC(notbusy);
END;

BEGIN { initialization code }
    semvalue := 1;
END; // of monSemaphore monitor

```

3.4 Other Concurrency Constructs

The BACI Concurrent Pascal compiler provides several low-level concurrency constructs that can be used to create new concurrency control primitives. For example, these functions can be used to create a "fair" (FIFO) queued semaphore. The code to accomplish this is beyond the scope of this user's guide.

3.4.1 ATOMIC Keyword

If a function or procedure is defined as ATOMIC, then the function or procedure is *non-preemptible*. The interpreter will not interrupt an ATOMIC subroutine with a context switch. This provides the user with a method for defining new primitives. The following program illustrates how a test_and_set primitive can be defined and used to enforce mutual exclusion:

```
PROGRAM tas;

ATOMIC FUNCTION test_and_set(VAR target : INTEGER ): INTEGER;
VAR
  u : INTEGER;
BEGIN
  u := target;
  target := 1;
  test_and_set := u;
END;

VAR
  lock : INTEGER := 0;

PROCEDURE proc(id : INTEGER);
VAR
  i : INTEGER := 0;
BEGIN
  WHILE (i < 10) DO
  BEGIN
    WHILE (test_and_set(lock)) DO { nothing } ;
    WRITE(id);
    lock := 0;
    i := i + 1;
  END;
END;

BEGIN // tas
  COBEGIN
    proc(1); proc(2); proc(3);
  COEND;
END.
```

3.4.2 PROCEDURE suspend;

The suspend procedure puts the calling thread to sleep.

3.4.3 PROCEDURE revive(process_number : INTEGER);

The revive procedure revives the process with the given number.

3.4.4 FUNCTION which_proc : INTEGER;

The which_proc function returns the process number of the current thread.

3.4.5 FUNCTION random(range : INTEGER): INTEGER;

The random function returns a “randomly chosen” integer between 0 and range - 1, inclusive. It uses a different random number generator stream than the one used by the interpreter; that is, random() calls do not affect interpreter execution.

4 Built-in String Handling Functions

4.1 PROCEDURE `stringCopy(dest : STRING; src : STRING);`

The `stringCopy` procedure copies the `src` string into the `dest` string. No check for string overrun is performed. For example,

```
VAR
    x : STRING[20];
BEGIN
    stringCopy(x,"Hello, world!");
    stringCopy(x,"");
END.
```

will initialize the string `x` to a well-known value. The second `stringCopy` resets the string `x` to a zero-length string. Either double-quotes (") or a single quote (') can be used as delimiters of a raw string, but the two types of delimiters cannot be mixed.

4.2 PROCEDURE `stringConcat(dest : STRING; src : STRING);`

The `stringConcat` procedure concatenates the `src` string to the end of the `dest`. No check for string overrun is performed.

4.3 FUNCTION `stringCompare(x : STRING; y : STRING): INTEGER;`

The `stringCompare` function has the same semantics as the `strcmp` function from the C string library: a positive number is returned if string `x` is lexicographically after the string `y`, zero is returned if the strings are equal, and a negative number is returned if string `x` is lexicographically before the string `y`.

4.4 FUNCTION `stringLength(x : STRING): INTEGER;`

The `stringLength` function returns the length of the string `x`, not including the termination byte.

4.5 FUNCTION `sscanf(x : STRING; fmt : RAWSTRING ; ...): INTEGER;`

Like the “real” `sscanf` in the C library, the `sscanf` function scans the string `x` according to the format string `fmt`, storing the values scanned into the variables supplied in the parameter list, and returns the number of items scanned. Only the `%d`, `%x`, and `%s` format specifiers of the real `sscanf` are supported. An additional format specifier `%q` (quoted string), unique to the BACI system, is supported. For this specifier, all characters delimited by a pair of double quotes (") will be scanned into the corresponding string variable. When the `%q` specifier is encountered in the format string, if the next non-whitespace character of the string being scanned is not a double quote, then the `%q` scan fails, and scanning of the string terminates.

The variables appearing after the format string are `var` parameters (reference variables).

In the following example, the value of `i` returned by the `sscanf` call will be 4, the value stored in the variable `j` will be 202, the value stored in the string stored in `string x` will be `alongstring`, the value stored in the variable `k` will be `0x3c03`, and the string stored in the `string y` will be a `long string`.

```
VAR
    x, y : STRING[50];
    i, j, k : INTEGER;
BEGIN
    stringCopy(x,"202 alongstring 3c03 \"a long string\");
    i := sscanf(x,"%d %s %x %q",j,x,k,y);
END.
```

4.6 PROCEDURE `sprintf(x : STRING; x, fmt : RAWSTRING,...);`

Like the “real” `sprintf` from the C library, the `sprintf` procedure creates a string stored in the variable `x`, using the format string `fmt` and the variables following the format string.

The `%d`, `%o`, `%x`, `%X`, `%c`, and `%s` format specifiers are supported, in the full generality of the real `sprintf`. In addition, the `%q` format specifier will insert a doubly-quoted string into the output string. The `%q` format specifier is equivalent to the `\ "%s\"` specifier.

For example, in the following code fragment

```
VAR
  x : STRING[80];
  y,z : STRING[15];
BEGIN
  stringCopy(y,"alongstring");
  stringCopy(z,"a long string");
  sprintf(x,"%12d.  .%-20s.  .%q.  .%08X.",202,y,z,0x3c03);
END.
```

the string `x` becomes

```
.          202. .alongstring          . ."a long string". .00003C03.
```

5 Using the BACI Pascal Compiler and PCODE Interpreter

There are two steps to executing a program with the BACI system.

1. Compile a “.pm” file to obtain a “.pco” file

Usage: `bapas [optional_flags] source_filename`

Optional flags:

```
-h show this help
-c make a .pob object file for subsequent linking
```

The name of the source file is required. If missing, you will be prompted for it. The file suffix “.pm” will be appended to the filename if you don’t supply it.

2. Interpret a “.pco” file to execute the program

usage: `baininterp [optional_flags] pcode_filename`

Optional flags:

```
-d enter the debugger, single step, set breakpoints
-e show the activation record (AR) on entry to each process
-x show the AR on exit from each process
-t announce process termination
-h show this help
-p show PCODE instructions as they are executed
```

The name of the PCODE file is required. If missing, you will be prompted for it. The file suffix “.pco” will be appended to the filename you give.

6 Sample program and output

The following listing was produced by the BACI Pascal compiler. The number to the right of the line number is the PCODE offset of the instruction that begins that line. The BACI compiler creates this listing from the file "inremen.pm". The listing is placed in the file "inremen.lst". An "inremen.pco" file is also created; this file is used by the interpreter below.

```
BACI System: BenAri Pascal PCODE Compiler, 10:32 21 Oct 1997
Source file: inremen.pm Sun Sep 12 08:45:34 1993
line pc
 1 0 PROGRAM increment;
 2 0 CONST
 3 0 m = 5;
 4 0 VAR
 5 0 n : INTEGER;
 6 0
 7 0 PROCEDURE incr( id : char );
 8 0 VAR
 9 0 i : INTEGER;
10 0 BEGIN (* incr *)
11 0 FOR i := 1 TO m DO
12 4 BEGIN
13 4 n := n + 1;
14 9 WRITELN( id , ' n =', n , ' i =', i , ' ', id );
15 22 END;
16 23 END; (* of incr *)
17 24
18 24 BEGIN (* main program *)
19 25 n := 0;
20 28 COBEGIN
21 29 incr( 'A' ); incr( 'B' ); incr( 'C' );
22 38 COEND;
23 39 WRITELN( 'The sum is ' , n );
24 43 END.
```

The following listing was produced by the BACI interpreter. The interpreter executes the program that was compiled into the file "inremen.pco".

```
Source file: inremen.pm Sun Sep 12 08:45:34 1993
Executing PCODE ...
A n =1 i =1 A
A n =2 i =2 C n =BA n =2 i =
13 i =1 C
A n =3 i =3 C n = B
4 i =2A
C
B n =5 i =2 CB
n =6 i =3 CB n =
A n =8 i =C n =8 i =44 7 i =C
C n =9A
i =5A n =10 i =5 A
3 B
CB
n =11 i =4 B
B n =12 i =5 B
The sum is 12
```