Parallel Programming with PCN

Ian Foster
Steve Tuecke

CRPC-TR91174
December 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439-4801

ANL-91/32, Rev. 1

Parallel Programming with PCN

by

Ian Foster and Steve Tuecke

Mathematics and Computer Science Division

December 1991

This work was supported in part by the National Science Foundation under Contract NSF CCR-8809615, by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the Air Force Office for Scientific Research under Contract AFOSR-91-0070, by the Office for Naval Research under Contract ONR-N00014-89-J-3201, and by the Defense Advanced Research Projects Agency under Contract DARPA-N00014-87-K-0745.

Preface

The PCN system is the product of the efforts of many people at Argonne National Laboratory, the California Institute of Technology, and the Aerospace Corporation. The PCN language was designed by Mani Chandy and Steve Taylor. The PCN toolkit was designed by Ian Foster and Steve Taylor and implemented by a team consisting of Sharon Brunett, Ian Foster, Steve Hammond, Carl Kesselman, Tal Lancaster, Dong Lin, Jan Lindhiem, Robert Olson, Steve Taylor, and Steve Tuecke. The Gauge performance analysis tool was provided by Carl Kesselman. The Upshot trace analysis tool was provided by Ewing Lusk. The expanded BNF syntax for PCN was provided by John Thornley. The two-point boundary value application was provided by Steve Wright.

Contents

Parallel Programming with PCN

Ian Foster and Steve Tuecke

Abstract

PCN is a system for developing and executing parallel programs. It comprises a high-level programming language, tools for developing and debugging programs in this language, and interfaces to Fortran and C that allow the reuse of existing code in multilingual parallel programs. Programs developed using PCN are portable across many different workstations, networks, and parallel computers.

This document provides all the information required to develop parallel programs with the PCN programming system. It includes both tutorial and reference material. It also presents the basic concepts that underly PCN, particularly where these are likely to be unfamiliar to the reader, and provides pointers to other documentation on the PCN language, programming techniques, and tools.

PCN is in the public domain. The latest version of both the software and this manual can be obtained by anonymous FTP from Argonne National Laboratory in the directory `pub/pcn` at `info.mcs.anl.gov` (c.f. Appendix A).

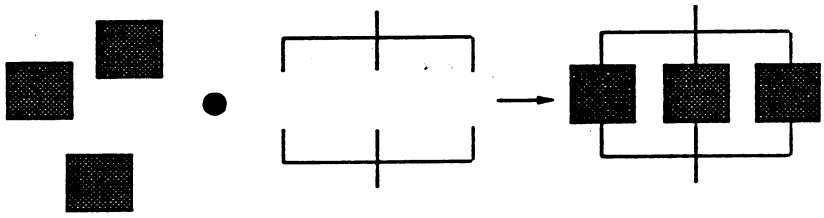
Part I

A Tutorial Introduction

1 Program Composition

Program Composition Notation (PCN) is both a programming language and a parallel programming system. As the name suggests, both the language and the programming system center on the notion of *program composition.*

Most programming languages emphasize techniques used to develop individual components (blocks, procedures, modules). In PCN, the focus of attention is the techniques used to put components together (i.e., to compose them). This is illustrated in the following figure, which shows a combining form being used to compose three programs.



This focus on combining forms is important for several reasons. First, it encourages reuse of parallel code: a single combining form can be used to develop many different parallel programs. Second, it facilitates reuse of sequential code: parallel programs can be developed by composing existing modules written in languages such as Fortran and C. Third, it simplifies development, debugging, and optimization, by exposing the basic structure of parallel programs.

It appears likely that a large proportion of all parallel programs can be developed with a relatively small number of combining forms. However, PCN does not attempt to enumerate potential combining forms. Instead, it provides a core set of three primitive composition operators — parallel, sequential, and choice composition — in a *core programming notation.* This is a simple, high-level programming language. More sophisticated combining forms (providing, for example, divide-and-conquer, self-scheduling, or domain decomposition strategies) can be implemented as user-defined extensions to this core notation. Such extensions are referred to as *templates* or *user-defined composition operators.* Program development, both with the core notation and with templates, is supported by a *portable toolkit.* These three components of the PCN system are illustrated in Figure 1.

This tutorial focuses on the core programming notation and toolkit. Material on templates will be added at a later date.

1.1 Core Programming Notation

The core PCN programming notation is a simple, high-level language that provides three basic composition operators: *parallel, sequential,* and *choice.* The lan-

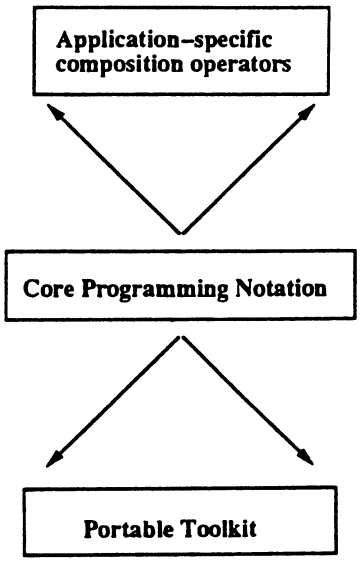


Figure 1: PCN System Structure

guage provides two types of variable: conventional, or *mutable* variables, and single-assignment, or *definitional* variables. Other distinctive features of the language include extensive use of recursion, support for both numeric and symbolic computing, and an interface to sequential languages such as Fortran and C. The syntax is similar to that of C.

1.2 Toolkit

The PCN toolkit provides support for each stage of the parallel program development process. It comprises a compiler, shell, foreign language interface and linker, standard libraries, process mapping tools, programmable transformation system, symbolic debugger, execution profiler, and trace analyzer. These facilities are all machine independent and can run on a wide variety of uniprocessors, multiprocessors, and multicomputers. They are supported by a *run-time system* that provides basic machine-dependent facilities.

Compiler The compiler translates PCN programs to a machine-independent, low-level form. An interface to the C preprocessor allows macros, conditional compilation constructs, and the like, to be used in PCN programs.

Shell The shell supports interactive program development, providing access to basic services such as I/O and compilation.

Foreign language interface and linker These permit Fortran and C procedures to be integrated seamlessly into PCN programs.

Process mapping tools These support process mapping on a variety of virtual machines.

Standard libraries A set of standard libraries provides access to Unix facilities (e.g., I/O) and other capabilities.

PDB PDB is the PCN symbolic debugger. It includes specialized support for debugging of concurrent programs.

Gauge Gauge is an execution profiler for programs written in PCN and other languages; it includes a graphical tool for interactive exploration of profile data.

Upshot Upshot is a trace analysis tool for programs written in PCN and other languages; it includes a graphical tool for interactive exploration of trace data.

PTN Program Transformation Notation (PTN) is a programmable transformation system integrated with the PCN compiler. It is used to implement the PCN compiler, process mapping strategies, and templates.

1.3 Cross Reference

The basic constructs of the PCN language are described in the following sections.

- Syntax: § 5.2 and Appendix G.
- Sequential composition: § 5.3.
- Mutable Variables: § 5.3.
- Parallel composition: § 5.4.
- Definitional Variables: § 5.4.
- Choice composition: § 5.5.

The components of the PCN toolkit are described in the following sections.

- Compiler: § 3.1, § 8.
- Shell: § 4.
- Foreign interface and linker: § 9.
- Debugging facilities: § 13.
- Standard libraries: § 17.
- Process mapping tools: § 11.

- PDB: § 14.
- Gauge: § 15.
- Upshot: § 16.

Machine-specific aspects of the PCN toolkit are described in §§ 19–23. Additional documentation on the PCN language, toolkit, and applications is cited in § 24.

The PTN transformation system is described in a separate document, as is `host-control`, a utility for managing execution of PCN programs on networks. See § 24 for more information.

2 Getting Started

We assume that PCN is already installed on your computer. (If it isn't, read the documentation provided with the PCN software release.) You will need to know *where* PCN is installed. Normally, this will be `/usr/local/pcn`, but some systems may place PCN in a different location.

Before you can use PCN, you must tell your Unix environment where to find the PCN software. If you are using the standard Unix C-shell (csh), you add one line to the *end* of the file `.cshrc` in your home directory. If PCN has been installed in `/usr/local/pcn`, this line is

```
set path = ($path /usr/local/pcn/bin)
```

The environment variable `path` tells the Unix shell where to find the various PCN programs (compiler, linker, etc.). This shell command adds the directory containing the various PCN executables to your shell's search path. You may have to log out and log in again for this to take effect.

3 An Example Program

We are now ready to compile and run our first PCN program. The syntax of PCN is similar to that of the C programming language in many respects. Hence, it is appropriate that our first program print "Hello world" (the first C program in several well-known texts does just this).

Module `program1.pcn`

```
hello(d)
{|| stdio:printf("Hello world\n", {}, d) }
```

A PCN program consists of one or more modules. Each module is contained in a separate file with a `.pcn` suffix. Our example program consists of a single module, `program1`, contained in a file `program1.pcn`. (We'll learn more about modules later.)

The example program has one procedure, `hello`. This procedure makes what is called an *intermodule call*: it calls the `printf` procedure in the `stdio` module to print "Hello world". The `stdio` module is distributed with the PCN system; it provides many of the functions of the Unix "standard I/O" library (§ 17.2).

3.1 Compiling a Program

The command `pncomp` is used to compile a PCN module. Because our program is contained in a file `program1.pcn`, we type

```
pncomp program1.pcn
```

We should see something like this (if % is the Unix prompt):

```
% pncomp program1.pcn
Compiling:  program1.pcn
PCN: Version 1.2; 1 node, 800k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
loaded co
%
```

This sample session, and all sample sessions in this manual, were run on a NeXT workstation. The invocation syntax and system messages may differ slightly on some other computers (cf. §§ 19–23). However, the same PCN programs can be compiled and run on any computer for which PCN is supported. We adopt the convention that text typed by the user is presented in *italic font*, while system output is presented in `typewriter font`.

The compiler will produce two files when compiling `program1.pcn`, namely, `program1.pam` and `program1.mod`. The `.pam` file contains PCN object code. The `.mod` file contains information about the program, to be used by the Gauge execution profiler (§ 15). Both the `.pam` and `.mod` files are completely machine independent: a PCN program compiled for one machine will work on any other machine without recompilation. However, if a PCN program invokes foreign procedures (Fortran or C), then these foreign procedures must be compiled and linked using the PCN linker, to generate a separate executable for each target machine (§ 9).

3.2 Running a Program

We are now ready to run our program. We first invoke the PCN shell, `pcn`. (We exit this shell by typing `exit(0);` on some computers, a `^D` (control-D) will also work.)

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
exit(0)
%
```

In this script, we started the shell and immediately exited it using `exit(0)`. Now, we start up the shell and run our program:

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
program1:hello(done)
Hello world
exit(0)
%
```

Once the shell is running, we type the command `program1:hello(done)` to invoke the `hello` procedure in our `program1` module. The shell looks in the current directory for the object file `program1.pam` produced by the compiler; if it does not find the file there, it looks in the PCN installation directory. (We can also specify other directories that should be searched; see § 25). If the file is found, the shell loads the object file and executes the procedure `hello`. Once our program has completed, we exit by typing `exit(0)`.

4 The Shell

The preceding section gave you a first taste of the PCN language and the PCN shell. We now look at the shell in more detail. We return to the language in § 5.

Recall that the shell, when first invoked, displays something like the following text:

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
```

The header provides some useful information about the version of PCN that is running (here, 1.2), the number of processors that are active (here, just 1), and the amount of memory available to PCN (here, 512 kilowords). The second line invites you to read a disclaimer file associated with the software. The asterisk on the third line means that the shell is running. On a large parallel machine, it may take a little time before this appears. We can now type one or more commands, separated by commas. A prompt (>) can be obtained on demand by typing a carriage return.

The shell provides three main functions. It maintains special *shell variables*. It accepts requests to execute *capabilities* and *intermodule calls*. Finally, in a parallel computer, it handles the *mapping* of procedure calls to remote processors. We discuss the first two of these functions here; mapping is discussed in § 10.1.

4.1 Intermodule Calls

We have already seen an example of an intermodule call in the preceding section. The call

```
program1:hello(done)
```

requests the shell to load the module `program1` (if it has not already been loaded) and to execute the procedure `hello` with a single argument, `done`. If either `program1` cannot be loaded (i.e., there is no readable file `program1.pam` in the current directory, the PCN installation directory, or the PCN directory path), or if `program1` does not contain a procedure `hello`, an error is reported.

4.2 Shell Variables

The shell maintains a dictionary of variables passed as arguments to intermodule calls or capabilities. Hence, the values of these variables persist after an intermodule call or capability completes, allowing communication between various procedure invocations initiated from within the shell.

Shell variables are called single-assignment, or *definitional*, variables. Definitional variables are a central concept in PCN: they provide an abstract representation of the communication and synchronization operations that are fundamental to parallel computing. A definitional variable initially has a special undefined value. Once defined (written) to a nonvariable term, it cannot subsequently be modified. An attempt to *read* an undefined definitional variable causes the reading procedure to *suspend*. An attempt to write an already-defined definitional variable results in an error.

Variables are represented in PCN by character strings formed from the set {`a-z,A-Z,0-9,_`} and starting with a letter or an underscore. Case is significant, and there is no maximum length. Definitional variables are not declared but instead are simply created when used. We have already seen one example of a definitional variable. In the call to procedure `hello` in `program1`,

```
program1:hello(done),
```

the `hello` procedure's `done` argument is a definitional variable.

4.3 Capabilities

The shell also provides a number of *capabilities*, procedures that can be invoked without specifying a module. The following basic capabilities are built into the shell:

`pp(X)` "Pretty print" the value of `X`. (If `X` is a list, the elements of `X` are printed.)

`forget()` Forget the bindings of all shell variables.

`exit(X)` Shut down the shell and exit, once the variable `X` is defined.

`load(M)`, `load(M,L,R)` Load a program module with name given by the string `M`, adding into the environment any capabilities it defines; define the variable `R` to be `L` when loading is complete.

The `load` capability allows us to extend the range of capabilities available for use in the shell. For example, the PCN system module `co` provides a capability `compile` that can be used to compile programs from within the shell. This is an alternative to the use of the `pcncomp` command. To use this capability, we first load the compiler module by typing `load("co");` this has the effect of adding the capability `compile` to the shell. We then type `compile("program1")` to compile a file `program1.pcn`:

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
load("co")
loaded co
compile("program1")
Written:  program1.pam
%
```

Several other system modules providing useful capabilities will be introduced in subsequent sections; a complete list is given in § 18.

4.4 The .pcnrc File

It can become tiresome to have to type `load("co")` each time the PCN shell is invoked. Fortunately, you can request that certain shell commands be performed automatically each time the shell is started. You simply create a `.pcnrc` file in either the directory in which PCN is to be invoked or your home directory (~). PCN looks for this file each time it starts up (it looks in the current directory first, and then in the home directory) and executes any commands contained in the file. The following is a typical `.pcnrc` file; this loads the compiler and the Gauge execution profiler.

File .pcnrc

```
load("co")
load("gauge")
```

4.5 Concurrency and Sequencing

Unlike the Unix shell, the PCN shell does not wait until one command is finished before invoking the next. Hence, it is possible to run several commands concurrently by typing them one after the other. For example, in the following script, we execute our program twice concurrently.

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
program1:hello(d1), program1:hello(d2)
Hello world
Hello world
```

It is important to be aware that `exit` will also be executed concurrently: this can lead to premature termination of other computations. For example, if we type

```
program1:hello(d1), exit(0)
```

it is possible for the shell to terminate before `program1` has completed printing `Hello world`.

We can avoid this problem by using a shell variable to sequence printing and exiting. The following example shows how a shell variable, `done`, is used to sequence the calls to `program1` and `exit`.

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
program1:hello(done), exit(done)
Hello world
%
```

Recall the example program:

Module program1.pcn

```
hello(d)
{|| stdio:printf("Hello world\n", {}, d) }
```

The hello procedure takes one argument, named d. It passes this argument to the printf procedure as its third argument. The printf procedure defines its third argument when it is finished printing. In PCN, arguments are always passed by reference, so the d variable in the hello procedure and the done variable in the call are one and the same. Hence, the program1:hello(done) command defines the shell variable done once printing is complete. Because the exit(done) call will not execute until its argument has been defined, the exit command shuts down the shell only after Hello world has appeared on the screen.

The definition operation effected by hello can be observed more closely by typing commands one at a time, as in the following script. The first call to pp shows that the variable done is initially undefined (done = _U2f450 where the syntax _Un represents a variable). The second call to pp shows that the variable done has been given the value 16 by hello. The value 16 is a return code from the printf command and is not relevant to the current discussion.

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
pp(done)
_U2f450
program1:hello(done)
Hello world
pp(done)
16
exit(done)
%
```

4.6 Common Errors

Illegal Define: Recall that a definitional variable (shell variable) can be defined only once. An attempt to redefine a shell variable results in an *Illegal define* (i_define) error. For example, in the following we make the mistake of passing the same definitional variable to two invocations of hello. An illegal define error is signaled.

```
% pcn
PCN: Version 1.2; 1 node, 512k heap.
(See the file:  /usr/local/pcn/DISCLAIMER)
*
program1:hello(done), program1:hello(done)
Hello world
Hello world
(0,6879) Warning:  Node 0:  i_define:  left-hand side
of definition already defined in fprintf1.40
```

We can avoid this error by introducing two different variables:

```
program1:hello(done1), program1:hello(done2)
```

Alternatively, we can `forget()` the value of all definitions between the two calls to our program.

```
program1:hello(done)
Hello world
forget()
definitions forgotten
program1:hello(done)
Hello world
```

Insufficient Memory. The PCN compiler invokes the C preprocessor (CPP) to process macros, etc. This program is executed as a separate process. Hence, if your computer has little swap space, it is possible for CPP to fail due to insufficient memory. If this occurs, you will get an error message indicating that not enough memory was available. To work around this problem, decrease the size of PCN internal memory by invoking `pcn` (or `pcncomp`) with the argument `-k N`, where `N` is less than the default value of 512, and retry the compile.

Getting started with PCN:

- PCN programs are contained in files with a `.pcn` suffix; compilation produces files with `.pam` and `.mod` suffixes.
- We compile programs by typing either `pncomp file.pcn` to the Unix shell or `compile("file")` to the PCN shell.
- We invoke the shell as `pcn` and exit it by typing `exit(0)` or `^D` (control-D).
- The shell provides four built-in capabilities: `pp`, `forget`, `exit`, and `load`.
- When using the compiler from the PCN shell, we load the capability `compile` by typing `load("co")`.
- Commonly used commands can be executed automatically by placing them in a `.pcnrc` file in the directory from which PCN is run, or in your home directory.
- The shell maintains shell variables that can be used to communicate information between different commands and to sequence execution of commands.

5 The PCN Language

The programming language Program Composition Notation (PCN) is an integral part of the PCN programming system: it is used to express concurrent algorithms and to compose code written in sequential languages. Like any programming language, PCN has a distinct syntax that must be mastered in order to write programs. However, the key to understanding PCN is understanding the concurrent programming model that it implements. Before presenting the PCN language, we introduce this model and the fundamental concurrent programming concepts on which it is based.

5.1 Concurrent Programming Concepts

Parallel programming is often considered "hard". However, experience shows that programming models that adhere to the following principles can significantly reduce the complexity of parallel programming.

First-Class Concurrency: Concurrent execution should be a first-class citizen in a programming model, not something appended to a sequential model.

Controlled nondeterminism: The result computed by a procedure should be fully determined by the procedure's inputs, except when explicitly specified

otherwise by the programmer.

Compositionality: It should be easy to understand both isolated program components and larger programs formed by the concurrent composition of these components.

Mapping independence: The way in which components of a concurrent computation are mapped to a parallel computer should not change the result computed.

PCN uses four simple ideas to realize a parallel programming model based on these principles. *Definitional variables* provide an abstract, machine-independent model of both communication and synchronization. *Concurrent composition* is the fundamental mechanism used to build up complex programs from simpler components. *Nondeterministic choice* is used to specify nondeterministic actions when required. *Encapsulation of state change* allows state change to be integrated into concurrent computations without compromising deterministic execution.

Definitional Variables. A single mechanism is provided for the exchange of information between concurrently executing program components (processes): the definitional variable. A definitional variable is initially undefined, can be assigned at most a single value, and subsequently cannot change. A process that requires the value of a definitional variable waits until the variable is defined.

Definitional variables can be used both to communicate values and to synchronize actions. If two concurrent processes, a producer and a consumer, share a definitional variable, then a value provided by the producer for this variable is automatically communicated to the consumer. Execution of the consumer is blocked until the value is provided.

The definitional variable has several benefits for concurrent programming. First, it avoids the nondeterminism that is so often associated with concurrency: choices made within program components on the basis of definitional variables cannot change. This means that components can be understood in isolation, as errors caused by time-dependent interactions cannot arise. Second, shared definitional variables provide a clearly defined and delineated interface between concurrently executing processes: interaction can only occur if processes share variables. Third, the definitional variable provides for mapping independence: processes sharing a definitional variable may interact irrespective of their location in a parallel computer.

Concurrent Composition. Complex programs are developed by the concurrent composition of simpler components. Hence, an application can be viewed as consisting of a (potentially large) number of lightweight execution threads. These execute concurrently, communicate via definitional variables, and block when required data is unavailable.

It is often desirable that the number of threads be larger than the number of processors, as this can allow the compiler and run-time system to adopt flexible scheduling strategies that overlap computation and communication, thus masking latency and improving parallel efficiency.

Nondeterministic Choice. The use of definitional variables as a communication mechanism avoids errors arising from time-dependent interactions: a choice made on the basis of a definitional variable cannot change. Hence, concurrent computations are deterministic. This is an important property that greatly simplifies parallel programming.

Nevertheless, it is sometimes useful to be able to specify nondeterministic execution, particularly in reactive applications. Nondeterminism is integrated into the programming model in a tightly controlled way. A form of guarded command is used to define the conditions under which a process may perform various actions. Only if the conditions associated with two or more actions are not mutually exclusive is execution nondeterministic.

Encapsulation of State Change. The familiar concepts of state change and sequencing that underly sequential languages such as Fortran and C are also important in parallel programming: many algorithms are most efficiently specified in these terms. However, state change must be carefully controlled if we are to avoid introducing unwanted nondeterminism.

The approach adopted in PCN is to insist that state change be encapsulated within sequential threads. Data structures that may be subject to state change cannot be shared by concurrently executing program components. This restriction prevents concurrent updates to state, which in turn avoids the possibility of time-dependent behavior.

Programming Model Summary. Execution of a parallel program forms a set of concurrently executing lightweight processes (threads) which communicate and synchronize by reading and writing shared definitional variables. Individual threads may apply the usual sequential programming techniques of state change and sequencing. Execution is deterministic, unless specialized operators are invoked to make nondeterministic choices.

Key concurrent programming concepts:

- Definitional variables
- Concurrent composition
- Controlled nondeterministic choice
- Encapsulation of state change

5.2 PCN Syntax

The syntax of PCN is modeled on that of the C programming language. In addition, the C preprocessor is applied to programs, so macros, conditional compilation, and file inclusion constructs can be used as in C (§ 8). In the following, we make frequent

reference to C when explaining features of PCN. However, these references are for illustrative purposes only, and a familiarity with C is not required to understand this material. A complete BNF grammar for the PCN syntax is provided in Appendix G.

Data Types. PCN's three simple data types — character, integer, and double-precision floating-point number (`char`, `int`, and `double`) — are as in C. One-dimensional arrays of these data types are also supported. Arrays are indexed from zero, as in C. There is also a complex data type, the tuple. This is introduced in § 5.8.

Strings. Strings are represented as character arrays, as in C. A character array A representing a string S of length k contains the ASCII representation of the characters of S in $A[0]..A[k-1]$ and the null character (`\0`) in $A[k]$. A constant string is denoted by the characters of the string between quotes; for example, `"PCN"` is a string consisting of the three characters: `P`, `C`, and `N` (followed by the null character). The empty string is denoted by `""`.

Expressions. Arithmetic expressions are as in C, except that the only operators are modulus, addition, subtraction, multiplication, and division (%, +, −, *, and /). The `length` function returns the number of elements in an array or 1 (one) if applied to a single number or character. The following are all valid expressions.

```
(1 + x)%y      i * length(g)      29 - x/g
```

The precedence and associativity of the PCN operators is as in C. The following table summarizes precedence and associativity rules. Operators on the same line have the same precedence, while rows are in order of decreasing precedence. Parentheses () can be used to override these default rules.

Operators	Associativity
− *(negation of numbers)* `length`	right to left
/ %	left to right
+ −	left to right

Variable Names. Variable names are as in C. A variable name is a character string formed from the set {`a-z,A-Z,0-9,_`} and starting with a letter or an underscore ("_"). Case is significant and there is no maximum length. The following are all valid variable names.

```
value    _2    Last_Item    x
```

Comments. A comment begins with /* and ends with */, as in C.

Procedures. A procedure consists of a heading followed by a declaration section followed by a block. The *heading* is the procedure name and a list of arguments (i.e., formal parameters), as in C. All arguments are passed by reference, unlike in C where arguments can be passed by value. The *declaration section* is a set of declarations for arguments and local variables. The scope of a variable is the procedure in which it appears: all variables appearing in a procedure are either arguments or local variables of the procedure. In particular, there is no notion of a global variable.

The body of a procedure consists of a *block*. The block is the basic component from which procedures are constructed. A block is either a composition, an assignment statement, a definition statement, an implication, or a procedure call. These constructs will be defined shortly.

Declarations. A declaration consists of a type (`char`, `int`, or `double`) followed by one or more variable names, each with an optional suffix to denote an array. An array suffix has the form `[`*size*`]`, *size* an integer or variable, if a variable is local, and `[]` if the variable is an argument. The following are all valid declarations.

```
int a[size];     double b[10], c[], d;     char c;
```

We shall see that declarations are not provided for all variables: the definitional variables used in PCN for communication and synchronization are distinguished by a lack of declaration.

5.3 Sequential Composition and Mutable Variables

We now explore the PCN language proper. We shall view PCN as providing three related sets of constructs. First, there are the *composition operators* — parallel, sequential, and choice — which encode three fundamental ways of putting program components together. Second, there are two types of *variables*: conventional, or mutable variables, and single-assignment, or definitional variables. Third, there are specialized language features introduced to support *symbolic processing*: tuples and recursion.

We first introduce the two components that will be most familiar to many readers: sequential composition and mutable variables.

The *sequential composition* operator is used to specify that a set of statements should be executed sequentially, in the order written in the program. In languages such as Fortran and C, this is of course the normal mode of execution. However, as PCN also allows for other sorts of composition, we distinguish it by a special syntax. A sequential composition has the general form

$$\{\ ;\ block_0,\ \ldots,\ block_k\ \},$$

where ";" is the sequential composition operator and $block_0$, ..., $block_k$ are other blocks.

A *mutable variable* in PCN, like a variable in Fortran or C, is declared to have some type (`char`, `int`, or `double`), initially has some arbitrary (undefined) value, and can be modified many times during its lifetime, by means of an assignment statement. An assignment statement is represented as follows,

```
variable := expression
```

where `variable` is a mutable variable or an element of a mutable array.

Example. The procedure `swap` exchanges the values stored at the `i`th and `j`th positions of an integer `array`. Its three arguments — `array`, `i`, and `j` — are declared to be an integer array and single integers, respectively. A local variable `temp` is also declared. The three assignments are placed in a sequential composition, to ensure that they execute in the correct order.

The procedure `swaptest` can be used to execute `swap`. This procedure declares a local integer array `a[3]`; initializes this array to contain the integers `0, 1, 2`; calls a procedure `printf` to display the contents of `a`; calls `swap` to exchange the `i`th and `j`th components; and finally calls `printf` again to display the modified array. Note that as procedure arguments are passed by reference, the array `a` in `swaptest` is the same data structure as `array` in `swap`. Note also that in `swaptest`, the sequential composition operator ensures that both the assignments to `a` and the calls to `printf` occur in the correct order.

```
swap(array,i,j)
int array[], i, j, temp;
{ ; temp     := array[j],
    array[j] := array[i],
    array[i] := temp
}

swaptest(i,j)
int a[3], i, j;
{ ; a[0] := 0, a[1] := 1, a[2] := 2,
    stdio:printf("Before: %d %d %d\n",{a[0],a[1],a[2]},_),
    swap(a,i,j),
    stdio:printf("After: %d %d %d\n",{a[0],a[1],a[2]},_)
}
```

Role of Sequential Composition. The example illustrates the two primary applications of sequential composition in PCN: sequencing of updates to mutable variables and sequencing of I/O operations.

5.4 Parallel Composition and Definitional Variables

We now consider two related constructs that may be unfamiliar to some readers: parallel composition and definitional variables.

The parallel composition operator specifies that a set of statements are to be executed concurrently. A parallel composition has the general form

$$\{||\ \texttt{block}_0, \ \ldots, \ \texttt{block}_k\},$$

where || is the parallel composition operator and $\texttt{block}_0$, ..., $\texttt{block}_k$ are other blocks. Execution within a parallel composition is fair: that is, it is guaranteed that execution of each block will eventually progress (unless that block has terminated). Execution of a parallel composition terminates when all of its constituent blocks have terminated.

Concurrent computations initiated within a parallel composition must be able to exchange data and synchronize their activities. It is important to understand that this *cannot* be achieved by using mutable variables (at least not without the introduction of complex locking mechanisms), as the order of read and write operations in a parallel composition, and hence the result of such operations, is not in general well defined.

Concurrent computations communicate and synchronize by means of *definitional* or single-assignment variables. We have already come across definitional variables in the introduction to this chapter and in our discussion of the PCN shell (§ 4). Here, we consider them in more detail.

Definitional variables are represented in the same way as mutable variables, with one exception: a solitary underscore character ("_") is used to represent an *anonymous* definitional variable. Each occurrence of "_" represents a unique variable.

Definitional variables are not declared. Any variable occurring in a procedure that is not explicitly declared in the procedure's declaration section is a definitional variable. Definitional variables initially have a special "*undefined*" value. They can be defined once, by means of a definition statement, and then cannot be modified. The definition statement is represented as

```
variable = expression,
```

where `variable` is a definitional variable. Note that a definition of the form `x = y` is allowed; this establishes `y` as an alias for `x`, so that any prior or subsequent definition for `y` also applies to `x`.

Example: Simple Divide and Conquer. The following program implements a simple divide-and-conquer strategy. As none of the variables in this procedure are declared, we see that all are definitional. Variables `prob` and `soln` are arguments; the rest are local to the procedure. When executed, procedure `div_and_conq` immediately executes a parallel composition containing four procedure calls. These execute concurrently, with execution order constrained only by availability of data. Variable `prob` is input and `soln` output. Procedure `split` consumes `prob` and hence will block until an input value is available. Likewise, the `solve` procedures block

until l_prob and r_prob are defined by split. Once the two calls to solve produce values for l_soln and r_soln, the combine procedure can proceed to produce soln.

```
div_and_conq(prob,soln)
{|| split(prob,l_prob,r_prob),
    solve(l_prob,l_soln),
    solve(r_prob,r_soln),
    combine(l_soln,r_soln,soln)
}
```

Properties of Definitional Variables

- Have as initial value a special "undefined" value.
- Read operations block until the definitional variable is given a value.
- Are defined ("written") by the definition operator ("=").
- Once defined, cannot be modified.
- Can be shared by procedures in a parallel composition.
- Are not explicitly declared.
- Can take on values of type **char**, **int**, **double**, or **tuple**.

It is instructive to compare mutable and definitional variables, as in the following table.

	Definitional	Mutable
Initial value	Special "undefined" value	Arbitrary value
Defined by	Definition operator (=)	Assignment operator (:=)
Read operation	Blocks if undefined	Always succeeds
Can be written	Once	Many times
Parallel composition	Can share	Cannot share
Explicitly declared	No	Yes
Types	tuple, int, double, char	int, double, char

Role of Parallel Composition. It is important to understand the distinct roles of the parallel and sequential composition operators. Parallel composition exposes opportunities for concurrent execution; sequential composition constrains execution order so as to sequence I/O operations or assignments to mutable variables. In general, it is a good idea to expose as much concurrency as possible in an application, as this provides the compiler and run-time system with maximum flexibility when

making scheduling decisions. In particular, they can seek to reduce the cost of remote data accesses by overlapping computation and communication.

5.5 Choice Composition

The third and final composition operator that we consider is the choice composition operator, "?". A choice composition has the general form

$$\{ \texttt{?} \quad \texttt{guard}_0 \texttt{ -> block}_0\texttt{, ..., guard}_k \texttt{ -> block}_k\}$$

where each $\texttt{guard}_i$ is a sequence of one or more tests. Valid tests include

$a < b$, $a > b$, $a <= b$, $a >= b$: arithmetic comparison

$a == b$, a != b : equality and inequality tests

`int(`a`)`, `char(`a`)`, `double(`a`)`, `tuple(`a`)` : type tests

`data(`a`)` : synchronization test

? = : tuple match

`default` : default action

We refer to a single "`guard -> block`" as an *implication.*

Choosing between Alternatives. Choice composition provides a mechanism for choosing between alternatives. In this respect it may be regarded as a parallel *if-then-else* or guarded command. Each guard specifies the conditions that must be satisfied for the associated block to be executed. *At most one* of these blocks will be executed; which one depends on the result of guard evaluation.

A choice composition is executed as follows. Each guard is evaluated from left to right. A guard succeeds if all of its tests succeed. If one or more guards succeed, exactly one of the corresponding blocks is chosen to be executed.

For example, the procedure **max** executes either `z = x` or `z = y`, depending on the value of `x` and `y`, and hence defines `z` to be the larger of `x` and `y`.

Module `max.pcn`: Version 1

```
max(x,y,z)
{ ? x >= y -> z = x,
     x <  y -> z = y
}
```

Synchronization. Choice composition also provides a synchronization mechanism. A test *suspends* when evaluated if it requires the value of an undefined definitional variable. (E.g., `x < 3`, where `x` is undefined.) Otherwise, it succeeds or fails depending on the value of its arguments.

A guard is evaluated from left to right. If any test suspends, the guard suspends. If any test fails, the guard fails. If all tests succeed, the guard succeeds.

If some guards suspend and all other guards fail, execution of the choice composition is suspended until more data is available. If all guards fail, execution of the choice composition terminates without doing anything. Hence, a call to the procedure `max` given above will suspend until both `x` and `y` have values, and then proceed as follows. If both `x` and `y` are numbers, the procedure executes either the first or second implication, depending on the values of `x` and `y`. If either `x` or `y` is not a number, the procedure terminates without doing anything.

The guard test `default` succeeds only if all other guards in a choice composition fail. For example, consider the following alternative formulation of the `max` procedure.

Module `max.pcn`: Version 2

```
max(x,y,z)
{ ? x >= y  -> z = x,
    default -> z = y
}
```

The two versions of `max` give the same behavior if `x` and `y` are numbers but behave differently if either `x` or `y` is not a number: in that case, the first program terminates without executing either implication, while the second program selects the second implication.

Choice composition rules:

- Evaluate each guard left to right.
- If any test suspends/fails, guard suspends/fails.
- If all tests succeed, guard succeeds.
- If all guards fail, process terminates.
- If no guards succeed and some suspend, process suspends.
- If some guards succeed, execute one implication body.
- Guard test `default` succeeds if all others fail.

Nondeterministic Choice. Choice composition also provides a mechanism by which nondeterminism is introduced into PCN programs. Nondeterministic choice is rarely required in parallel programming. However, it can be important in reactive applications.

We first illustrate the use of nondeterministic choice with a trivial example. We may rewrite the max procedure given earlier as follows. Note that the two implications are not mutually exclusive. If x == y, either implication may be taken. This program is nondeterministic in the sense that the action that it performs is not determined solely by its input, although of course the answer computed is still determined precisely by the input.

```
max(x,y,z)
{ ? x >= y -> z = x,
    x <= y -> z = y
}
```

We now consider a reactive programming example. A procedure switch has two definitional inputs corresponding to the outputs of two sensors in a mechanical device. If either sensor is activated, the corresponding input variable will be given a value. The switch procedure is to return a result value if either sensor is activated, with the value specifying which sensor was activated.

```
switch(sensor1,sensor2,alarm)
{ ? data(sensor1) -> alarm = 1,
    data(sensor2) -> alarm = 2
}
```

The guard test data succeeds as soon as its argument has a value. Hence, the output variable alarm takes value 1 if sensor1 is activated and 2 if sensor2 is activated. It can take either value if both are activated.

Choice Composition used for three purposes:

- Choosing between alternatives.
- Synchronization.
- Nondeterministic choice.

5.6 Definitional Variables as Communication Channels

Consider two procedure calls (processes), a producer and a consumer, that share a definitional variable, x.

```
producer(x), consumer(x)
```

The two processes can use the shared variable to communicate data, simply by performing read and write operations on the variable. For example, assume that the producer is defined to write the variable, as follows.

```
producer(x)
{|| x = "hello" }
```

The definition `x = "hello"` has the effect of communicating the message `"hello"` to the consumer. The consumer receives this value simply by reading (examining) the variable. For example, the following consumer procedure checks to see whether `x` has the value `hello`. Note the use of choice composition and the `default` guard.

```
consumer(x)
{ ? x == "hello" -> stdio:printf("Hello",{},_),
     default -> stdio:printf("Huh?",{},_)
}
```

The shared definitional variable `x` is used here to both communicate a value between the producer and consumer and to synchronize the actions of these processes. The shared definitional variable can be thought of as a *communication channel.*

The use of definitional variables to specify communication has two advantages. First, it avoids the distinction that is made in many parallel languages between interprocessor and intraprocessor communication. This means that no special "packing" or "unpacking" operations need be performed when communicating. This in turn facilitates the retargetting of programs to different parallel computers. Second, it provides great flexibility in the communication strategies that can be specified. In particular, it is possible (as we shall see below) to include variables in data structures and hence to establish dynamic communication structures.

An apparent difficulty of this formalism is that each definitional variable can be used only to communicate a single value. Fortunately, this is not the case. We show in § 5.9 below how a single shared variable can be used to communicate a *stream* of messages between processes.

5.7 Specifying Repetitive Actions

We have now encountered the constructs used in PCN to express concurrent and sequential execution, communication between concurrent computations, and state change within sequential computations. We need one more construct before we can build large programs, namely, a mechanism for specifying repeated actions.

You are probably familiar with the use of iteration to specify repetition. For example, in Fortran we may write `do i=1,10` to specify 10 repetitions of a loop, with `i` ranging from 1 to 10. PCN does not use iteration but rather *recursion* as its fundamental repetition construct. You will be familiar with recursion if you have used C (or Prolog, Strand, or Lisp); it tends to be more verbose than iteration, but has the advantages of allowing richer repetition structures and of working well with definitional variables.

We introduce the use of recursion in PCN with a simple example. Consider the following procedure, which computes the sum of the elements with indices in the range from..to in array. This procedure is defined in terms of a choice composition with a parallel composition as the body of the first implication and a simple definition statement as the body of the second implication.

Module sumarray.pcn: Version 1

```
sum_array(from,to,array,sum)
{ ? from <= to ->
        {|| sum_array(from+1,to,array,sumrest),
            sum = array[from] + sumrest
        },
    from > to -> sum = 0
}
```

The first implication states that if from <= to, then the sum of elements from..to is the value of element array[from] plus the sum of elements from+1..to. The second implication defines the sum to be 0 in the case when from > to.

This procedure uses recursion to repeat the summation over all the elements of the array. A recursive procedure normally specifies two alternative courses of action: continuation and termination. These are combined in a choice composition with guards specifying associated continuation and termination conditions.

In the example, the continuation action consists of summing array[index] and sumrest, and making a recursive call to sum_array to compute sumrest; these actions are to be performed if from <= to. The termination action consists of defining sum = 0; this is to be performed if from > to.

Recursive procedure specifies:

- Termination condition and actions.
- Continuation condition and actions.

Parallel algorithms based on divide-and-conquer techniques frequently make multiple recursive calls to the same procedure. For example, the following program implements a divide-and-conquer algorithm for summing the elements of an array. The task of summing an array is recursively decomposed into the tasks of summing the left and right subarrays.

Module sumarray.pcn: Version 2

```
sum_array(from,to,array,sum)
{ ? from < to ->
        {|| sum_array(from,(from+to)/2,array,sumleft),
            sum_array((from+to)/2+1,to,array,sumright),
            sum = sumleft + sumright
        },
    from == to -> sum = array[from]
}
```

This example makes apparent the advantages of recursion as a repetition construct in a parallel language: the doubly-recursive formulation of sum_array exposes concurrency that is not directly available in an iterative solution.

5.8 Tuples

The programs presented thus far have all dealt with simple data structures: characters, integers, double precision numbers, and arrays of the same. These data structures will be familiar to most readers from sequential languages such as Fortran and C. PCN also provides another sort of data structure called the *tuple*. Similar data structures are used in symbolic languages such as Prolog, Strand, or Lisp.

A tuple is a definitional data structure used to group together other definitional data structures. A tuple has the following general form.

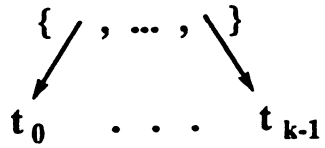
$$\{ \texttt{term}_0, \ldots, \texttt{term}_{k-1} \} \qquad (k \geq 0)$$

where $\texttt{term}_0$, ..., $\texttt{term}_{k-1}$ are definitional data structures. The following are all valid tuples.

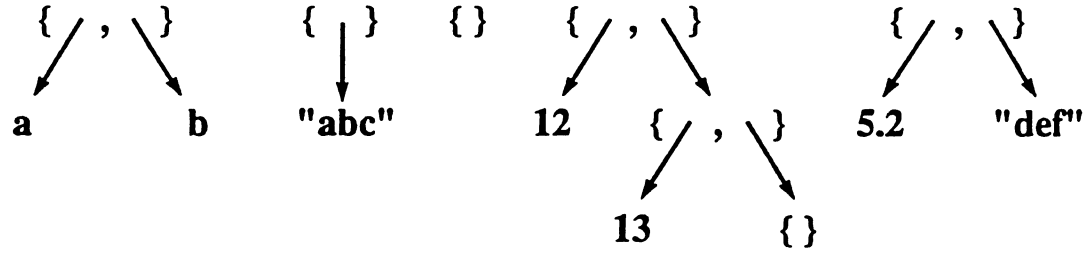
```
{a,b}    {"abc"}    {}    {12,{13,{}}}    {5.2,"def"}
```

Note that tuples can be nested: in the fourth tuple on the previous line, the tuple {} is nested inside the tuple {13,{}}, which is in turn nested inside the tuple {12,{13,{}}}. Note also that tuples can contain elements of different types.

It is useful to think of tuples as representing trees. A tuple $\{t_0, \ldots, t_{k-1}\}$ represents a tree with a root and k offspring.



The tuples listed above can be drawn as follows.



Building Tuples. Tuples can be written in a program, either as an argument to a procedure call or as the right-hand side of a definition statement. For example, the block

```
{|| proc(1,{x,y,{z}}), x = "abc", y = {123} }
```

invokes a procedure proc with the tuple {"abc",{123},{z}} as its second argument.

Alternatively, the primitive operation make_tuple can be used to build a tuple of specified size, with each argument a definitional variable. For example, the call

```
make_tuple(3,tup)
```

defines tup to be the three-tuple {_, _, _}.

Accessing Tuples. Tuple elements can be referenced in the same way as array elements: $t[i]$ is element i of a tuple t, for $0 \leq i < \texttt{length}(t)$. Hence, the statements

```
make_tuple(3,tup), tup[0] = "abc", tup[1] = {123}, tup[2] = {z}
```

produce the tuple passed as an argument to proc previously.

The guard test "?=" (match) can be used to decompose a tuple into its constituent components. A match has the general form

$$\texttt{tup ?= } \{t_0, \ldots, t_{k-1}\},$$

where the t_i are either definitional variables or nonvariable terms. A match succeeds if tup has arity k and each of its arguments matches the corresponding t_i, suspends if tup is not defined or if one of the matches with a t_i suspends, and fails otherwise. A variable t_i is defined to be the corresponding tup argument. For example, the match

```
tup ?= {"abc", a, {b}}
```

succeeds if tup = {"abc",{123},{z}}, defining a = {123} and b = z. It suspends if tup = {x,{123},{z}}, as the first element of the matching typle is "abc", but the first element of tup is the undefined variable x. It fails if tup = {"def",{123},{z}}, as the first element of the right-hand side tuple ("abc") does not match the first element of tup ("def").

Comparing Tuples. The guard tests == and != can be used to compare tuples as well as strings, numbers, and arrays. An equality test `x == y` succeeds if `x` and `y` are tuples with the same arity and corresponding subterms are also equal. The equality test is applied to subterms left to right; if any subterm test fails or suspends, the overall test also fails or suspends, respectively. The test also fails if `x` and `y` have different arities. An inequality test `x != y` succeeds if `x == y` would fail, fails if `x == y` would succeed, and suspends otherwise.

Syntactic Sugar. PCN provides "syntactic sugar" for several common uses of tuples. For example, it is common to want to place a string in the first element of a tuple, as a label. This usage is supported by an alternative notation, which permits

$$\{\texttt{"string"}, t_1, \ldots, t_{k-1}\}$$

to be written as

$$\texttt{string}(t_1, \ldots, t_{k-1})$$

It is important not to confuse this syntax with a function call. (The only functions supported by PCN are the arithmetic operations allowed in expressions and the `length` function, so there should never be any cause for confusion.)

List Notation. A *list* is a two-tuple in which the first element represents the head of the list and the second element the tail. By convention, the zero-tuple ({}) represents the empty list. For example, the structure {1,{2,{3,{}}}} is the list containing the numbers 1, 2, and 3.

This notation is clumsy, so PCN provides an alternative syntax: a list {h,t} may be written as [h|t], the empty list as [ ], a list such as {1,{2,{3,{}}}} as [1, 2, 3], and a list such as {1,{2,{3,tail}}} as [1, 2, 3|tail].

Example: List Length. The procedure `listlen` computes the length `len` of a list `l`. For example, a call `listlen([1,2,3,4],len)` gives the result `len = 4`. Note the use of an auxiliary procedure `listlen1`, which accumulates the length so far in `acc` and then returns the final result as `len`.

```
listlen(l,len)
{|| listlen1(l,0,len1)}

listlen1(l,acc,len)
{ ? l ?= [_|l1] -> listlen1(l1,acc+1,len),
    default -> len = acc
}
```

Example: Building a List. The procedure `buildlist` builds a list `l` of length `len`. For example, a call `buildlist(4,l)` gives the result `l = [4,3,2,1]`.

```
buildlist(len,l)
{ ? len > 0 ->
    {|| l = [len|l1],
        buildlist(len-1,l1)
    },
    default -> l = []
}
```

Example: List Transducer. The procedure `listadd` is an example of what is called a *list transducer*. It traverses one list and constructs another containing the result of applying a simple operation to each element in the first list: in this case, the operation is simply to add one to each element. For example, a call `listadd([1,2,3,4],nl)` gives the result `nl = [2,3,4,5]`.

```
listadd(l,nl)
{ ? l ?= [e|l1] ->
    {|| nl = [e+1|nl1],
        listadd(l1,nl1)
    },
    default -> nl = []
}
```

5.9 Stream Communication

We have seen how two or more concurrent computations that share a definitional variable can use that variable to exchange data. The *producer* of the data simply defines the shared variable to be the data to be communicated (e.g., `x = "hello"`). The *consumer*(s) of the data can then use the data in computation.

A shared definitional variable would not be very useful if it could only be used to exchange a single value. Fortunately, there are simple techniques that allow a single definitional variable to be used to communicate many values. The most important of these is the *stream*. A stream is a data structure that permits communication of a sequence of messages from a producer to one or more consumers. A stream acts like a queue: the producer places elements on one end, and the consumer(s) take them off the other.

By convention, stream communication is implemented in PCN in terms of list structures. Imagine a producer and a consumer sharing a variable `x`. The producer

defines `x = [msg|xt]` and the consumer matches `x ?= [msg|xt]`. The effect of these operations is to both communicate `msg` to the consumer and create a new shared variable `xt` that can be used for further communication. This process can be repeated arbitrarily often to communicate a stream of messages from the producer to the consumer. Hence, a stream is a list structure, incrementally constructed by a producer and deconstructed by a consumer. The empty list ($\square$) is used to represent the end of a stream.

Example: Summing Squares. We illustrate the stream communication protocol in a program that computes the sum of the squares of the integers from 1 to N. We decompose this problem into two subproblems: constructing a stream of squares and summing a stream of numbers. The first subproblem is solved by the procedure `squares`, which recursively produces a stream (i.e., list) of messages N^2, $(N-1)^2$, ..., 1. The second subproblem is solved by the procedures `sum` and `sum1`, which recursively consume this stream (list). The auxiliary procedure `sum1` accumulates the sum so far in `sofar` and returns the final result as `sum`.

Note the structure of the producer (`squares`) and consumer (`sum1`) procedures in the following program. Both are recursively defined. In the producer, the recursive case incrementally constructs a list `sqs` of squares by defining `sqs = [n*n|sqs1]` and calling `squares` to compute `sqs1`; the termination case defines `sqs = []`. In the consumer, the recursive case deconstructs a list `ints` of integers by matching `ints ?= [i|ints1]` and calling `sum1` to consume the rest of the messages; the termination case returns a result.

Module sumsquares.pcn

```
sum_squares(N,sum)
{|| squares(N,sqs), sum(sqs,sum) }

squares(n,sqs)                          /* Producer:          */
{ ? n > 0 -> {|| sqs = [n*n|sqs1], /*   Produce element, */
                 squares(n-1,sqs1) /*    & recurse       */
              },
    n == 0 -> sqs = []                  /*   Close list.      */
}

sum(ints,sum)
{|| sum1(ints,0,sum)}

sum1(ints,sofar,sum)                    /* Consumer:          */
{ ? ints ?= [i|ints1] ->                /*   Consume element, */
        sum1(ints1,sofar+i,sum),        /*    & recurse       */
    ints ?= [] -> sum = sofar           /*   End of list: stop*/
}
```

Send/Receive Operations. Some readers may find it useful to think of a stream as an abstract data type on which four operations are defined: `send`, `close`, `recv`, and `closed`. The first two are procedure calls used by a stream producer, and the latter two are guard tests used by a stream consumer. All take a definitional variable (`s`) as an argument; `send` and `recv` also return a new definitional variable (`s1`) representing a new stream to be used for the next communication.

`send(s,msg,s1)` : Send `msg` on stream `s`, returning as `s1` a new stream for subsequent communication.

`close(s)` : Close stream `s`.

`recv(s,msg,s1)` : Succeed if a message is pending on stream `s`, defining `msg` to be the message and `s1` the new stream.

`closed(s)` : Succeed if stream `s` has been closed.

These operations can be defined by the following macros.

File `sendrecv.h`

```
#define send(s,msg,s1) s = [msg|s1]
#define close(s)       s = [ ]
#define recv(s,msg,s1) s ?= [msg|s1]   /* Guard test */
#define closed(s)      s == [ ]        /* Guard test */
```

These definitions can be placed in a file (e.g., `sendrecv.h`) and included in your programs, if you prefer to think in terms of `send` and `recv` operations instead of definition and match operations on streams. For example, the `squares` and `sum1` procedures presented previously (module `sumsquares.pcn`) can be rewritten as follows.

```
#include "sendrecv.h"              /* Include macros */

squares(n,sqs)
{ ? n > 0 -> {|| send(sqs,n*n,sqs1),
                 squares(n-1,sqs1)
              },
    n == 0 -> close(sqs)
}

sum1(ints,sofar,sum)
{ ? recv(ints,i,ints1) -> sum1(ints1,sofar+i,sum),
    closed(ints)       -> sum = sofar
}
```

However, it would be a mistake to think of lists as simply a clumsy notation for streams, and to restrict your use of streams to the four basic operations provided in `sendrecv.h`. The fact that streams are data structures that can be manipulated in the same way as any other data structure provides enormous flexibility.

Example: Stream Filter. We illustrate this flexibility with a list transducer that filters a stream `x`, generating a stream `y` identical to `x` but with no consecutive duplicates. (For example, a call `filter([1,1,4,3,5,5,2],y)` defines `y = [1,4,3,5,2]`.)

This is not a complex example. However, it illustrates several stream-processing strategies. Note in particular the use of the match operator to check for two pending messages (as follows: `x?=[msg1,msg2|x1]`), the pushing of unused elements back onto the stream in the recursive calls (e.g., `filter([msg2|x1],y)`), and the definition of `y` to be all remaining elements of `x` in the termination case (`y = x`).

```
filter(x,y)
{ ? x ?= [msg1,msg2|x1] ->
    { ? msg1 == msg2 -> filter([msg2|x1],y),
      default -> {|| y = [msg1|y1],
                     filter([msg2|x1],y1)
                 }
    },
    default -> y = x      /* x is [msg] or [] */
}
```

5.10 Advanced Stream Handling

The stream construct provides direct support for one-to-one communication: that is, communication between a single producer and a single consumer. It also supports broadcast communication: that is, generation of a single stream to be received by several consumers. For example, in the composition

```
{|| producer(s), consumer(s), consumer(s) },
```

both consumers receive any values generated by the producer.

Three other communication patterns are also important in practical applications: many-to-one, one-to-many, and bidirectional. The first and second are supported in PCN by specialized primitives. The third is achieved by means of a specialized programming technique.

Mergers: Many-to-One Communication. A *merger* is a PCN system program that allows the construction of an output stream that is the nondeterministic interleaving of a dynamically varying number of input streams. (The merger is hence the second source of nondeterminism in PCN, with choice composition being the first.) The only constraint on message order in the output stream is that the order of messages from individual input streams be preserved. A merger is created with a procedure call of the form

```
sys:merger(in,out),
```

where `in` is an initial input stream and `out` is the output stream. An additional input stream `newin` is registered with the merger by appending a message of the form `{"merge",newin}` to any open input stream. An input stream is closed in the usual way (`s = []`); the output stream is closed automatically when all input streams are closed.

The following code fragment illustrates the use of the merger. This organizes communication between two producer processes and a single consumer, so that the consumer receives on `instream` an intermingling of the streams generated by the two producers.

```
{|| producer(s1), producer(s2)
    instream = [{"merge",s1},{"merge",s2}],
    sys:merger(instream,outstream),
    consumer(outstream)
}
```

Note that the merger must be able to determine whether or not each input message is a `{"merge",_}` term. Hence, messages of the form `var` or `{var,term}` (where `var` is an undefined variable) should not be sent to a merger: these will cause the merger to delay until `var` is given a value.

Distributors: One-to-Many Communication. A *distributor* is a PCN system program that routes each message received on its input stream to one of several output streams. A message of the form `{N,Msg}` cause the distributor to route `Msg` to the `Nth` output stream. A distributor is created with a call of the form

```
sys:distribute(N,In),
```

where `N` is the number of output streams needed and `In` is the input stream. Messages can then be sent to the distributor to register output streams. We register a stream `S` as the `Nth` output stream by sending a message with the form

```
{"attach",N,S,Done},
```

where `Done` is a definitional variable that is defined by the distributor to signal that the stream `S` has been registered.

We request the distributor to route a message `Msg` to the `Nth` output stream by sending the following message.

```
{N, Msg}
```

It is important to ensure that a stream has been registered before requesting that a message be routed to that stream. One way of doing this is to register all streams with the distributor before sending any messages. The following program achieves this. A call `make_distributor(in,ss)` creates a distributor with `ss` as its output streams. (The number of streams in `ss` is computed by the procedure `listlen` defined previously.) The input stream `in` is passed to this distributor only after all output streams have been registered.

```
make_distributor(in,ss)
{|| listlen(ss,len),
    sys:distributor(len,tod),
    register(0,ss,tod,in)
}

register(i,ss,tod,in)
{ ? ss ?= [s|ss1] ->
        {|| tod = [{"attach",i,s,done}|tod1],
            data(done) -> register(i+1,ss1,tod1,in)
        },
    ss ?= [] -> tod = in
}
```

If the input stream to the distributor is closed (`In = []`), then the distributor closes all registered output streams and shuts down.

Two-Way Communication. Many parallel algorithms require two-way communication between concurrently executing processes. In some cases, this can be achieved by defining two communication streams, one for use in each direction. However, it is also possible to achieve two way communication with a single definitional variable, by using a technique called an *incomplete message*.

We introduce the incomplete message technique with a simple example. Consider a program input capable of providing boundary conditions for two different numerical models (e.g., spectral and finite difference). This can be composed with a procedure implementing a particular numerical model, as follows.

```
input(xs), model(xs)
```

The definitional variable `xs` will be used to implement a stream.

The first thing that `input` does is to query the program it is composed with, to determine that program's input requirements. It does this by sending a message of the form

```
{"what_input",response},
```

where response is an undefined definitional variable. The other program (which of course must be ready to accept such a message) defines response to specify the required input type, allowing the first program to read response and generate the appropriate input data.

Possible definitions for input and model are as follows. In this example, the model procedure specifies that it expects input in terms of spectral coefficients by defining response = "spectral". This communication causes the input procedure to execute spectral_input.

```
input(x)
{|| x = [{"what_input",response}|xs],
    { ? response == "spectral" -> spectral_input(xs),
        response == "finite_diff" -> fd_input(xs)
    }
}

model(x)
{ ? x ?= [{"what_input",response}|xs] ->
    {|| response = "spectral",
        process_input(xs)
    }
}
```

In this example, a single shared variable, xs, has been used to achieve two-way communication. This is a simple example of a very powerful programming technique which can be used to establish a wide variety of communication patterns. The key idea is for one process to define a shared variable to be a tuple containing "holes" (undefined variables). Consumer(s) of this tuple can then fill in these holes (define the variables) to communicate additional values to the original producer or even to other consumers.

We use a more complex example to strengthen understanding of the incomplete message technique. Consider the problem of exploring a large search space with a heuristic search method. Assume that it is possible to define multiple *searchers*, each capable of exploring part of the search space, and that individual searchers can improve their efficiency by exploiting global information about the best-known partial solution. We collect and disseminate global information by defining a *controller* process to which each searcher periodically sends information about its current best partial solution. The controller responds to each such message by updating its view of the best partial solution and returning the best known partial solution.

A PCN implementation of this search method provides each searcher with a stream to the controller and uses a merger to combine the multiple searcher streams into a single controller input stream. For example, the following code links two searchers and a controller.

```
{|| searcher(s1), searcher(s2),
    sys:merger([{"merge",s1},{"merge",s2}],s),
    controller(s)
}
```

The searcher is defined as follows. A call to `first_attempt` yields an initial approximate solution (`value`), which is passed to the recursively-defined procedure `search`. The `search` procedure sends the approximate local solution to the controller in a `{value,response}` tuple, where `response` is an undefined definitional variable used to communicate information back from the controller to the searcher. Depending on the `response` received from the controller, the searcher either terminates or calls `next_attempt` and repeats the process.

The controller receives a stream of approximate solutions from the workers. It processes each message by calling `improve_estimate` to improve its own estimate of the global best solution, and returning either this estimate or the signal "stop" (indicating that a solution has been found) to the searcher.

```
searcher(trials)
{|| first_attempt(value),
    search(trials,value)
}

search(trials,value)
{|| trials = [{value,response}|trials1],
    { ? response == "stop" -> trials1 = [],
        default ->
        {|| next_attempt(value,response,next_value),
            search(trials1,next_value)
        }
    }
}

controller(trials,bound)
trials ?= [{value,response}|trials1] ->
    {|| improve_estimate(bound,value,newbound,result),
        { ? result == "solution" -> response = "stop",
            default -> response = newbound
        },
        controller(trials1,newbound)
    }
```

Specialized Communication Structures:

- Many-to-one: `merger`.
- One-to-many: `distributor`.
- Bidirectional: incomplete message.

5.11 Interfacing Parallel and Sequential Code

The two worlds of parallel and sequential, definitional and mutable, have so far been regarded as distinct. In practice, the two worlds must interact whenever a sequential program component is integrated into a concurrent program. Such interactions are governed by three simple rules. The first restricts the way in which mutable variables can be used within parallel blocks, while the second and third specify copying operations performed by the PCN compiler when data is transferred between the definitional and mutable worlds by defining a definition in terms of a mutable, or vice versa. This copying avoids aliasing between state maintained in different sequential threads, and hence ensures that state change within individual threads does not lead to time-dependent interactions with concurrently executing processes.

Mutable Variables and Parallel Composition. Mutable variables may occur in parallel compositions, but only if their usage obeys the following rule.

Rule 1: A mutable variable can be shared by blocks in a parallel composition only if no block modifies the variable.

This restriction prevents errors due to time-dependent, nondeterministic updates to a mutable variable. The restriction is not currently enforced by the compiler, and so the programmer must be careful to ensure that all programs are valid.

Note that there is no similar restriction on the use of definitional variables within sequential blocks.

Mutable → Definition. The following rule states what happens when a definitional variable is defined in terms of a mutable variable.

Rule 2: When a mutable occurs on the right hand side of a definition statement, the current value of the mutable is snapshotted (copied) and the definition then proceeds as if a definitional value were involved.

For example, in the following code, `c = 5` and `d = 4` when computation is complete.

```
proc1(c,d)
int a;
{ ; a := 3,
    c = 2 + a,
    a := 4
    d = a
}
```

Snapshotting a mutable array creates a definitional copy of the array that can be read but not modified. For example, in the following c is defined to be a copy of the mutable array a. Subsequent changes to a do not affect the value of the definitional array c.

```
proc2(c,d)
int a[5];
{ ; initialize(a),
    c = a,
    ...
}
```

Definition → Mutable. The following rule states what happens when a mutable variable is assigned an expression involving a definitional variable.

Rule 3: When a definitional variable occurs on the right-hand side of an assignment, the assignment suspends until the variable has a value and then proceeds.

For example, if c is a definition with value 3 in the following program, then a has value 5 after the assignment.

```
proc3(a,c)
int a, b;
{ ; b := 2
    a := b + c
}
```

Note that if the right-hand side of the assignment is not an expression, then the assignment will copy the definitional value into the mutable variable. For example, in the following code fragment, the definitional value c is copied into the mutable array a. The array a can be modified subsequently without affecting c.

```
int a[5];
a := c
```

Example. The following example illustrates the use of copying to avoid aliasing. The procedure `proc` has two definitional arguments: it produces as `output` the result of applying a transformation `solve` to `input`. It calls the procedure `solve` to effect the transformation; this is defined to operate on mutable data structures. Hence, `proc` declares a local mutable array `temp`, assigns `temp` the value `input`, applies `solve` to `temp`, and then defines `output` to be the updated value of `temp`. Two copying operations take place, from `input` to `temp`, and from `temp` to `output`.

```
proc(input,output)
double temp[SIZE];
{ ; temp := input,
    solve(temp),
    output = temp
}
```

5.12 Review

PCN encourages a compositional approach to parallel programming, in which complex programs are built up by the parallel composition of simpler components. Program components composed in parallel execute concurrently. They communicate by reading and writing definitional (single-assignment) variables. The use of definitional variables avoids time-dependent interactions, allowing individual components to be understood in isolation. In addition, read and write operations on definitional variables can be implemented efficiently on both shared memory and distributed memory parallel computers. Hence, parallel composition and definitional variables address three of the concerns listed at the beginning of this chapter: *concurrency*, *compositionality*, and *mapping independence*.

The choice operator is used to encode conditional execution and synchronization. It also provides a means of introducing *controlled nondeterminism* into programs. (The merger is the other mechanism used to specify nondeterministic actions in PCN programs.)

The sequential composition operator and mutable variables together provide a mechanism for integrating state change into definitional programs. This state change may be performed in PCN or in lower-level sequential languages.

A final aspect of PCN which may be unfamiliar to some readers is its use of tuples and recursion. These constructs provide support for symbolic processing. They augment arrays, iteration, and other language constructs provided by languages such as Fortran and C for numeric processing. An increasing number of applications have both numeric (regular, floating point) and symbolic (irregular, rule based)

components. PCN's symbolic processing capabilities are intended to support such mixed-mode applications.

6 Programming Examples

We present PCN programs that solve programming problems concerned with list and tree manipulation, sorting, and a two-point boundary value problem.

6.1 List and Tree Manipulation

Membership in a List. Develop a program `member` with arguments `l`, `e` and `r`, where `l` is a list, and at termination of execution of the program, `r = TRUE` if and only if `e` appears in list `l`. Assume that `FALSE = 0` and `TRUE = 1`, to be consistent with C.

```
#define TRUE  0
#define FALSE 1

member(l,e,r)
{ ? l ?= [v|l1], v == e -> r = TRUE,
    l ?= [v|l1], v != e -> member(l1,e,r),
    l ?= [] -> r = FALSE
}
```

Membership in a List (Mutables). Now consider a program with the same specification, except that `e` and `r` are now mutables. The mutable `r` is to be set to `TRUE` or `FALSE`; `e` (and of course `l`) should not be changed.

```
#define TRUE  0
#define FALSE 1

member(l,e,r)
int e, r;
{ ? l ?= [v|l1], v == e -> r := TRUE,
    l ?= [v|l1], v != e -> member(l1,e,r),
    l ?= [] -> r := FALSE
}
```

The only difference between the two programs is the addition of the type declarations and the substitution of the `:=` operator.

Reversal of a List. Develop a program `reverse` with arguments `x`, `b` and `e`, that defines `b` to be the list of elements in `x`, in reverse order, concatenated with `e`. For example, if `x = ["A","B"]` and `e = ["C","D"]`, then `b` is to be defined as `["B","A","C","D"]`. (The name `b` stands for the beginning of the reversed list, and `e` stands for the end of the reversed list.)

```
reverse(x,b,e)
{ ? x ?= [v|xs] ->  reverse(xs,b,[v|e]),
    x ?= [] -> b = e
}
```

This program can be used to simply reverse a list by calling it with `e = []`. For example, a call `reverse([1,2,3],b,[])` yields `b = [3,2,1]`.

The `reverse` procedure illustrates an important programming technique called the *difference list*. A call to `reverse` constructs a list `b` consisting of the values computed by `reverse` *followed by* the values provided as `e`. This allows lists constructed in several computations to be concatenated without further computation. For example, the calls

```
reverse([1,2,3],b,e), reverse([4,5,6],e,[])
```

construct the list `[3,2,1,6,5,4]`.

Height of a Binary Tree. Develop a program `height` with arguments `t` and `z`, where `t` is a binary tree, and `z` is to be defined to be the height of the tree. A tree `t` is either the empty tuple, {}, or a 3-tuple {`left, val, right` }, where `left` and `right` are the left and right subtrees of `t`.

```
height(t,z)
{ ? t ?= {left, _, right} ->
    {|| height(left, l), height(right, r),
        { ? l >= r -> z = l+1,
            l <  r -> z = r+1
        }
    },
    t ?= {} -> z = 0
}
```

The program can be read as follows. The height of a nonempty tree is 1 plus the larger of the heights of the left and right subtrees. (The heights of the subtrees are determined by two recursive calls to `height`.) The height of an empty tree is 0.

Preorder Traversal of a Binary Tree. Develop a program preorder with arguments t, b and e, where t is a binary tree, and b and e are lists. Binary trees are represented using tuples as in the last example. List b is to be the list consisting of the val of all nodes of the tree in preorder, concatenated with list e. (A traversal of a tree in preorder visits the root, then the left subtree, and finally the right subtree.)

```
preorder(t,b,e)
{ ? t ?= {left,val,right} ->
        {|| b = [val|m1],
            preorder(left,m1,m2),
            preorder(right,m2,e)
        },
    t ?= {} -> b = e
}
```

The program uses the difference list technique introduced previously in the reverse example: each call to preorder constructs a list b consisting of the elements in its subtree t followed by the supplied list e.

6.2 Quicksort

We present an implementation of the well-known quicksort algorithm, qsortD, which uses lists of definitional variables; later, we provide an *in-place* quicksort, qsortM, that uses mutable arrays. It is instructive to compare the two programs: the definitional program is significantly shorter and easier to understand than the mutable program. However, it makes less efficient use of memory.

Definitional Quicksort. Program qsortD has two input arguments, x and e, and one output argument, b: x and e are definitional variables that are not defined by the program, and b is a definitional variable that is defined by the program. All three are lists of numbers. The output b is specified to be the list x sorted in increasing order, concatenated with list e. For example if e = [5, 4] and x = [2, 1], then b = [1, 2, 5, 4]. If e is the empty list, then b is x sorted in increasing order.

```
qsortD(x,b,e)
{ ? x ?= [mid|xs] ->
        {|| part(mid,xs,left,right),
            qsortD(left,b,[mid|m]),
            qsortD(right,m,e)
        },
    x ?= [] -> b = e
}

part(mid,xs,left,right)
{ ? xs ?= [hd|tl] ->
    { ? hd <= mid ->
            {|| left = [hd|ls],  part(mid,tl,ls,right) },
        hd >  mid ->
            {|| right = [hd|rs], part(mid,tl,left,rs) }
    },
    xs ?= [] -> {|| left = [], right = [] }
}
```

The qsortD procedure operates as follows. If x is nonempty, let mid be its first element and let xs be the remaining elements. The call part(mid,xs,left,right) defines left to be the list of values of xs that are at most mid, and right to be the list of values of xs that exceed mid. Call qsortD(right,m,e), thus defining m to be the sorted list of right appended to e. Call qsortD(left,b,[mid|m]), thus defining b to be the sorted list of left followed by mid followed by m. Otherwise, if x is the empty list, then define b to be e.

The part procedure operates as follows. If xs is not empty, then let hd and tl be the head and tail (respectively) of xs. If hd is at most mid, define ls and right by part(mid,tl,ls,right), and define left as hd followed by ls. If hd exceeds mid, define left and rs by part(mid,tl,left,rs), and define right as hd followed by rs. If xs is the empty list, define left and right to be empty lists.

In-Place Quicksort. Program qsortM has two input parameters, l, and r, both of which are definitional variables, and one input-output parameter C, which is a one-dimensional mutable array of integers. Let C^{init} be the initial value of C, and let C^{final} be the value of C on termination of the program. Then C^{final} is to be a permutation of C^{init}, where C^{final}[l, ..., r] is C^{init}[l, ..., r] in increasing order, and the other elements of C are to remain unchanged. (If l $\geq$ r then C^{final} is C^{init}.)

```
qsortM(l,r,C)
int C[];
l < r ->
    { ; split(l,r,C,mid),
        qsortM(l,mid-1,C),
        qsortM(mid+1,r,C)
    }

split(l,r,C,mid)
int C[], left, right, temp;
l <= r ->
    { ; left := l+1, right := r, s = C[l],
        part1(l,r,C,s,left,right), temp := l,
        swap(temp,right,C), mid = right
    }

part1(l,r,C,s,left,right)
int C[], left, right;
left <= right ->
    { ; left_rightwards(r,C,s,left),
        right_leftwards(l+1,C,s,right)
        left <= right ->
            { ; swap(left,right,C),
                left := left + 1,
                right := right - 1
            },
        part1(l,r,C,s,left,right)
    }

left_rightwards(r,C,s,left)
int C[], left;
left <= r, C[left] <= s ->
    { ; left := left+1, left_rightwards(r,C,s,left) }

right_leftwards(l,C,s,right)
int C[], right;
right >= l, C[right] > s ->
    { ; right := right-1, right_leftwards(l,C,s,right) }

swap(i,j,C)
int i, j, C[], temp;
{ ; temp := C[i], C[i] := C[j], C[j] := temp }
```

Execution of `split(l,r,C,mid)` permutes `C` and assigns a value to `mid` such that $l \leq mid \leq r$, and such that all elements in `C[l, ..., mid-1]` are at most `C[mid]`, and all elements in `C[mid+1, ..., r]` exceed `C[mid]`.

The program `qsortM` operates as follows. If $l \geq r$, then `qsortM` takes no action, leaving `C` unchanged. If $l < r$, then `split` is called, and after `split` terminates execution, `C[l, ..., mid-1]` and `C[mid+1, ..., r]` are sorted independently.

The `split` program operators as follows. If $l > r$ then `split` terminates execution without taking any action. If $l \leq r$, then program `split(l,r,C,mid)` calls `part1(l,r,C,s,left,right)` after setting `left = l+1`, `right = r` and `s = C[l]`; program `part` leaves `s` unchanged, modifies `left` and `right`, and permutes elements of `C[l+1, ..., r]` so that, at termination of `part1`, `left = right +1`, and all elements in `C[l+1, ..., right]` are at most `s`, and all elements in `C[right+1, ..., r]` exceed `s`.

After termination of `part1`, program `swap` is called to exchange `C[l]` (which is `s`) with `C[right]`. After the swap, all elements in `C[l, ..., right-1]` are at most `s`, and `C[right] = s`, and all elements in `C[right+1, ..., r]` exceed `s`. The program terminates after `mid` is defined as `right`.

Program `part1` moves `left` rightwards and `right` leftwards until they cross (i.e., `left = right+1`).

6.3 Two-Point Boundary Value Problem

Our last programming example is a solution to a more substantial numerical problem. The problem that we consider arises when solving the linear boundary value problem in ordinary differential equations, namely

$$y' = M(t)y + q(t), \qquad t \in [a,b], \quad y \in R^n,$$
$$\text{such that} \quad B_a y(a) + B_b y(b) = d.$$

In most algorithms designed to solve this problem, the most computationally intensive task is the construction and solution of a linear algebraic system of equations, which typically has the form

$$\begin{bmatrix} B_a & & & & B_b \\ A_1 & C_1 & & & \\ & A_2 & C_2 & & \\ & & \ddots & \ddots & \\ & & & A_k & C_k \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ f_1 \\ f_2 \\ \vdots \\ f_k \end{bmatrix}.$$

Here each of the blocks has dimension $n \times n$, and k is often substantially larger than n. Construction of this system is trivially parallelizable. A more substantial challenge is to solve it in a parallel computing environment. It is important that the solution process be *stable* in a numerical sense; otherwise, the computed answer may be hopelessly inaccurate. Simple algorithms such as block elimination are therefore not appropriate. The algorithm described here uses a "structured orthogonal factorization" technique, in which orthogonal transformations are used to compress each

two successive block rows of the linear system into a single block row. This produces a "reduced" system that has the same structure as the original system, but is half the size. The compression process can be applied recursively until a small system

$$\begin{bmatrix} B_a & B_b \\ \tilde{A}_1 & \tilde{C}_1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ \tilde{f}_1 \end{bmatrix}$$

remains.

The PCN code that implements this algorithm creates a set of k processes connected in a tree structure. A wave of computation starts at the k/2 leaves of the tree and proceeds up the tree to the root. The leaves perform the initial compression described above, while at the higher levels of the tree the compression is applied recursively, and at the root the small system above is solved. Finally, computation propagates down the tree to recover the remaining elements of the solution vector.

Input to the PCN code is provided at each leaf i ($0 \leq i < k/2$) as two $n \times n$ blocks (A_i and C_i) and one n vector (f_i), and at the root as two $n \times n$ blocks (B_a and B_b) and one n vector (d).

The PCN code consists of two main parts. The first part is the code that creates the process tree. This creates a `root` process and calls a doubly-recursive `tree` procedure to create $k/2$ `leaf` processes and $k/2-1$ `nonleaf` processes. Shared definitional variables (`strm, left, right`) establish communication channels between the nodes in the tree.

```
solve(k,t0,t1)
{|| root(strm)@root,
    tree(strm,{t0,t1},1,k/2)
}

tree(strm,as,from,to)
{ ? from == to -> leaf(from*2,strm,as),
    from<to -> {|| mid=from+(to-from)/2,
                   nonleaf(left,right,strm),
                   tree(left,as,from,mid),
                   tree(right,as,mid+1,to)
               }
}
```

The second part of the program defines the actions performed by the `leaf`, `nonleaf`, and `root` processes. We consider the `leaf` process first. A single leaf process initializes two sets of blocks — `a1`, `c1`, `f1`) and (`a2`, `c2`, `f2` — and then calls `compress` to produce `a`, `c`, `f`. It sends a message to its parent containing the computed values and slots for return values (`ybot`, `ytop`) which will be computed by its parent. The `recover` procedure delays until values are received for `ybot` and `ytop`, and then computes the solution, `y`.

```
leaf(id,parent,as)
double a1[MM],c1[MM],f1[M],a2[MM],c2[MM],f2[M],
       a[MM],c[MM],f[M],y[M],r[MM];
as ?= {t0,t1} ->
    { ; init_(id-1,a1,c1,f1,t0,t1),
        init_(id,a2,c2,f2,t0,t1),
        compress_(a1,c1,f1,a2,c2,f2,a,c,f,r),
        parent={a,c,f,ybot,ytop},
        recover(a1,c1,f1,r,ybot,ytop,y)
    }
```

The nonleaf procedure receives messages from left and right offspring. It calls compress to compress the a1, c1, f1 and a2, c2, f2 received from its offspring, producing a,c,f. These newly compressed values are communicated to the parent in the process tree. Once values for ytop and ybot are produced by the parent, the recover operation can proceed, producing ymid; values are then returned to the left and right offspring by the four definition statements.

```
nonleaf(left,right,parent)
double ymid[M],a[MM],c[MM],f[M],r[MM];
left ?= {a1,c1,f1,ybot1,ytop1},
right ?= {a2,c2,f2,ybot2,ytop2} ->
    { ; compress_(a1,c1,f1,a2,c2,f2,a,c,f,r),
        parent={a,c,f,ybot,ytop},
        recover_(a1,c1,f1,r,ybot,ytop,ymid),
        ybot1=ymid, ytop1=ytop,
        ybot2=ybot, ytop2=ymid
    }
```

The root process receives a single message containing the completely reduced blocks. It calls comp_root to perform the final computation, producing ybot1 and ytop1 which it returns to its offspring with two definitions.

```
root(child)
double ybot1[M],ytop1[M],ba[MM],bb[MM],brhs[M];
child ?= {a,c,f,ybot,ytop} ->
    { ; init_root_(m,ba,bb,brhs),
        comp_root_(a,c,f,ba,bb,brhs,ybot1,ytop1),
        ytop=ytop1, ybot=ybot1
    }
```

7 Modules

Recall from § 3 that a PCN program consists of one or more modules. Each module is contained in a separate file with a `.pcn` suffix. A module contains zero or more procedures.

Procedures in one module can invoke procedures in other modules by means of *intermodule calls.* An intermodule call has the following general form.

$$\text{module:procedure\_name}(\text{arg}_0, ..., \text{arg}_n)$$

A procedure can be invoked by an intermodule call only if it has been *exported* by the module in which it is defined. By default, all procedures in a module are exported. However, you can specify that only a subset of the procedures in a module are to be exported, by providing one or more `-exports` directives. An exports directive has the general form

$$\texttt{-exports(proc}_0\texttt{, ..., proc}_k\texttt{)}$$

and specifies that the module in which it appears exports procedures named by the strings $\texttt{proc}_0$, ..., $\texttt{proc}_k$. For example, the directive `-exports("procA","procB")` names `procA` and `procB` as exported.

In general, it is good practice to provide an `-exports` statement in each module, and to export only those procedures that are called from other modules.

PCN procedures can also import capabilities provided by the shell (§ 4.3). A different syntax is used for this purpose. We write the following.

```
! capability_call
```

For example, the call `! pp(x)` invokes the pretty printer capability on variable `x`.

A module can also contain `-foreign` directives, to specify the location of foreign procedures called by the module. This directive is described in § 9.2.

8 The C Preprocessor

The PCN compiler applies the C language preprocessor (`cpp`) to each PCN module before it compiles it. Hence, PCN programs can make use of cpp's capabilities, such as include files, macros, and conditional compilation. All three of these capabilities are used in the following program.

Module cpp_ex.pcn

```
-exports("go")

#include <stdio.h>
#define ARRAY_SIZE 10

go()
double a[ARRAY_SIZE];
{ ;
#ifdef OLD_VERSION
    stdio:printf("Old version\n",{},_),
#else
    stdio:printf("New version\n",{},_),
#endif
    do_something_with_array(a)
}
```

When the PCN compiler applies cpp to a PCN program, it automatically defines the symbol "PCN". This symbol can be used for conditional compilation. For example, the following header file can be used in both PCN and C components of a program, hence ensuring that the symbol `ARRAY_SIZE` is defined in the same manner everywhere. The `#ifndef` means that the declaration of `my_c_procedure()` is used only in the C compilation.

File cpp_ex.h

```
#define ARRAY_SIZE 10
#ifndef PCN
extern void my_c_procedure();
#endif
```

We can pass additional arguments to cpp when compiling PCN programs. For example, suppose we wish `OLD_VERSION` to be defined when compiling the program `cpp_ex.pcn` shown above. This can be achieved by using the `-D` flag to cpp. From the PCN shell we run the following command.

```
compile("cpp_ex","-DOLD_VERSION")
```

Several CPP arguments can be combined in a single string. If compiling using `pcncomp`, we use the `-cpp` flag, as follows.

```
pcncomp cpp_ex.pcn -cpp "-DOLD_VERSION"
```

9 Integrating Foreign Code

Programming examples presented thus far have focused on the use of PCN to compose procedures written in PCN. Exactly the same syntax and techniques can also be used to compose procedures written in other ("foreign") languages. Fortran and C are currently supported.

We deal here with the PCN/foreign interface, the mechanism used to import foreign procedures, and the mechanism used to link foreign object code with the PCN run-time system.

9.1 PCN/Foreign Interface

The PCN/foreign interface is defined as follows:

- The actual parameters in a call to a foreign program can be mutables or definitional variables of type `char`, `int`, or `double`, or arrays of these types.
- Execution of a foreign procedure delays until all definitional arguments have values.
- All parameter passing is by reference.
- A foreign procedure cannot modify definitional arguments.

The last restriction is not currently enforced by the compiler, so the programmer must be careful to ensure that all programs satisfy this constraint.

Note that a consequence of this definition is that all output generated by a foreign procedure *must be returned in mutable arguments.* Sufficient storage must be allocated for these mutables prior to calling the foreign procedure.

Two important differences exist between the execution of PCN and foreign procedures called from PCN: (1) PCN procedures can execute even if not all definitional arguments do have values. Indeed, they can compute values for definitional arguments. In contrast, foreign procedure calls delay until all definitional arguments have values, and can modify mutable arguments only. (2) PCN procedures can be passed tuples as arguments, whereas foreign procedures can be passed simple types only.

As parameter passing is by reference, arguments to a C procedure called from PCN must be declared as pointers. That is, the PCN types `char`, `int`, and `double` correspond to the C language types `char *`, `int *`, and `double *`.

Fortran. The PCN types `char`, `int`, and `double` correspond to the Fortran types `CHARACTER`, `INTEGER`, and `DOUBLE`. As Fortran also passes arguments by reference, no special treatment of arguments is required. It is necessary to append the suffix '_' to the name of a Fortran procedure called from PCN.

For example, the following PCN procedure calls a C procedure `natural_log(a,b)` to compute `b = ln(a)` and a Fortran procedure `power(a,b,c)` to compute $c = a^b$.

Note the '_' suffix on the call to pow and the use of a local mutable tmp for the result of the natural_log computation.

Module foreign.pcn

```
proc(a,b,c)
double a,b,c,tmp;
{ ; natural_log(a,tmp), power_(tmp,b,c)}
```

The C and Fortran procedures invoked by this program can be written as follows.

File cfile.c

```
#include <math.h>

void natural_log(a,b)
double *a,*b;
{ *b = log(*a); }
```

File ffile.f

```
SUBROUTINE POWER(A,B,C)
DOUBLE PRECISION A,B,C
C = A**B
RETURN
END
```

9.2 Importing Foreign Procedures

We specify the files in which foreign procedures are to be found by an -foreign directive in each PCN module that calls (i.e., imports) foreign procedures. An -foreign directive specifies zero or more object files, libraries, and archives, either as file names or as absolute path names. Path names containing ~ are not supported. The string $ARCH$ in a path name is expanded by the linker (see below) to the type of the machine on which the linker is being run (for example, delta, ipsc860, sun3, sun4, symmetry, NeXT; a complete list of machine names is given in Appendix B). This permits the same PCN object files to be used on different machines.

Assume that the C procedure natural_log and the Fortran procedure power are contained in the files cfile.c and ffile.f given above. We complete the PCN program given in the preceding section as follows:

Module foreign.pcn

```
-foreign("$ARCH$/cfile.o","$ARCH$/ffile.o")

proc(a,b,c)
double a,b,c,tmp;
{ ; natural_log(a,tmp), power_(tmp,b,c)}
```

This example indicates that the linker is to look in the subdirectory with name given by the type of machine (`delta`, `ipsc860`, `sun3`, `sun4`, `symmetry`, `NeXT`, etc.) for the object files `cfile.o` and `ffile.o`.

9.3 pcncc: The PCN Linker

Before we can execute a PCN program that composes code written in Fortran and C, we must link the Fortran and C object code with the PCN run-time system (pcn) to create a *custom pcn* capable of executing our program. This is achieved by using the PCN linker, pcncc.

In its most basic application, pcncc is invoked with the names of the PCN modules (the `.pam` files) comprising the multilingual application. It scans the `-foreign` directives of these modules and, if it can locate the named object files and libraries, links these with the PCN run-time system to produce a custom pcn called pcn. For example,

```
pcncc foreign.pam
```

will generate a custom pcn executable named pcn in the directory in which pcncc was executed. We may wish to give this custom pcn a special name. This is achieved with the `-o` option.

```
pcncc foreign.pam -o mypcn
```

A number of optional arguments to pcncc cause it to perform more specialized functions. Some of these are described here; a complete list of pcncc argument can be obtained by running:

```
pcncc -h
```

The following linker options are commonly used:

`-o <filename>` : Write the custom pcn into the file named `<filename>`.

`-fortran` : This indicates that the foreign code includes Fortran procedures, and requests pcncc to include libraries and initialization code needed by Fortran programs.

`-banner <string>` : Add the banner, `<string>`, to the normal PCN banner, in order to distinguish this custom pcn from the default pcn run-time system.

In addition, pcncc is used in slightly different ways on different machines; machine-specific aspects are discussed in § 19–23.

10 Using Parallel Computers

There are two aspects to running PCN programs on parallel computers: specifying how components of a concurrent computation are to be distributed among processors (*mapping*), and invoking the PCN run-time system with multiple processors.

10.1 Mapping

An important property of PCN is that, in general, the process mapping strategy applied in an application can change performance but cannot change the result computed. (The only exceptions to this rule are if foreign code uses global variables — e.g., common blocks — or if PCN code includes nondeterministic procedures.)

For this reason, it is common to develop PCN programs in two stages. First, program logic is developed and debugged on a workstation, without concern for process mapping. Second, a process mapping strategy is specified and its efficiency is evaluated on a parallel computer, typically by using the Gauge execution profiler.

Process mapping strategies are specified by annotating procedure calls in parallel compositions to indicate where they should execute within the parallel computer. In the rest of this section, we describe a simple set of annotations supported directly in the PCN compiler. More sophisticated mapping strategies are supported by specialized compiler tools; these are described in §PMT.

The annotations that we describe here present the programmer with an abstract view of a parallel computer as a set of `N` computing sites, or nodes, connected in a ring and numbered `0 ... N-1`. They allow the programmer to specify that a procedure should execute on the `I`th, next, previous, or a randomly selected node within this ring.

Any call to a PCN procedure (but not to a foreign procedure or PCN primitive) can be annotated with one of the following mapping directives, with the specified meaning. These annotations can also be applied to intermodule calls typed to the PCN shell.

`@I` Node number `I` modulo the number of nodes, `N`.

`@fwd, bwd` Next or previous node in the ring.

`@random` A randomly selected node.

On some multicomputers, such as the Symult s2010 (but not on the Intel iPSC/860, multiprocessors or networks), it is useful to distinguish the *host node* on which terminal interaction takes place and *compute nodes* on which computation takes place. On these machines, the host node is not included in the "virtual ring" addressed

by the annotations just listed; instead, we map to this processor using the following annotation:

`@host` Host node (in multicomputer).

On machines that do not distinguish between the *host node* and other *compute nodes*, the annotation `@host` maps to the highest numbered node, `N-1`.

We illustrate the use of mapping notations with a simple example.

Printing Nodes. This program prints a simple message from each of n nodes. It uses the `@fwd` mapping notation to map each recursive call to `printnodes` to a different processor. Note that if n is greater than the number of physical processors, the `printnodes` procedure will "wrap around," and more than one `printnodes` call may execute on each processor.

Module `parallel1.pcn`

```
printnodes(n)
n > 0 ->
    {|| stdio:printf("Hello from node %d\n",{n},_),
        printnodes(n-1)@fwd
    }
```

It is also possible to annotate procedures with variables that will be defined to be integers (`@I`) at run time. In this case, the variable name must be enclosed in single backward quotes (`` `var` ``) to distinguish it from other annotations. Hence, the preceding program could also be written as follows.

Module `parallel2.pcn`

```
printnodes(n)
n > 0 ->
    {|| stdio:printf("Hello from node %d\n",{n},_),
        printnodes(n-1)@`n`
    }
```

In this case, the `printnodes` procedure is mapped successively to nodes `n`, `n-1`, ...

10.2 Using Multiple Processors

A PCN program annotated with mapping directives will execute correctly on a single processor. However, in order for the mapping directives to improve (or degrade!) performance, it is necessary to run the program on multiple processors. You do this by invoking the PCN shell on multiple processors, then running the application program in the usual way.

The syntax used to start PCN on multiple processors varies according to the type of parallel computer. On multicomputers, we are generally required first to allocate a number of nodes and then to load PCN in these nodes. For example, on the Intel iPSC/860, we must log into the host computer (System Resource Manager: SRM) and type the following commands to allocate 64 nodes, run PCN, and finally free the allocated nodes.

```
% getcube -t 64
% runpcn pcn.ipsc860
% killcube
% relcube
```

On multiprocessors (e.g., Sequent Symmetry and some workstations), we generally need only to specify the number of processors, with a -n flag. For example, to run on 24 processors, we type the following.

```
% pcn -n 24
```

The -n option can also be used to spawn multiple communicating PCN nodes on a uniprocessor workstation. This is not normally useful, however, as all nodes will just multitask on that workstation's simple processor.

When running on a network, we generally need to provide a configuration file, indicating the names of the computers on which PCN is to run. See § 23 for nore information about running PCN on networks.

For details about how to use PCN on your computer, turn to the discussion of machine dependencies in §§ 19–23.

11 Process Mapping Tools

Recall from § 10 that PCN allows process mapping strategies to be specified by annotations attached to procedure calls: `@N`, `@fwd`, `@bwd`, and `@random`. These particular annotations are supported directly in the compiler, which makes them particularly convenient and easy to use.

We describe here an alternative set of process mapping tools, also based on annotations. These differ from the simple facilities described in § 10 in three respects:

- A richer set of process mapping strategies is supported.
- The implementation is considerably more efficient than that provided in the compiler. (The overhead associated with a process mapping operation is just two PCN procedure calls, rather than tens of procedure calls.)
- Special compiler tools must be used to compile the application program.

11.1 Annotations

The process mapping tools described in this section allow the programmer to specify process mapping in terms of five different abstract views of a parallel computer: array, ring, mesh, torus, or random. Each of these *virtual machines* has associated annotations which specify mapping in terms of its abstract topology.

The name of the virtual machine that is to be applied in a particular program must be specified by means of a directive included in each module of the application. This directive has the general form:

`-directive("virtual_machine",`*type*`)`

where *type* is one of `"array"`, `"ring"`, `"mesh"`, `"torus"`, or `"random"`.

Ring. In the ring virtual machine, the nodes of a parallel computer are viewed as being connected in a ring. Annotations `@fwd` and `@bwd` specify that a procedure should execute on the next or previous node in the ring, relative to the node on which the annotated procedure was invoked. (The use of these annotations has already been illustrated in § 10.)

Array. In the array virtual machine, the nodes of a parallel computer are viewed as being connected in a linear array. Execution starts in the first node in the array. Annotations `@fwd` and `@bwd` specify that a procedure should execute on the next or previous node, relative to the node on which the annotated procedure was invoked. An attempt to map beyond either end of the array is signaled as a run-time error.

Torus. In the torus virtual machine, the nodes of a parallel computer are viewed as being connected in a 2-dimensional torus. Annotations `@north`, `@east`, `@south`, and `@west` specify where a procedure call should execute, relative to the node on which it was invoked.

Mesh. In the mesh virtual machine, the nodes of a parallel computer are viewed as being connected in a 2-dimensional mesh. Execution starts in the southwest corner of the mesh. Annotations `@north`, `@east`, `@south`, and `@west` specify where a procedure call should execute, relative to the node on which it was invoked. An attempt to map beyond any edge of the mesh is signaled as an error.

Random. This virtual machine supports a single annotation: `@random`. As its name suggests, this specifies that a procedure should execute on a randomly-specified node.

11.2 Compiling Programs

In order for the process mapping tools described in this section to be applied to a PCN program, both of the following conditions must be satisfied.

1. Each module in the program must contain the same `virtual_machine` directive (used to specify the name of the virtual machine to be applied). E.g.:

```
-directive("virtual_machine","ring")
```

2. The entry points to the program must be specified in an `entrypoints` directive. E.g.:

```
-directive("entrypoints","go1","go2")
```

3. Any modules that are called from within a module but that are not to be transformed (i.e., do not contain annotated calls) must be named in a `dont_transform` directive. E.g.:

```
-directive("dont_transform","stdio")
```

The entry points of a parallel program are the procedures which will actually be invoked from the PCN shell.

The compilation process then proceeds in two stages. First, each module in the program is compiled with a specialized mapping compiler, invoked with the capability `vm`. (This is loaded by typing `load("vm_co")`). For example, in the following sequence the files `cell.pcn` and `grid.pcn` are compiled.

```
load("vm_co")
loaded vm_co
vm("cell")
VM: file cell, machine mesh
Written:  cell.pam
vm("grid")
VM: file grid, machine mesh
Written:  grid.pam
```

This compilation process applies some specialized transformations to the application modules and then invokes the PCN compiler. In addition to the usual `.pam` and `.mod` files, a temporary `.tem` file is written for each module containing information about the virtual machine used in the module, annotated calls, and exported and imported procedures.

Assuming that each application module compiled without error, we now proceed to the second stage of the compilation process. The various modules of our application are linked by calling the capability `link`. This first checks the various modules for consistency (e.g., it verifies that all use the same virtual machine, and that all modules used by the application have been transformed). Then, it generates a link module that we use to actually invoke our application.

A call to the linker specifies the name of the modules included in the application and the name of the link module. For example:

```
link(["cell","grid"],"gridgo")
Opened file cell.tem
Opened file grid.tem
Consistency check succeeds
Written:  gridgo.pam
```

11.3 Running the Compiled Program

Any procedure named as an entrypoint can be invoked in the link module. An additional first argument must be provided that specifies the size of the machine, as an integer number of nodes in the `ring`, `array`, or `random` machines, or a `{m,n}` tuple in the `torus` or `mesh` machines. For example, the following call invokes the procedure `go1` on a 16×32 mesh.

```
gridgo:go1({16,32})
```

12 Higher-Order Programs

PCN provides simple support for higher-order programming. In particular, it allows module and procedure names in procedure calls to be substituted wth variables, which can then be defined to be strings at run time. Variables are distinguished from strings in procedure calls by the use of enclosing back quotes, as follows.

```
..., `op`(...), ...          /* op is a variable */
..., m:`op`(...), ...        /* op is a variable */
..., `mod`:f(...), ...       /* mod is a variable */
..., `mod`:`op`(...), ...    /* mod & op are variables */
```

We illustrate the use of these higher-order features with a procedure `map_list` that applies a supplied operator to each element of a list, collecting the results of these computations in an output list. The supplied operator is assumed to be a procedure name (e.g., `"f"`); the `map_list` procedure invokes this procedure with two arguments (e.g., `f(e,v)`).

```
map_list(op,list,vals)
{ ? list ?= [e|l1] ->
    {|| `op`(e,v),
        vals = [v|v1],
        map_list(op,l1,v1)
    },
    list ?= [] -> vals = []
}
```

For example, if the procedure square is defined as follows:

```
square(e,v) {|| v = e*e }
```

then a call `map_list("square",[1,2,3],vals)` will define `vals` to be the list `[1,4,9]`.

The `map_list` procedure will only work correctly if the supplied operator (op) is located in the same module as `map_list`. The following program is more general: it allows the supplied operator to be a `mod:proc(arg)` term. Note the use of quoting in the match operation.

```
map_list2(op,list,vals)
{ ? list ?= [e|l1], op ?= `mod`:`proc`(arg) ->
    {|| `mod`:`proc`(arg,e,v),
        vals = [v|v1],
        map_list2(op,l1,v1)
    },
    list ?= [] -> vals = []
}
```

13 Debugging PCN Programs

PCN provides a rich set of facilities for locating syntactic, logical, and performance errors in programs.

Syntax errors are detected and reported by the compiler. An error message consists of a line number and a grammar rule. The line number indicates the location of the error in the program, and the grammar rule indicates the part of the grammar given in Appendix G in which parsing failed.

Warning messages are also generated by the compiler to indicate type mismatches between procedure definitions and calls, etc. It is good programming practice to write programs that do not generate warnings.

Support for detection of *logical errors* is provided by a special debugging version of the PCN run-time system and shell, called `pcn.pdb`. Typing `pcn.pdb` at the Unix prompt invokes a shell identical to that invoked by `pcn`, but providing two additional features:

1. Bounds checking is performed on all array and tuple accesses.

2. Typing `^C` (control-C) causes control to pass to the PCN symbolic debugger, PDB. PDB is described in detail in § 14.

Additional, low-level logical debugging support is provided by command line arguments that cause the PCN run-time system to print detailed information about individual procedure calls. These facilities are described in § 26; their use is not recommended in normal circumstances.

We use the term *performance error* to refer to programs that compute correct answers but for some reason do not make efficient use of available computer resources. Two tools are integrated with the PCN system to assist in the detection of performance errors: Gauge and Upshot. These are described in § 15 and § 16, respectively.

Gauge is an execution profiler: it collects information about the amount of time that each processor spends in different parts of a program. It also collects procedure call counts, message counts, and idle time information. Two properties of Gauge make it particularly useful: profiling information is collected automatically, without any programmer intervention, and the volume of information collected does not increase with execution time. A powerful data exploration tool permits graphical exploration of profile data.

Upshot is a more low-level tool that can provide insights into the fine-grained operation of parallel programs. Upshot requires that the programmer instrument a program with calls to event logging primitives. These events are recorded and written to a file when a program runs. A graphical trace analysis tool then allows the programmer to identify temporal dependencies between events.

Part II
Reference Material

14 PDB: A Symbolic Debugger for PCN

Debuggers play an important role when programming in any language, including PCN. However, PCN is considerably different from most other programming languages (e.g., C, Fortran). For example, PCN uses both light-weight processes and dataflow synchronization extensively. Therefore, a PCN debugger must have special capabilities designed to meet PCN's atypical requirements.

PDB, the PCN debugger, fits this bill. It incorporates features found in most debuggers, such as the ability to interrupt execution and examine program arguments. In addition, it incorporates capabilities that support atypical features of PCN, such as light-weight processes and dataflow synchronization. In particular, PDB allows you to examine enabled and suspended processes and to control the order in which processes are scheduled for execution.

The operation of PDB is complicated by the fact that the PCN run-time system does not support PCN directly, but rather a simpler language called core PCN, which lacks sequential composition and nested blocks. The PDB debugger operates on core PCN rather than PCN; hence, some understanding of the transformations used by the compiler to translate PCN to core PCN is necessary before PDB can be used effectively.

14.1 The PCN to Core PCN Transformation

Nested Blocks. Nested blocks within PCN programs (except for sequential or parallel blocks nested in a top-level choice block) are replaced with calls to separate *auxiliary procedures* that contain these blocks. An auxiliary procedure is given the name of the procedure from which it was extracted, followed by an integer suffix. The choice of integer suffix is somewhat arbitrary. But in general, suffixes are assigned in the order in which the corresponding auxiliary procedure calls appear in the original procedure.

Sequential Composition. Additional auxiliary procedures may be introduced as "wrappers" on operations occurring in sequential compositions. A wrapper delays execution of an operation until previous computations in the sequential composition have completed.

Wrappers are also generated to encode calls to primitive operations for which arguments may not be available at run-time. Such wrappers delay computation until definitional arguments are defined. For example, a wrapper for the assignment `x:=y`, where y is a definition, will delay execution until y has a value.

Wrappers are named in the same manner as other auxiliary procedures: with a procedure name followed by a number.

Sequencing Variables. Every procedure has two additional variables added to its argument list. These variables are used for sequencing of procedure calls. They are commonly referred to as the *Left* and *Right* sequencing variables. A procedure will suspend until its *Left* variable is defined. When the procedure and its offspring have completed execution, *Right* is defined to be the same as *Left*. These variables often (but not always) occur at the end of the argument list.

Within a sequential block, the *Right* variable of one procedure call is the same as the *Left* variable of the next. This ensures that procedures execute in strict sequence. For example, the sequential block

Example of a sequential block

```
p()
{ ; q(),
    r(),
    s()
}
```

is transformed to a procedure similar to the following.

Example of a transformed sequential block

```
p(L,R)
data(L) ->
    {|| q(L,M1),
        r(M1,M2),
        s(M2,R)
    }
```

Within a parallel block, all procedure calls use the parent procedure's *Left* variable as their own *Left*, and a temporary variable as their *Right*. The temporary *Right* variables are passed to a barrier procedure which defines the *Right* variable for the parallel block when all of the temporary variable have been defined. For example, the parallel block

Example of a parallel block

```
p()
{|| q(),
    r(),
    s()
}
```

is transformed to a procedure similar to the following (where `p.1` is the barrier procedure)

Example of a transformed parallel block

```
p(L,R)
data(L) ->
    {|| q(L,M1),
        r(L,M2),
        s(L,M3),
        p.1(M1,M2,M3,R)
    }

p.1(M1,M2,M3,R)
data(M1), data(M2) -> R = M3
```

Barrier Processes. As demonstrated in the previous example, the PCN compiler sometimes generates calls to special *barrier* procedures. These are used to organize synchronization of procedures in a parallel block. These auxiliary programs are named in the same manner as other auxiliary procedures created by the compiler. However, they can usually be distinguished by the fact that all but one of their arguments are the *Right* synchronization variables of other procedures. Fortunately, these auxiliary barrier procedures can generally be ignored when debugging.

Wildcards. A procedure name is a `mod:procedure` pair. Some PDB commands that take procedure names as arguments allow the use of a limited form of a wildcard facility to specify a set of procedures. An asterisk (*) placed at the end of a procedure name designates all procedures that begin with the specified name. For example, `mod:program1*` designates all procedures in module `mod` whose names begin with `program1`. The degenerative case of a procedure wildcard is simply a *. (E.g., `mod:*`.) In this case, all procedures within the appropriate module are specified.

Module names can also be specified with this limited wildcard facility. For example, a module wildcard of `env*` designates all modules whose names start with `env`, and a simple * designates all loaded modules.

14.2 Naming Processes

Execution of a PCN program can create a large number of lightweight processes. Each process executes a PCN procedure — either a procedure named in the original source, or an auxiliary procedure introduced by the transformation to core PCN.

In order to simplify the task of distinguishing between the many processes that may be created during execution of a program, PDB associates three distinct labels with each process.

1. The name of the procedure that the process is executing. (Nonunique)
2. An instance number. (Unique)

3. The process reduction in which the process was created. (Nonunique)

As we shall see in §14.5, PDB also provides information about the status of a process, for example, whether it is able to execute or is waiting for data.

14.3 Using the Debugger

pcn.pdb and pcncc.pdb. We assume familiarity with the PCN development environment. Recall that this is invoked by the command pcn and that it provides a simple command interpreter and other facilities. In applications that include foreign code, the linker pcncc is used to link the foreign code into the environment. PDB is supported by an extended version of this environment called `pcn.pdb` and an extended linker, `pcncc.pdb`.

Recall that the PCN command interpreter allows you to invoke *commands* (with the form `mod:prog(...)`) interactively. These commands may execute both PCN code contained in a number of *modules* and foreign code contained in other files.

PDB allows debugging to be enabled and disabled on a per-module basis. Only processes executing procedures contained in enabled modules are visible within the debugger.

Control-C. You enter PDB during program execution by interrupting the program with an interrupt signal. This signal is typically invoked by typing Control-C (^C).

Once control as passed to the debugger, PDB commands can then be used to enable/disable debugging on modules, examine the state of the computation, or resume execution of the PCN program. Once resumed, normal PCN execution continues until you interrupt the program execution again, causing control to revert back to the debugger. It is also possible to specify that control pass to the debugger if the active queue becomes empty. This is accomplished by setting the debugger variable `empty_queue_break` (§ 14.6).

Abbreviating PDB Commands. PDB commands can be abbreviated to the shortest string that uniquely identifies the command. (There are a few exceptions to this rule. For example, since the `show` command is typically used extensively, it can be abbreviated to `s`, even though `s` does not uniquely identify this command.)

To find out the shortest abbreviation for PDB commands, use the PDB help facility by typing `help` at the PDB command prompt.

.pdbrc When PDB starts up, it searches for a `.pdbrc` file in the current directory, and then in your home directory (~). Any PDB commands found in such a file are executed. This feature allows the state of PDB to be initialized every time PDB is run.

14.4 Obtaining Transformed Code

As described in § 14.1, a PCN program is transformed to core PCN before execution. When debugging programs with PDB it is sometimes helpful to have this transformed version of the code available for reference, since that transformed version is really the code that is being executed.

Assuming you have the PCN source file `prog.pcn`, you can create a transformer output file in two ways. One way is to run the following command from the Unix shell.

```
pcncomp -E prog.pcn
```

Alternatively, you can run the following command from the PCN shell, after loading the `co` module.

```
transform("prog")
```

In both cases, a file named `prog_tfd.pcn` will be created that contains a nicely formatted representation of the core PCN corresponding to the original PCN program, `prog.pcn`.

14.5 Examining the State of a Computation

We are now ready to describe PDB commands. We first describe commands that allow us to examine the state of a PCN computation. For you to understand how these commands work, we need to say a little bit about how the PCN run-time system manages execution of PCN programs.

Queues. The PCN run-time system manages the execution of processes created to execute procedure calls in parallel blocks. Like a simple computer operating system, it selects processes from an *active queue* and executes them either until either they block because of a read operation on an undefined definitional variable or until a timeslice is exceeded. In the former case, the process is moved to a *variable suspension queue* associated with the undefined definitional variable (unless the process requires two or more variables, in which case it is moved to a *global suspension queue*). In the latter case (a timeslice), the process is moved to the end of the active queue. PDB also maintains a fourth *pending queue*. This is used to hold processes from the active queue that the user has indicated are to be delayed (i.e., removed from consideration by the PCN scheduler).

In summary, every PCN process is to be found on one of the following four queues:

`active` The *active queue* contains processes that may be scheduled for execution.

`pending` The *pending queue* contains processes that the user has tagged to be delayed. These cannot be executed until returned to the active queue.

`globsusp` The *global suspension queue* contains processes that are suspended on more than one variable.

`varsusp` The *variable suspension queue* contains processes that are suspended on just one variable.

When describing commands, we shall use the notation <queue> to represent a queue selector— one of `active`, `pending`, `globsusp`, and `varsusp`; or `suspension` (both `globsusp` and `varsusp`) and `all` (all process queues).

We shall also use the notation <process> to represent a process specification; this is one of the following:

- *n*: *n* is an integer, representing an index into a process queue;
- *m* – *n*: *m* and *n* are integers, representing a range of indices into a process queue;
- #*n*: *n* is an integer, representing a process instance number;
- ^*n*: *n* is an integer, representing the reduction during which a process was created;
- _U*h*: *h* is a hexadecimal number, representing an undefined variable that is somewhere in a process's argument list;
- *modulename:blockname*, representing all processes of a given name;
- `all`.

As noted in § 14.1, a limited wildcard facility allows a single <process> specifier to represent several processes.

Examining Queue Contents. The `summary`, `list`, and `show` commands allow the user to examine the four process queues at increasing levels of detail. These commands (and the queue-manipulation commands described in the next section) operate only on processes contained in modules for which debugging is enabled. The set of enabled modules is initially the empty set (i.e., no modules are enabled); the set can be modified by using the **debug** and **nodebug** PDB commands.

In the following descriptions, all arguments that are listed within square brackets ([ ]) are optional:

`summary` [<queue>] [<process>]: prints a summary of the contents of the designated <process> on the designated <queue>. This includes module and procedure names (sorted by module and then procedure) and the number of occurrences of each procedure on each queue.

`list` [<queue>] [<process>]: prints a short listing of the processes specified by <process> on the specified <queue>.

`show` [<queue>] [<process>]: prints a detailed description of the processes specified by <process> in the specified <queue>. If the process is on the variable suspension queue, the variable that it requires in order to continue execution is also shown.

Modifying Queues. The **move** and **switch** commands are used to control how processes in the active queue are selected for execution. They can be applied only to the active and pending queues.

move <queue> <process> [<where>]: This moves zero or more designated processes in a designated queue (**active** or **pending**) to immediately before position *where* in the same queue. If *where* is **end**, then the designated processes are moved to the end of the queue. By default, <where> is **end**.

switch <queue> <process> [<where>]: This moves zero or more designated processes from a designated queue (**active** or **pending**) to the other queue (i.e., **pending** or **active**, respectively), inserting them immediately before position *where*. If *where* is **end**, the designated processes are placed at the end of the queue. By default, <where> is **end**.

14.6 Debugger Variables

PDB maintains a number of internal variables that can be included in PDB commands and, in some cases, modified by the programmer. PDB variables are distinguished in expressions by a prefix $.

Modifiable Variables. The following variables can be used to control aspects of PDB's behavior. They can be modified within PDB by using the "=" command.

$print_array_size : An integer representing the maximum size (i.e., number of elements) of an array displayed by **print**.

$print_tuple_depth : An integer representing the maximum depth of a tuple displayed by **print**.

$print_tuple_width : An integer representing the maximum width (i.e., number of elements) of a tuple displayed by **print**.

$emulator_dl : An integer representing the emulator debug level. This turns on the printing of debugging information in the main emulator loop. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging.

$gc_dl : An integer representing the garbage collection debug level. This turns on the printing of debugging information in the garbage collector. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging.

$parallel_dl : An integer representing the parallel code debug level. This turns on the printing of debugging information in the parallel emulator code. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging.

`$global_dl` : An integer representing the global debug level. This turns on the printing of debugging information not covered by the `$emulator_dl`, `$gc_dl`, or `$parallel_dl` debug setting. It takes an integer value between 0 and 9, where 0 is no debugging and 9 is the most debugging.

`$reduction_break` : An integer representing the next reduction at which to break into PDB.

`$empty_queue_break` : A Boolean value. When this value is set to **yes**, the system will break into PDB whenever the process queues are empty, and therefore there are no schedulable processes. When this value is set to **no**, the system will not break into PDB whenever the process queues are empty.

Read-Only Variables. The following variables contain information about various aspects of the state of the computation. They can be included in expressions but cannot be modified directly.

`$modules` : A list naming all of the modules that are being debugged. This list can be modified through the **debug** and **nodebug** commands.

`$module` : The name of the current module.

`$process` : The name of the current process.

`$args` : The arguments of the current process. Note that this variable is defined only at the entry to a block.

`$instance` : The instance number of the current process.

`$reduction` : The reduction during which the current process was created.

`$current_reduction` : The current reduction number.

14.7 Miscellaneous Commands

This section describes miscellaneous debugger commands that were not described in other parts of this manual.

In the following, <expr> denotes either a PCN variable name (to be interpreted in the context of the current process) or a constant.

abort : Abort execution of the PCN run-time system. See also **continue**, **next**, and **quit**.

continue : Continue with next process (head of the active queue). See also **abort**, **next** and **quit**.

debug <*module*> : Enable debugging in the specified *module*. The <*module*> argument can be a module wildcard. See also **nodebug**.

help [<*topic*>]: Give help for *topic*. If *topic* is left off, then general help will be given.

load <*filename*> : Execute the PDB commands that are in the file *filename*.

modules : List the names of the modules that are currently loaded in the system, indicating for each whether it was compiled in debug mode (in the current PCN release, this column always says "n") and whether debugging is enabled.

next : Execute the next process (head of the active queue), and then break into the debugger again when it has completed. See also `abort`, `continue`, and `quit`.

nodebug <*module*> : Disable debugging in the specified *module*. The <*module*> argument can be a module wildcard. See also `debug`.

print <expr>: Print the given expression. An expression is a variable, integer, real, or string. <expr> is either a single expression or a comma separated list of expressions that is enclosed in parentheses.

quit : Quit from the debugger; disable debugging on all modules. See also `abort`, `continue`, and `next`.

vars : List the names and values of all PDB variables.

14.8 Orphan Processes

An *orphan* is a process suspended on a variable for which there are no potential producers. (More precisely, a variable to which no other process possesses a reference.) Such a process can never be scheduled for execution. A program that generates orphan processes is not necessarily erroneous. However, it is good programming practice to ensure that orphans are not generated, i.e. that all processes in a program terminate.

Orphan processes can be detected by the garbage collector invoked by the PCN run-time system to reclaim space occupied by inaccessible data structures. Normally, the garbage collector destroys these processes silently. However, the debugging version of the PCN system (`pcn.pdb`) prints a warning message for each orphan encountered.

15 The Gauge Execution Profiler

Gauge is an execution profiler for PCN programs. It collects profile data which can subsequently be graphically displayed by using an interactive data exploration tool.

15.1 Data Collection

Before Gauge can be used, the Gauge profiler must be loaded by issuing a

```
load("gauge")
```

command to the PCN shell. This command can be put into your `.pcnrc` file (§ 4.4). No special compilation is required to profile a program.

A profile is generated by executing the program from a `profile` command (a capability provided by the `gauge` module) rather than the usual `module:goal` command. This command has the following general form.

```
profile(program,modulelist,nodes,filename,done)
```

For example, the command

```
profile(foo:bar(1,2),["foo","foo1"],"all","footest25",done)
```

will run the program `bar` in module `foo` and collect profile data for modules `foo` and `foo1` from every processor (`all`). The profile is stored in a file called `footest25.cnt` when execution of `bar` completes.

The following, alternative forms of the `profile` command are also recognized:

```
profile(program,modulelist,nodes,filename)
profile(program,modulelist,done)
profile(program,modulelist)
```

The arguments to the `profile` command are as follows:

`program` Specifies the program to execute and its arguments. The program is specified exactly as if it was being executed directly from the shell.

`modulelist` A list of strings that are the names of the modules from which profile data is to be collected. If only one module is needed, a single string can be specified instead of a one-element list.

`nodes` Specifies the nodes from which profile data is collected. This argument can be a single node number (i.e., 2), a range of node numbers (i.e., 2-20), or the string `all`. In the form of the profile command without a node argument, profile data is collected from all processors.

`filename` File in which to place the profile data. Profile files have a ".cnt" extension; this is automatically added to the file specified by the filename argument. In the form of the `profile` command without a filename argument, the profile data is stored in a file named `profile.cnt`.

`done` A definitional variable that is defined to be the empty list ($\square$) when the profile is complete.

Note that the `profile` command collects profile data only after the application program has completely terminated. In some cases, it may be preferable for the application program to signal when a profile should be generated, by binding a definitional variable. Commands that provide this capability are listed in § 18.2, as are commands for timing execution of PCN programs.

15.2 Data Exploration

Gauge provides a graphical interactive tool for exploring profile data collected using the `profile` command. This tool combines three sorts of data to provide detailed information about execution time on a per-procedure and per-processor basis, idle time, number of messages, volume of messages and other program execution statistics. These data are

- instruction counts collected by the compiler (the `.mod` files),
- profile data collected by the run-time system (the `.cnt` file), and
- information about the computer on which the program was run (the `host` file).

The first of these data is generated by the compiler, the second is generated automatically if the `profile` command is used, and the third may need to be specified by the user (see §15.3 below).

The data exploration tool is invoked by typing the following Unix command:

```
xpcn
```

This creates a top-level window with three parts. The top section of the window is a command window. You can click the left mouse button on one of the commands to obtain *help*, to `exit`, or to invoke `gauge`.

The middle section indicates the current Xpcn directory.

The bottom window gives a list of `.cnt` files and directories in the current directory. Files are selected by pressing the left mouse button while the pointer is over the file name. If you wish to change selections, just press the left mouse button over a different file, or no file if you want to eliminate all selections.

The directory window serves two purposes. If you select a `.cnt` file in the directory window using the left mouse button and then select the *Gauge* command from the top row of buttons, Gauge is invoked on that file.

Gauge can also be invoked on a `.cnt` file by double-clicking on its name in the directory window. Double-clicking on a directory name opens that directory, thus allowing navigation of the directory system.

Xpcn has an online help facility. To use it, select the "help" button on any window. Either the scroll bar or the page-down (Control-v) and page-up (Meta-v) commands can be used to position the help text within the help window. When finished, you can dismiss the help screen using the *close* button on the bottom of the screen. If you leave the screen up, it will be reused to display the next help message.

Occasionally something might go wrong, and xpcn will generate a Warning Message in a popup window. Nothing else can be done until this window is dismissed by clicking the left mouse button in it.

The only command-line arguments recognized by xpcn are those recognized by the X Toolkit Intrinsics. This means that X-window arguments such as `-icon` can be used.

15.3 The Host Database

When you invoke Gauge on a `.cnt` file, a warning message may be displayed indicating that your machine does not appear in the host database. (Click on the warning window to make it disappear.) This means that you must add the machine on which your application was run to the *host database* that Gauge accesses to determine various machine characteristics when displaying performance data.

The program `pcnhost` is provided to simplify the task of adding entries to the host database. A call to this program has the form

```
pcnhost machinetype
```

or

```
pcnhost -h hostname machinetype
```

The `machinetype` argument specifies an architecture type for machine computer `hostname`. If a host name is not specified, the name of the machine on which the pcnhost command is executed is added to the database. The following machine types are currently supported:

- `symmetry-b`, `symmetry`: Sequent Symmetry Rev B
- `sparcstation-1`, `ss1`: A Sun SPARCstation 1
- `sun3`: A Sun 3 workstation
- `s2010`: A Symult s2010 multicomputer
- `rs6000`: An IBM RS6000 workstation.
- `ipsc860`: An Intel iPSC/860 (i860 processing nodes)
- `ipscii`: An Intel iPSC/II (386 nodes)

Note that updates to the database are not synchronized. If more than one update is being made simultaneously, information can be lost.

15.4 X Resources

Xpcn requires a resource file to operate properly. This should be in

```
$(INSTALL_DIR)/lib/app-defaults/Xpcn
```

where `$(INSTALL_DIR)` is the directory where xpcn has been installed (typically, `/usr/local/pcn`). The line

```
xrdb -merge $(INSTALL_DIR)/lib/app-defaults/Xpcn
```

should be added to one's `.xinitrc` or `.xsession` file. If a color workstation is being used,

```
xrdb -merge $(INSTALL_DIR)/lib/app-defaults/Xpcn.server
```

is also needed. Of course, any customized resource files could be used.

In addition, the Unix shell variable XFILESEARCHPATH must be set to tell the X Window system where to find some files. For example:

```
setenv XFILESEARCHPATH = '/usr/local/pcn/lib/%L/%T/%N%S:\
       /usr/local/pcn/lib/%l/%T/%N%S:/usr/local/pcn/lib/%T/%N%S:\
       /usr/lib/X11/%L/%T/%N%S:/usr/lib/X11/%l/%T/%N%S:\
       /usr/lib/X11/%T/%N%S'
```

16 The Upshot Trace Analyzer

Upshot is a trace collection and analysis tool. You need to know the following three procedures in order to use Upshot in PCN.

1. How to instrument a program.
2. How to run an instrumented program and collect a log.
3. How to analyze a log.

16.1 Instrumenting a Program

You instrument your program by adding calls to

- the foreign procedure `pcn_log_event` in PCN procedure(s), and/or
- calls to `c_log_event` in C procedures, and/or
- calls to `log_event` in Fortran procedures.

`pcn_log_event(event_type,return_code)` This C procedure is used to log events from within PCN procedures.

`event_type` is an input argument and must be a positive integer.

`return_code` is an output argument (it must be a mutable) and is set to 0 if the event is logged successfully and to 1 otherwise.

`int c_log_event(event_type)` This C function is used to log events from within C procedures. The function returns 0 if the event is logged successfully and 1 otherwise.

`event_type` is an input argument and must be a positive integer.

`int log_event(event_type)` This C function is used to log events from within Fortran procedures. The function returns 0 if the event is logged successfully and 1 otherwise.

`event_type` is an input argument and must be a positive integer.

16.2 Collecting a Log

To collect a log, we must first load the upshot module into the PCN shell by typing `load("upshot")`. Then, we run our program, using one of the following forms.

```
log(mod:call(...),size,nodes,file)
log(mod:call(...),size,nodes,file,done)
log(mod:call(...))
log(mod:call(...),size,done)
```

Where `size` is the maximum number of events that can be logged in a single node (default 10,000), `nodes` is a node specification, as in Gauge (default `"all"`), `file` is the prefix for the log files (default `"upshot.log"`); and `done` is a variable to be bound when logging has completed.

When you run your program, you will see (for example)

```
log(mod:foo())
Initialized logs:  size 10000
<program output>
Written logs to upshot.log
```

If a program that includes log statements is executed without a `log` wrapper, a warning will be printed once on each node in which logging is attempted:

```
Upshot:  Not initialized
```

A warning is also displayed if the maximum number of events is exceeded on any node. For example, if the event buffer size is set to 40, and more than 40 events are logged, you will receive the warning

```
Upshot:  Event buffer overflow (40 events)
```

16.3 Analyzing a Log

Execution of a program using `log` produces one log file for each node; these files are called `upshot.log.0`, `upshot.log.1`, etc. (An alternative prefix can be specified in the `log` command.)

These files must be merged by using the Unix command **mergelogs** to create a single log file, for example,

```
mergelogs upshot.log.* > log
```

We can then call Upshot to display a set of time lines, one per processor, with the various events logged by our program displayed on the appropriate time lines:

```
upshot -l log
```

Frequently, we are not interested in the events themselves but rather in execution states defined in terms of starting events and ending events. (E.g., we might define a "busy" state as starting when we an event is logged indicating that a message has been received on a stream, and ending when an event is logged indicating that a response has been sent.) We define states in a states file, specifying each state in terms of a unique integer identifier, a starting and an ending event type, a color, and a label. For example:

File `my.sts`

```
1  10  11  blue    init_ico
2  12  13  red     init_rh
3  14  15  pink    init_geo
4  16  17  yellow  get_side
```

Upshot does not support nested states. That is, it is not meaningful for a trace to include sequences in which two start state events occur without an intervening end state event.

The name of any state file is specified to Upshot by means of the `-s` command line option, as follows.

```
upshot -l log -s my.sts
```

17 Standard Libraries

The `sys` and `stdio` modules are distributed with the PCN system and may be called from within user programs to invoke a variety of useful functions. They are invoked via intermodule calls.

In the following discussion, the notations ↓ and ↑ on program arguments denote input and output arguments, respectively.

17.1 System Utilities

The `sys` module provides the following programs. Many of these are provided for compatibility with the Strand parallel programming system.

`merger(Is↓,Os↑)` merges messages appearing on input stream `Is` to produce output stream `Os`. If the input stream `Is` contains a message of the form `merge(S)`, then the stream `S` is also merged with `Os`. The output stream `Os` is closed when all merged input streams are closed. (Cf. § 5.10 for more details.)

`distribute(N↓,Is↓)` distributes messages received on input stream `Is` to `N` output streams; output streams are numbered `0` to `N-1`. (Cf. § 5.10 for more details.) The distributor may receive two types of message on input stream Is:

attach(N1↓,S↓,D↑) causes stream S to be attached to output stream numbered N1; D is defined when the action is complete to signify that messages may subsequently be forwarded to stream S.

{N2↓,M↓} causes the message M to be appended to output stream numbered N2.

When the input stream Is is closed, all output streams are closed.

hash(N↓,Is↓) creates a hash table of size N and receives messages on input stream Is. Four messages may be sent to a hash table:

add(K↓,V↓,S↑) causes the value V to be added to the hash table under key K; if there was already an entry for key K, then status S=0, otherwise S=1.

lookup(K↓,V↑,S↑) causes a lookup operation on key K. If there is an entry for key K, then V is the associated value and status S=1, otherwise S=0.

del(K↓,V↑) deletes the entry for key K and returns the value V associated with the entry if one existed, otherwise returns -1.

dump(L↑,D↑) dumps the contents of the hash table into a list L and defines D when the operation is complete.

integer_to_list(I↓,Lb↑,Le) difference list Lb-Le contains integers providing an ASCII representation of integer I

integer_to_string(I↓,S↑) S is a string that represents integer I

real_to_list(R↓,L↑) difference list L contains integers providing an ASCII representation of real number R

real_to_string(R↓,S↑) S is a string that represents real number R

string_to_real(S↓,R↑) R is the real number represented by string S

string_to_list(S↓,Lb↑,Le) difference list Lb-Le contains integers providing an ASCII representation of string S

list_to_integer(L↓,I↑) I is the integer represented by the ASCII values in list L

list_to_string(L↓,S↑) S is the string represented by the ASCII values in list L

list_to_real(L↓,R↑) R is the real number represented by the ASCII values in list L

list_to_tuple(L↓,T↑) T is the tuple with elements specified by list L

tuple_to_list(T↓,Lb↑,Le) difference list Lb-Le contains the arguments of tuple T

logand(A↓,B↓,C↑) integer C is the bitwise-and of integers A and B

logor(A↓,B↓,C↑) integer C is the bitwise-or of integers A and B

`logcomp(A↓,B↑)` B is the bitwise complement of integer A

`logreal(A↓,B↑)` B is the real version of A

`logint(A↓,B↑)` B is the integer version of A

`logabs(A↓,B↑)` B is the absolute value of A

`str_len(S↓,L↑)` L is the length of string S

`arity(T↓,A↑)` A is the number of arguments in tuple T

17.2 Standard I/O

The stdio module provides a set of PCN procedures that are analogous to the C language standard input/output (stdio) library. It is important to realize that calls to stdio are sequenced only if they occur within a sequential block. Output generated by *parallel* calls to printf or other output procedures may be interleaved.

Most of the stdio procedures takes an output argument, status. This argument should be an undefined variable when the call is made. It will be defined by the stdio procedure to an appropriate return code. This argument can be used both to check the status of the I/O call and to sequence subsequent execution.

The stdio procedures that deal with files rather than the keyboard or screen require a file descriptor (fd) argument. This argument should be a mutable of type FILE (defined in the C header file stdio.h).

We now summarize the procedures provided by the stdio module. The arguments to all of these procedures follow as closely as possible their corresponding C procedures. Please refer to a C programming manual for more complete descriptions.

`fopen(filename↓,type↓,fd↑,status↑)` opens the file named filename. The file is opened for the given type of I/O operation, where type is a string containing an appropriate combination of "r", "w", "a'', and "+". The mutable fd is assigned to be the file pointer. status is defined to be 0 if the open succeeds; otherwise it will be set to the error number (C errno).

`fdopen(fildes↓,type↓,fd↑,status↑)` opens the file with the integer file descriptor fildes. The other arguments are the same as for fopen().

`fclose(fd↓,status↑)` closes the file designated by fd. status is defined to be EOF if there is an error.

`fflush(fd↓,status↑)` flushes all buffered data for the output file designated by fd to be written to that file. The file remains open. status is defined to be EOF if there is an error.

`putc(c↓,fd↓,status↑)` appends the character c to the designated output stream fd. status is defined to be the character written, or EOF if there is an error.

`fputc(c↓,fd↓,status↑)` is the same as putc().

putchar(c↓,status↑) is the same as putc() to standard output (the screen).

putw(w↓,fd↓,status↑) appends the word character w (an integer rather than a character) to the designated output stream fd. status is defined to be the word written, or EOF if there is an error.

printf(format↓,args↓,status↑) prints formatted output to standard output. The format string accepts the same format as the C language's printf() procedure, with two additions: it can contain a %t, which means to print a grounded term, and %lt, which means to print an ungrounded term. The %t and %lt can also take an integer immediately after the %, which means to only print to that depth. The args argument is a tuple of all the arguments to printf, as required by the format. (Since PCN procedures cannot take a variable number of arguments, as in C, all of the data arguments must be combined into a single argument using a PCN tuple.) status is defined to be the number of characters written, or EOF if there is an error.

fprintf(fd↓,format↓,args↓,status↑) is the same as printf(), except that output will go to fd rather than to standard output.

sprintf(buf↑,format↓,args↓,status↑) is the same as printf(), except that the output is placed into the definitional variable buf.

getc(fd↓,c↑) gets one character from the input stream fd and defines it to c. c is defined to be EOF on end of file or an error.

fgetc(fd↓,c↑) is the same as getc().

getchar(c↑) is the same as getc() from standard input (the keyboard).

getw(fd↓,w↑) gets one word (e.g., an integer) from the input stream fd and defines it to w. w is defined to be EOF on end of file or an error.

ungetc(c↓,fd↓,status↑) pushes the character c back onto the input stream fd. status is defined to be the pushed character, or EOF if there is an error.

scanf(format↓,args↑,status↑) is similar to the scanf() procedure in C. It takes its input from standard input and places the values that it reads in the definitional variables contained in the tuple args.

fscanf(fd↓,format↓,args↑,status↑) is the same as scanf(), except that the input comes from the passed stream, fd.

sscanf(buf↓,format↓,args↑,status↑) is the same as scanf(), except that the input comes from the passed buffer, buf.

stdout(fd↑) assigns the mutable fd to be the file pointer for standard output (stdout).

stdin(fd↑) assigns the mutable fd to be the file pointer for standard input (stdin).

`stderr(fd↑)` assigns the mutable `fd` to be the file pointer for standard error (`stderr`).

`fseek(fd↓,offset↓,whence↓,status↑)` calls the C qfseek function with the `fd`, `offset`, and `whence` arguments to set the position for the next input or output operation on this file. The `status` argument is defined to be 0 if the operation completes successfully, or -1 if it fails.

`ftell(fd↓,offset↑)` calls the C ftell function with the `fd` argument. The `offset` argument is defined to be the offset from the beginning of the file to the current position, or -1 if there is an error.

`rewind(fd↓)` calls the C rewind function with the `fd` argument to set the position to the beginning for the next input or output operation on this file. This is equivalent to `fseek(fd,0,0,_)`.

17.3 Examples of Use

Opening and Closing Files. The following examples illustrates the use of the `fopen`, `fclose`, `stderr`, and `fprintf` procedures. Note the include statement for `stdio.h`, which includes a definition for `FILE`.

```
#include <stdio.h>

open_test(fname)
FILE fd, err;
{ ; stdio:fopen(fname, "r", fd, status),
    { ? status == 0 ->
        { ; stdio:printf("File \"%s\" opened\n",{fname},_),
            /* ... */
            stdio:fclose(fd,_)
        },
        default ->
        { ; stdio:stderr(err),
            stdio:fprintf(err,
               "Error opening \"%s\" for reading\n",{fname},_)
        }
    }
}
```

Writing to a File. This example opens a file `ptest` for writing, writes the characters `ABC` to this file, and then closes the file.

```
#include <stdio.h>

putc_test()
FILE fd;
{ ; stdio:fopen("ptest","w",fd,_),
    /* Use ASCII decimal for character literals */
    stdio:putc(65,fd,_),      /* 'A' */
    stdio:putc(66,fd,_),      /* 'B' */
    stdio:putc(67,fd,_),      /* 'C' */
    stdio:fclose(fd,_)
}
```

Writing to the Screen. This example writes the characters ABC followed by a newline character to the screen (standard input).

```
#include <stdio.h>

putchar_test()
{ ; stdio:putchar(65,_),      /* 'A' */
    stdio:putchar(66,_),      /* 'B' */
    stdio:putchar(67,_),      /* 'C' */
    stdio:putchar(10,_)       /* '\n' */
}
```

Printing to the Screen. The following program uses the printf command to print a variety of terms of the screen. Note the use of the %t format command to print arbitrary terms. When executed, the program acts as follows.

```
> p_test:printf_test()
Str:  A string
Real:  -1.230000
List:  ["A string",-1.230000,{"a",1,2,3}]
Tup:  {"a",1,2,3}
```

The program can be modified to write the same text to a file by adding an fopen call, substituting fprintf for printf throughout, and finally closing the file.

Module p_test.pcn

```
#include <stdio.h>

printf_test()
{ ; str = "A string",
    r = 0 - 1.23,      /* No unary minus in PCN */
    tup = a(1,2,3),
    ls = [str,r,tup],
    stdio:printf("Str: %s\nReal: %f\n",{str,r},_),
    stdio:printf("List: %t\nTup: %t\n",{ls,tup},_)
}
```

Creating Strings. We illustrate the use of the `sprintf` command to create a string. When executed, the `sprintf_test` procedure prints the string `file_5`.

```
#include <stdio.h>

sprintf_test()
{ ; i = 5,
    stdio:sprintf(mystring,"file_%d",{i},_),
    stdio:printf("mystring = %s\n",{mystring},_)
}
```

Reading Characters. This example shows the use of the `stdin` and `getc` procedures to read a series of characters from the keyboard (standard input). The procedure `getc_test` prints a prompt, reads characters until an end of line is reached, and then prints the result.

```
> r_test:getc_test()
Enter line:  my line

Line entered:  my line
```

The program can also be written using the `getchar` procedure (which reads directly from standard input), avoiding the need for the call to `stdin`.

Module r_test.pcn

```
#include <stdio.h>

getc_test()
FILE fd;
{ ; stdio:stdin(fd),
    stdio:printf("Enter line: ",{},_),
    getc_test1(fd,ls),
    sys:list_to_string(ls,str),
    stdio:printf("\nLine entered: %s\n",{str},_)
}

getc_test1(fd,ls)
FILE fd;
{ ; stdio:getc(fd,ch),
    { ? ch == 10 -> ls = [],
        default ->
        { ; ls = [ch|ls1],
            getc_test1(fd,ls1)
        }
    }
}
```

Reading Terms. The following program uses the scanf procedure to read arbitary terms from the keyboard (standard input).

```
#include <stdio.h>

scanf_test()
{ ; stdio:printf("Enter term (or [] to stop): ",{},_),
    stdio:scanf("%t",{Ls},status),
    { ? status != [], status != EOF ->
        { ; stdio:printf("Term: %t\n",{Ls},_),
            scanf_test()
        },
        default -> stdio:printf("Done\n",{},_)
    }
}
```

Reading Terms from a File. The following program uses `fscanf` to read terms from a file and print them on the screen, associating a term with each term. For

example, if a file input contains

```
["a", "b", {-3, -4}]
{-2.5, 0xa, 1e4, 1.5e-3, 010},
```

then execution of the program proceeds as follows. Note how the list structure in the first term is printed as a nested tuple.

```
> f_test:fscanf_test("inp")
(1) '{"a",{"b",{{-3,-4},[]}}}'
(2) '{-2.500000,10,10000.000000,0.001500,8}'
```

Module f_test.pcn

```
#include <stdio.h>

fscanf_test(input)
FILE fd;
{ ; stdio:fopen(input,"r",fd,_),
    show(fd,1),
    stdio:fclose(fd,_)
}

show(fd,index)
FILE fd;
{ ; stdio:fscanf(fd,"%t\n",{term},stat),
    stat != EOF, stat != 0 ->
        { ; stdio:printf("(%d) '%lt'\n",{index,term},_),
            show(fd,index+1)
        }
}
```

18 Standard Capabilities

PCN shell capabilities are described in § 4.3. This section contains a summary of the standard capabilities that are distributed with the PCN system.

In the following discussion, the notations ↓ and ↑ on program arguments denote input and output arguments, respectively.

18.1 co

The compiler capabilities are loaded from the PCN shell by running `load("co")`. These capabilities support full or partial compiles of PCN and PTN programs, the

application of PTN transformations to PCN programs, and the running of lint, the PCN program checker.

`compile(file↓)` Invoke the lint program checker and the PCN compiler on the source `file`.

`compile(file↓,L↓,R↑)` Check and compile the PCN source `file`. Wait for `L` to be defined before running the compile, and define `R` to be the same as `L` when the compile has completed.

`compile(file↓,CPP_flags↓)` Check and compile the PCN source `file`. Pass the `CPP_flags` string to the C preprocessor.

`compile(file↓,CPP_flags↓,L↓,R↑)` Check and compile the PCN source `file`. Pass the `CPP_flags` string to the C preprocessor. Wait for `L` to be defined before running the compile, and define `R` to be the same as `L` when the compile has completed.

`fcompile(file↓)` Fast compile the PCN source `file`. That is, don't invoke the lint program checker.

`fcompile(file↓,L↓,R↑)` Fast compile the PCN source `file`. Wait for `L` to be defined before running the compile, and define `R` to be the same as `L` when the compile has completed.

`fcompile(file↓,CPP_flags↓)` Fast compile the PCN source `file`. Pass the `CPP_flags` string to the C preprocessor.

`fcompile(file↓,CPP_flags↓,L↓,R↑)` Fast compile the PCN source `file`. Pass the `CPP_flags` string to the C preprocessor. Wait for `L` to be defined before running the compile, and define `R` to be the same as `L` when the compile has completed.

`transform(file↓)` Transform the PCN source `file`. The transformed version of the program is written to the file *file*`_tfd.pcn`.

`tcomp(file↓)` Compile the PTN source `file`.

`tcomp(file↓,tracelevel↓)` Compile the PTN source `file`. If `tracelevel` is "`pcn`" or `pcn(`*level*`)`, then only the transformation stage is done, and the transformed version of the program is written to the file *file*`_tfd.pcn`. If `tracelevel` is "`none`" then all output is suppressed. Otherwise, `tracelevel` (or *level*) should be an integer which indicates the debugging trace level to be used during the compile.

`tcomp(file↓,tracelevel↓,L↓,R)` Compile the PTN source `file`. The `tracelevel` is the same as with `tcomp(file,tracelevel)`. Wait for `L` to be defined before running the compile, and define `R` to be the same as `L` when the compile has completed.

`form(file↓,operator↓)` Apply the PTN operator to the PCN source `file`. `file` may also be a list of files to which the operator is applied.

`form(file↓,operator↓,tracelevel↓)` Apply the PTN operator to the PCN source `file`. `file` may also be a list of files to which the operator is applied. The `tracelevel` flag is the same as in `tcomp(file,tracelevel)` above.

`form(file↓,operator↓,tracelevel↓,L↓,R↑)` Apply the PTN operator to the PCN source `file`. `file` may also be a list of files to which the operator is applied. The `tracelevel` flag is the same as in `tcomp(file,tracelevel)` above. Wait for `L` to be defined before running the transformation, and define `R` to be the same as `L` when the transformation has completed.

`trun(operator↓)` Execute the PTN operator, without any loading, storing, or compiling of programs and directives.

`trun(operator↓,tracelevel↓)` Execute the PTN operator, without any loading, storing, or compiling of programs and directives. The `tracelevel` argument is the same as in the `form` capability described above.

`trun(operator↓,tracelevel↓,L↓,R↑)` Execute the PTN operator, without any loading, storing, or compiling of programs and directives. The `tracelevel`, `L`, and `R` arguments are the same as in the `form` capability described above.

18.2 gauge

We type `load("gauge")` to load the Gauge profiler capabilities into the PCN shell. These capabilities support the collecting of profiles and the timing of runs. More information about Gauge is provided in § 15.

`profile(program↓,modulelist↓,nodes↓,file↓,done↑)` Collect a profile for a PCN program run. `program` is the PCN program to run, including arguments. `modulelist` is a list of module names for which profiles will be collected. `nodes` is the nodes on which the profile is to run, and is either an integer (e.g., 5), an integer range (e.g., 4-11), or the string "all". `file` is the file into which the profile will be written. A ".cnt" extension will automatically be added to this file name. `done` is a definitional variable that is defined to be the empty list ([]) when the profile is complete.

`profile(program↓,modulelist↓,nodes↓,file↓)` Collect a profile for a PCN program run. The arguments are the same as for the `profile` command described above.

`profile(program↓,modulelist↓,done↑)` Collect a profile for a PCN program run. The arguments are the same as for the `profile` command described above. The profile will be run on all nodes and the profile will be written into the file "profile.cnt".

`profile(program↓,modulelist↓)` Collect a profile for a PCN program run. The arguments are the same as for the `profile` command described above. The profile will be run on all nodes and the profile will be written into the file "profile.cnt".

`tprofile(program↓,rundone↓,modulelist↓,nodes↓,file↓,done↑)` Collect a profile for a PCN program run. This is the same as the `profile` capability described above except that the profile is taken when the `rundone` argument is defined instead of when the program completes.

`tprofile(program↓,rundone↓,modulelist↓,nodes↓,file↓)` Collect a profile for a PCN program run. This is the same as the `profile` capability described above except that the profile is taken when the `rundone` argument is defined instead of when the program completes.

`tprofile(program↓,rundone↓,modulelist↓,done↑)` Collect a profile for a PCN program run. This is the same as the `profile` capability described above except that the profile is taken when the `rundone` argument is defined instead of when the program completes.

`tprofile(program↓,rundone↓,modulelist↓)` Collect a profile for a PCN program run. This is the same as the `profile` capability described above except that the profile is taken when the `rundone` argument is defined instead of when the program completes.

`time(program↓)` Time the execution of the program run, `program`. The time is printed when the run completes.

`time(program↓,stats↑,done↑)` Time the execution of the program run, `program`. When the run completes `done` is defined and statistics from the run are defined to `stats` and can be printed using the `print_statistics` capability.

`ttime(program↓,rundone↓)` Time the execution of the program run, `program`. The time is printed when the run completes. The program is considered completed when `rundone` is defined.

`ttime(program↓,rundone↓,stats↑,done↑)` Time the execution of the program run, `program`. When the run completes `done` is defined and statistics from the run are defined to `stats` and can be printed using the `print_statistics` capability. The program is considered completed when `rundone` is defined.

`print_statistics(stats↓)` Print the statistics in the `stats` argument that was created by a previous use of the `time` or `ttime` capability.

18.3 upshot

The Upshot trace collector capabilities are loaded into the PCN shell by running `load("upshot")`. These capabilities support the collecting of traces from program runs. More information about Upshot can be found in § 16.

`log(program↓,size↓,nodes↓,file↓,done↑)` Collect a trace of a PCN program run. `program` is the PCN program to run, including arguments. `size` is the maximum number of events that can be logged on a single node. `nodes` is the nodes on which the trace is to run, and is either an integer (e.g., 5), an integer range (e.g., 4-11), or the string "all". `file` is the file into which the trace will be written. `done` is a definitional variable that is defined to be the empty list ([]) when the trace is complete.

`log(program↓,size↓,nodes↓,file↓)` Collect a trace of a PCN program run. The arguments are the same as for the `log` command described above.

`log(program↓,size↓,done↑)` Collect a trace of a PCN program run. The arguments are the same as for the `log` command described above. The trace will be run on all nodes and the trace will be written into the file "upshot.log".

`log(program↓)` Collect a trace of a PCN program run. The arguments are the same as for the `log` command described above. The trace will be run on all nodes, the trace will be written into the file "upshot.log", and the maximum number of events that can be logged on a single node is 10000.

`tlog(program↓,rundone↓,size↓,nodes↓,file↓,done↑)` Collect a trace of a PCN program run. This is the same as the `log` capability described above except that the completed when the `rundone` argument is defined instead of when the program completes.

`tlog(program↓,rundone↓,size↓,nodes↓,file↓)` Collect a trace of a PCN program run. This is the same as the `log` capability described above except that the completed when the `rundone` argument is defined instead of when the program completes.

`tlog(program↓,rundone↓,size↓,done↑)` Collect a trace of a PCN program run. This is the same as the `log` capability described above except that the completed when the `rundone` argument is defined instead of when the program completes.

`tlog(program↓,rundone↓)` Collect a trace of a PCN program run. This is the same as the `log` capability described above except that the process is completed when the `rundone` argument is defined instead of when the program completes.

18.4 vm_co

We type `load("vm_co")` to load compiler capabilities for process mapping into the PCN shell. These capabilities support the compilation of programs that use specialized process mapping facilities described in §PMT.

`vm(<files>↓)` Compile `<files>` (a single file name or a list of file names).

`vm(<files>↓, tracelevel↓)` Compile `<files>`, displaying debug information as specified by the integer `tracelevel`. (0 means no trace information; 4 is a lot.)

`vm(<files>↓, tracelevel↓, L↓, R↑)` As above, and in addition wait for L to be defined before running the compile, and define R to be the same as L when the compile has completed.

`link(<files>↓, "outfile"↓)` Link the specified file or files, creating a link module named `outfile.pam`.

`link(<files>↓, "outfile"↓, tracelevel↓)` Ditto, displaying debug information as specified by the integer `tracelevel`.

`link(<files>↓, "outfile"↓, tracelevel↓, L↓, R↑)` Ditto, and in addition wait for L to be defined before running the link, and define R to be the same as L when the link has completed.

19 Intel iPSC/860 Specifics

The PCN linker for the iPSC/860 is called `pncc.ipsc860`. This works in the same way as other versions of `pncc` (§ 9) but requires that you compile on a machine with i860 crosscompilers installed (typically a Sun SPARCstation). The following example uses the i860 crosscompiler `icc` to compile a C file `my_c.c` and a Fortran file `my_f.f` and then links the resulting object files with `foo.pam` using the `-fortran` flag on the PCN linker:

```
% icc -c my_c.c -node
% if77 -c my_f.f -node
% pncc.ipsc860 -fortran foo.pam
```

The "custom pcn" that is generated by the linker is named `pcn.ipsc860` by default. This name can be overridden by using the `-o` flag on `pncc.ipsc860`.

The custom pcn is run by logging into the iPSC/860 host (SRM), allocating the appropriately sized cube, and invoking the custom pcn. Once PCN terminates, we free the cube. In the example, we assume that the host is called `gamma`:

```
% rlogin gamma
% getcube -t 4
% runpcn
% killcube
% relcube
```

Notice that the custom pcn, `pcn.ipsc860`, is actually invoked by just running `runpcn`. The `runpcn` program is a small wrapper that locates a PCN executable by searching the current directory followed by the installation directory for a file called `pcn.ipsc860`. If this search is successful, it loads and executes this file on the iPSC/860 nodes.

The `runpcn` can also be called with a `-t` flag as a command line argument. In this case, it also handles allocation and release of an appropriate sized cube on the iPSC/860. For example, the following command will allocate a cube with 4 nodes, run `pcn.ipsc860`, and release the cube:

```
% runpcn -t 4
```

As mentioned above, `pcncc.ipsc860` can be asked to produce a custom pcn with a name other than `pcn.ipsc860`. The name of this custom pcn can be given as an argument to `runpcn`. The following commands illustrate this.

```
% pcncc.ipsc860 -fortran foo.pam -o mypcn
% rlogin gamma
% runpcn -t 4 mypcn -k 512
```

This example also shows how to pass command line arguments (in this case, `-k 512`) to the PCN executable. In general, all `runpcn` arguments are passed to the executable except for the `-t` argument (which must be first) and the custom pcn name (which must be second).

20 Intel Touchstone DELTA Specifics

Creating a custom pcn for the Intel Touchstone DELTA is identical to creating one for the iPSC/860, except that you should use `pcncc.delta` rather than `pcncc.ipsc860`. However, you must be careful that you use the DELTA versions of `icc` and `if77`, instead of the iPSC/860 versions. This is best achieved by changing your Unix search path.

Running the custom PCN on the DELTA is significantly different from on the iPSC/860, however, because the DELTA does not use an SRM (the front-end, i386-based Unix PC on the iPSC/860). Instead, all `.pam` files (including the system files) and the custom pcn need to be copied onto the DELTA's CFS filesystem using either `ftp` or `rcp`.

The `PCN_PATH` environment variable can be used to tell the custom pcn where to find the `.pam` files, in case the installation directory on CFS is different from that on the crosscompilation machine.

Once all relevant files have been copied onto CFS, the custom pcn can be run using the `mexec` command. This command specifies the height and width of the submesh to allocate, and the executable to load on the nodes in the submesh. For example, the following command would load the custom pcn called `pcn.delta` onto a 4 by 8 node mesh:

```
% mexec "-t(4,8)" -f pcn.delta
```

If you wish to supply arguments to the custom pcn, those arguments must be part of the `-f` flag:

```
% mexec "-t(4,8)" -f "pcn.delta -k 700"
```

21 Sequent Symmetry Specifics

Running PCN on the Sequent Symmetry is similar to running PCN on a workstation. The `pcncc` command is identical to that on workstations. The `-n` flag is used to run PCN with several nodes. For example, the following command starts a 10-node PCN run-time system:

```
% pcn -n 10
```

The Symmetry has two different C compilers which can be used to compile C foreign code. They are `cc` and `atscc`. `atscc` should be used if it is available, as it supposedly produces better code than the standard `cc` compiler. Fortran code should be compiled using the `fortran` compiler.

22 Symult s2010 (Cosmic Environment) Specifics

The Symult s2010 multicomputer uses the Cosmic Environment (CE) as its operating system. Hence, much of the following also applies to other CE systems.

The PCN linker for the CE is called `pcncc.ce`. This can be used for sun3 and sun4 host and ghost nodes, and the Symult s2010 nodes. The following example shows how one would make a custom pcn, called `mypcn`, for the sun4 host, using the PCN files `file1.pam` and `file2.pam`.

```
% pncc.ce -cehost -o mypcn file1.pam file2.pam
```

This assumes that `file1.pcn` and/or `file2.pcn` contain reference(s) to file(s) containing foreign functions.

We create a custom pcn called `mypcn.s2010` for the s2010 nodes as follows:

```
% pncc.ce -cenode s2010 -o mypcn file1.pam file2.pam
```

To run the custom pcn `mypcn` (i.e., `mypcn.ce` and `mypcn.s2010`) on 4 s2010 nodes, allocating a heap size of 240K on each node, we type

```
% getmc s2010 4 mem=4 /* allocate 4Meg nodes */
% mypcn -nk 240
% freecube
```

As a final example, we show how to create a custom pcn for ghost nodes, with the foreign code written in Fortran:

```
% pncc.ce -cenode gh -fortran -o mypcn file1.pam
```

23 Network Specifics

The network version of PCN (net-PCN) uses Berkeley stream interprocess communication (TCP sockets) to communicate between nodes. A node can run on any machine that supports TCP. Hence, a single PCN computation can run on several workstations of a particular type, several workstations of differing types, several processors of a multiprocessor, or a mix of workstations and multiprocessor nodes. Current restrictions are listed in § 23.6.

Net-PCN currently operates on the NeXT, Sun, DECstation, HP9000, IBM RS/6000, and SGI Iris.

Using net-PCN is the same as using PCN on other platforms except that the user must specify on which machines PCN nodes are to run and may also be required to specify where on those machines PCN is to be found and the commands necessary for running net-PCN nodes on the given machines.

There are several different ways of starting net-PCN, each appropriate for different types of network. We shall consider each of these in turn, starting with the easiest. First, we provide some background information on the Unix remote shell command `rsh`, which is used to start net-PCN nodes.

23.1 Using `rsh`

The Unix remote shell command `rsh` is a mechanism by which a process on one machine (e.g., `my-host`) can start a process on another machine (e.g., `my-node`). A remote shell command can only proceed if `my-host` has been given permission to start processes on `my-node`. There are two ways in which this permission can be granted.

- The file `/etc/hosts.equiv` exists on `my-node` and contains an entry for `my-host`. This file must be created by the system administrator.
- The file `.rhosts` exists in the home directory of the user running the remote shell on `my-node` and contains a line of the form

  ```
  my-host username
  ```

  where `username` is the name of the user login on `my-host`. This file is created by the user.

Some sites disallow the usage of `.rhosts` files. If `.rhosts` usage is disallowed and the host machine is not in `/etc/hosts.equiv`, remote shells cannot be used to create remote processes. Alternative mechanisms must be used, as described below.

The full syntax of the rsh command is as follows:

```
rsh hostname -l username command arguments
```

The `username` here is the login to be used on the remote machine. If `username` is not specified, it defaults to the login name of the user on the local machine. Furthermore, if the login name used on the local machine is different than the login name on the remote machine, the `.rhosts` file for the account on the remote machine must have an entry allowing access for that account on the host machine.

23.2 Specifying Nodes on the Command Line

The simplest way to start PCN on a network of machines is to use the -nodes <*nodelist*> argument to pcn, where *nodelist* is a colon separate list of machine names on which PCN nodes are to run. For example,

```
pcn -nodes pelican:raven:plover
```

will start a four-node PCN, with one node on the machine from which this command is run (the *host*) and one node on each of the machines named in the *nodelist*: `pelican`, `raven`, and `plover`.

This startup method only works if:

1. `rsh` (§ 23.1) works from the host to each machine in *nodelist*.

2. Each of the nodes shares a common filesystem with the host. The reason for this is that the host runs each node in the directory in which `pcn` is invoked. If the host and a node have different filesystems, the `rsh` used to start up that node is likely to fail.

If any of these conditions does not hold, then net-PCN must be started by using one of the alternative methods described below.

Note that we can always create multiple nodes on a single processor by using the `-n` command line flag. The command

```
pcn -n nnodes
```

forks `nnodes - 1` nodes on the local machine (resulting in a total of `nnodes` processes) which communicate using sockets. This can be useful for debugging purposes.

23.3 Using a PCN startup file

The second net-PCN startup method that we consider can be used if nodes do not share a common file system with the host. However, it still requires that `rsh` work from the host to each node.

This method uses a startup file to define the locations of remote PCN node processes. Lines in this file identify the machines on which nodes are to be started.

Startup File Syntax. A line of the form

fork *n-nodes*

causes *n-nodes* node processes to be started on the local machine. These nodes communicate with the other nodes via sockets, even though they are on the same machine as the host.

A line of the form

exec *n-nodes*: *command $ARGS$*

causes *command* to be executed. *command* is the command that invokes PCN on the appropriate machine. The host process replaces *$ARGS$* at run-time with the necessary arguments to PCN to cause it to start *n-nodes* node processes.

Blank lines in startup files and lines starting with whitespace, `%`, or `#` are ignored.

Examples of Startup Files. A startup file containing the lines

```
fork 1
exec 1: rsh fulmar pcn $ARGS$
```

starts one node on the local machine (in addition to the host node) and one node on the host `fulmar`, using the PCN executable called `pcn`.

A startup file containing the line

```
exec 1: rsh fulmar -l bob pcn $ARGS$
```

starts one node called pcn on host fulmar using the PCN executable pcn and the account for username bob. If we assume the PCN host is being run by user olson on host host-machine, then the .rhosts file in the home directory of user bob on fulmar must contain the entry

```
host-machine olson
```

A startup file containing the line

```
exec 3: rsh fulmar "cd /home/olson/pcn; ./custom-pcn $ARGS$"
```

runs three nodes on fulmar PCN executable custom-pcn after changing to the directory /home/olson/pcn.

A startup file containing the line

```
exec 2: sh -c 'echo "pcn $ARGS$ &" | rsh fulmar /bin/sh'
```

is a more complex example that starts up two nodes on fulmar. This example has the desirable side effect that the rsh process exits after starting the PCN node, whereas in the other examples the rsh will not complete until the node process completes.

Using a Startup File. We execute net-PCN with a startup file pcn-startup by using the -s flag on the PCN command line.

```
pcn -s pcn-startup
```

23.4 Starting net-PCN without rsh

If your computer system does not support the use of rsh, you will need to start remote nodes by hand or by using a utility called host-control. See the separate manual: R. Olson, Using host-control, Argonne National Laboratory Technical Memo ANL/MCS-TM-154.

23.5 Ending a Computation

Normally all nodes of a net-PCN computation will exit upon completion of the computation or upon abnormal termination of PCN. If for some reason this is not the case, you must log on to each machine that was executing a net-PCN node and manually kill the PCN process.

23.6 Limitations of net-PCN

PCN_PATH. You must be careful when using net-PCN on remote machines to ensure that the PCN nodes can find the .pam files they need. The nodes will execute with the PCN_PATH environment of your account on the host machine. If this environment variable is not set on the remote machine, the PCN installation directory will be used. If the needed .pam files are not in the path, the startup file must change to the correct directory.

Number of Nodes. The number of nodes available in a net-PCN computation is limited by the number of file descriptors available to a process (an operating system-imposed limit). On modern versions of Unix, there are generally more than sixty file descriptors available. Hence, in practice, this is not likely to be a major problem.

Heterogeneous Networks. Currently, no support exists for executing net-PCN between machines with different byte orders. We know that net-PCN does execute correctly between different machines if they use the same byte-ordering connection (we have run net-PCN successfully between Sun-3, Sun-4, and NeXT computers). However, you must be careful when using foreign code in this case, because, for example, structure packing in C may differ between different compilers.

24 Further Reading

PCN Language The basic text for the PCN language, which provides both an introduction to the language and a discussion of techniques used to reason about PCN programs, is

> M. Chandy and S. Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

Programming and Proof Techniques The following book provides a particularly readable and entertaining presentation of many of the basic parallel programming techniques used in PCN:

> I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, N.J., 1989.

The proof theory for PCN is based in part on that for Unity, which is described in detail in

> M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

PCN Toolkit The PTN program transformation tool is described in

> I. Foster, Program Transformation Notation: A tutorial, Technical Report ANL-91/38, Argonne National Laboratory

The `host-control` program used to manage network implementations of PCN is described in

> R. Olson, Using host-control, Technical Memo ANL/MCS-TM-154, Argonne National Laboratory.

PCN Implementation The techniques used to compile PCN for parallel computers are described in

> I. Foster and S. Taylor, A compiler approach to concurrent program refinement (in preparation).

A detailed description of the PCN run-time system can be found in

> I. Foster, S. Tuecke, and S. Taylor, A portable run-time system for PCN, Technical Memo ANL/MCS-TM-137, Argonne National Laboratory, 1991.

The design, implementation, and use of the Gauge performance analysis system is described in

> C. Kesselman, *Integrating Performance Analysis with Performance Improvement in Parallel Programs*, PhD thesis, UCLA, 1991.

Applications Papers describing PCN applications include

> I. Chern and I. Foster, Design and parallel implementation of two methods for solving PDEs on the sphere, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

> D. Harrar, H. Keller, D. Lin, and S. Taylor, Parallel computation of Taylor-vortex flows, *Proc. Conf. on Parallel Computational Fluid Dynamics*, Stuttgart, Germany, Elsevier Science Publishers B.V., 1991.

Part III
Advanced Topics

25 Customizing Your Environment

Your PCN operating environment can be customized in several ways through the use of Unix environment variables. For example, recall than when you execute an intermodule call or issue a `load` command, PCN searches first the current directory followed by the PCN installation directory for the necessary module (`.pam` file). However, we can change this module search path through the use of the `PCN_PATH` environment variable.

The following table shows the environment variables that can be used to customize PCN operations, along with brief descriptions and default values for each. The term `INSTALL` in a default value represents the directory in which PCN installed: normally, `/usr/local/pcn`. The defaults will be used if the environment variables is not set. That is, the environment variables override the default values.

Variable name	Description	Default value
PCN_PATH	Module search path	.:INSTALL/pams
PTN_OPERATORS_PATH	PTN operator search path	INSTALL/ptn/operators
PTN_LIB_PATH	PTN library search path	INSTALL/ptn/lib
PCN_LIB	PCN library directory	INSTALL/lib

Each of `PCN_PATH`, `PTN_OPERATORS_PATH`, and `PTN_LIB_PATH` is a colon-separated list of directories. Directories in a these lists cannot be relative to your home directory. That is, you cannot have "`~`" in the path. (For example, setting the `PCN_PATH` to "`~`/mypams" will *not* work.)

For example, to add your own pams directory to the PCN module search path, you would use the following command to set the `PCN_PATH`.

```
setenv PCN_PATH ".:/usr/local/pcn/pams:/home/me/mypams"
```

26 Run-Time System Debugger Options

The PDB version of the run-time system incorporates a variety of low-level execution tracing facilities. These facilities are controlled through the following four debug-level variables. The value of each variable can range from 0 to 9, with 0 meaning no trace output and 9 maximum trace output.

Emulator Debug Level : This controls debugging information in the main process scheduling loop. For example, level 2 causes all intermodule calls to be

printed, level 3 additionally prints the entry and exit of foreign procedures, and level 9 prints a complete trace of the PCN Abstract Machine instruction being executed.

Garbage Collector Debug Level : This controls debugging information in the garbage collector. For example, level 2 causes a short summary to be printed each time a garbage collection occurs.

Parallel Debug Level : This controls debugging information relating to the parallel aspects of the system. For example, level 5 causes debugging information about the low level message handling between nodes to be printed.

Global Debug Level : This controls debugging information not covered by the other three variables. For example, level 1 causes startup parameters and boot arguments to be printed.

The four debug levels can be manipulated in two ways. On a single node, they can be modified through the use of the PDB variables ($emulator_dl, $gc_dl, $parallel_dl, and $global_dl) described in § 14.6.

The debug levels can also be set from the command line. The following command line arguments set the various debug levels on all nodes, including the host.

`-d <level>` : This sets all debug levels.

`-e <level>` : This sets the emulator debug level. It overrides the level set by the `-d` flag.

`-g <level>` : This sets the garbage collector debug level. It overrides the level set by the `-d` flag.

`-p <level>` : This sets the parallel debug level. It overrides the level set by the `-d` flag.

The following command enables low-level trace information after the a specified number of procedure calls.

`-r <reduction_number>` : Do not print any debugging output until the number of procedure calls given by `reduction_number` have been executed.

The following command line arguments can be used to set debug levels selectively in different nodes of a multiprocessor.

`-node <node_number>` : Only apply the following node debug level flags to a particular node, `node_number`. If this argument is not used or `node_number` is -1 then apply the following node debug level flags to all nodes.

`-nd <level>` : This sets all debug levels on the appropriate node(s).

`-ne <level>` : This sets the emulator debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-ng <level>` : This sets the garbage collector debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-np <level>` : This sets the parallel debug level on the appropriate node(s). It overrides the level set by the `-nd` flag.

`-nr <reduction_number>` : Do not print any debugging output on the appropriate node(s) until the `reduction_number` reduction has been reached.

For example, the following command would set the emulator debug level to 3 and the garbage collector debug level to 2 on node 5 of a 10 node run.

pcn -n 10 -ne 3 -ng 2 -node 5

All debugging messages are preceded by the node number from which the message originated and reduction number on that node when the message was printed. When debug levels are set on multiple nodes simultaneously the debugging output from these nodes will be interleaved. The node and reduction number can help you sort out these interleaved messages.

Interleaving problems can be avoided by telling the run-time system to log all debugging messages to files, instead of to the screen, by putting a `-log` on the command line. This will cause the system to create a `Logs` directory into which all debugging output will be printed. Further, the debugging output from each node will be put in a separate file in this `Logs` directory.

Part IV
Appendices

A Obtaining the PCN Software

The PCN software is available by anonymous FTP from Argonne National Laboratory, in the pub/pcn directory on info.mcs.anl.gov. The latest version of this document is also available at the same location. The following session illustrates how to obtain the software in this way.

```
% ftp info.mcs.anl.gov
Connected to anagram.mcs.anl.gov.
220 anagram.mcs.anl.gov FTP server (Version 5.60+UA) ready.
Name (info.mcs.anl.gov:XXX): anonymous
331 Guest login ok, send ident as password.
Password:  /* Type your user name here */
230- Guest login ok, access restrictions apply.
Argonne National Laboratory Mathematics & Computer Science Division
All transactions with this server, info.mcs.anl.gov, are logged.
230 Local time is Fri Nov 8 18:26:39 1991
ftp> cd pub/pcn
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
pcn.v1.2.tar.Z
README
pcn_prog.ps.Z
pcn_prog.tar.Z
226 Transfer complete.
78 bytes received in 1.3e-05 seconds (5.9e+03 Kbytes/s)
ftp> binary
200 Type set to I.
ftp> get pcn.v1.2.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for pcn.v1.2.tar.Z (XXX bytes).
226 Transfer complete.
local:  pcn.v1.2.tar.Z remote:  pcn.v1.2.tar.Z
XXX bytes received in YY seconds (ZZ Kbytes/s)
ftp> quit
221 Goodbye.
```

B Supported Machines

The following table lists the machines on which PCN is currently supported, along with the architecture name.

arch name	machine name
dec5000	DECstation 5000 and 3100
delta	Intel Touchstone Delta
hp9k3	HP 9000 series 300 running HP/UX
hp9k8	HP 9000 series 800 running HP/UX
intel386	Intel 386 Unix PC running System V 3.2
ipsc860	Intel iPSC/860
iris	Silicon Graphics Iris
next040	NeXT
rs6000	IBM RS/6000
s2010	Symult 2010 running the Cosmic Environment
sun3	Sun 3 (Motorola 680x0 based)
sun4	Sun 4 (SPARC based)
symmetry	Sequent Symmetry

C Reserved Words

The following words may not be used as variable names or procedure names in PCN programs.

append_stream
char
close_stream
data
decrement_stream
default
directive
double
exports
foreign
init_recv
init_send
int
length
PCN
stream
stream_send
stream_recv
tuple
_p_*
_pdb_*

D Incompatibilities with Previous Releases

1. The sizeof() command has been changed to length(). It returns the number of elements in an array, or the arity of a tuple.

2. The sequential operator now sequentializes everything, not just operations involving mutable variables.

3. The PCN_PATH environment variable must now contain "." explicitly if you want the current directory to be searched when the system tries to load a .pam file. In previous releases, the current directory was always searched first, regardless of whether "." was in your PCN_PATH or not.

4. Meta operations now use `*var*` (matching back quotations) instead of '*var* (unmatched single quote) to denote a string that is to be interpreted as a variable name.

E Common Questions

What does it mean when PCN prints an `Illegal tag` message? This usually means that PCN internal data structure has been corrupted somehow. The usual way in which this happens is that user code writes past the beginning or end of an array (either in PCN or foreign code).

To help detect this situation: If you use arrays in your PCN code then you can do bounds checking by running the program under `pcn.pdb`. If you use arrays in Fortran code, many Fortran compilers have a flag to turn on bounds checking (also known as range checking).

When do I have to relink programs with `pcncc`? You only need to relink programs with `pcncc` when a foreign object file changes. In other words, if you change and recompile foreign code, then you need to run `pcncc` to link this new code into the run-time system.

If you only change the PCN portion of your program, then you should not have to relink everything with `pcncc`, although it does not hurt if you do.

F Known Deficiencies

1. The pcncomp program does not return a nonzero return code when compilation fails. This limits its utility in make files.

2. It is possible for PCN to exhaust its internal storage in an irrecoverable manner. If this happens in a single node run, a `No space left` message is printed and PCN terminates. If this happens in a multiple node run, a `No space - broadcasting` message is printed and PCN hangs indefinitely. (In the multiple node case, it is sometimes possible for PCN to recover, but it is not likely.)

   To work around this problem, try running PCN with a large internal data space (heap) by using the `-k` flag. By default, the heap size is 512 kilo-cells. For example, to double the heap size to 1024 kilo-cells run:

   ```
   pcn -k 1024
   ```

   To calculate the amount of memory required for a particular heap size, simply multiply the number kilo-cells of heap space by 8096. This yields the number of bytes of memory that will be allocated for heap space for this size heap.

3. It is possible for PCN to overflow its internal storage in an irrecoverable manner. If this happens, a `heap overflow` message is printed and PCN terminates.

   To work around this problem, try running your program using the `-gs` flag to increase the buffer zone at the top of the heap, which should reduce the chance of overflowing the heap. The default value for this parameter is 32 kcells. Try doubling or tripling this value. For example:

   ```
   pcn -gs 64
   ```

   Another strategy for working around this problem is to break up any procedures that allocate large amounts of array data into smaller procedures that allocate less storage.

G PCN Syntax

We present two BNF grammars for the PCN syntax. The first is that used by the parser; error messages printed by the PCN compiler include a rule from this grammar to indicate why parsing failed. The second is an expanded, more readable version of this grammar.

G.1 Parser BNF

```
Programs      : Form | Programs Form
Form          : Program | Directive

Directive     : EXPORTS Args | FOREIGN Args | DIRECTIVE Args

Program       : Heading Declarations Implication
Heading       : ID Names
Names         : '(' ')' | '(' NameList ')'
NameList      : ID | NameList ',' ID

Declarations  : /*empty*/ | Declarations Declaration
Declaration   : CHART Mutables ';' | INTT Mutables ';' |
                DOUBLET Mutables ';' | STREAMT Mutables ';'
Mutables      : Mutable | Mutables ',' Mutable
Mutable       : ID Dimension
Dimension     : /*empty*/ | '[' ']' | '[' INTEGER ']' | '[' ID ']'

Implication   : Block | Guard IMPLY Block
Block         : Var ASGN Exp | Var '=' Term | ID | Call | '{' Op Blocks '}'
Blocks        : Implication | Blocks ',' Implication
Op            : '|' '|' | '?' | ';' | STRING | ID Args
Guard         : Tests | DEFAULT
Tests         : Test | Tests ',' Test
Test          : ID MATCH Rhs | Con Eq Con | Exp Ar Exp | Type '(' Term ')'
Con           : Exp | STRING | '[' ']' | '{' '}'
Rhs           : List | Tuple | Call
Eq            : EQ | NEQ
Ar            : '<' | '>' | LEQ | GEQ
Type          : INTT | DOUBLET | CHART | TUPLET | DATAT | UNKNOWNT
Call          : LocalCall | RemoteCall | SystemCall
SystemCall    : '!' ID Args
RemoteCall    : LocalCall '@' INTEGER | LocalCall '@' QID PArgs
LocalCall     : QID ':' QID Args | QID Args
QID           : ID | "'" ID | "'" ID "'"
Args          : '(' ')' | '(' ArgList ')'
ArgList       : Term | ArgList ',' Term
PArgs         : /* empty */ | Args
```

```
Term      : STRING | List | Tuple | Exp | Call
List      : '[' ']' | '[' Elements ']' | '[' Elements '|' Term ']'
Tuple     : '{' Elements '}' | '' ''
Elements  : Term | Elements ',' Term

Exp       : ETerm | Exp '+' ETerm | Exp '-' ETerm
ETerm     : Factor | ETerm '*' Factor |
                     ETerm '/' Factor |
                     ETerm '%' Factor
Factor    : Num | '(' Exp ')' | LENGTH '(' ID ')'
Num       : Numeric | '-' Numeric | Var
Numeric   : INTEGER | REAL
Var       : ID Subscript
Subscript : /* empty */ | '[' Exp ']'
```

G.2 Expanded BNF

The following syntactic conventions are employed in this expanded BNF:

nonterminal ::= production

[ ]	Surround an optional element.
{ }	Surround an element that may occur zero or more times.
|	Separates alternatives.
boldface	Indicates reserved words.
"quotes"	Indicate characters that appear literally.

The symbols *unsigned-integer*, *unsigned-real*, *character-string*, and *identifier* denote terminal symbols and are not defined further here.

Comments are delineated by the start-comment symbol /* and the end-comment symbol */.

Compilation Module

compilation-module ::= program-or-directive { program-or-directive }
program-or-directive ::= program-declaration | directive

Directive

directive ::= directive-name "(" directive-arguments ")"
directive-name ::= **-directive** | **-exports** | **-foreign**
directive-arguments ::= [directive-argument { "," directive-argument }]
directive-argument ::= character-string

Program Declaration

```
program-declaration      ::= program-heading mutable-declarations program-body

program-heading          ::= identifier "(" formal-parameters ")"
formal-parameters        ::= [ formal-parameter { "," formal-parameter } ]
formal-parameter         ::= identifier

mutable-declarations     ::= { mutable-type mutable-declaration-list ";" }
mutable-type             ::= int | double | char
mutable-declaration-list ::= mutable-declaration { "," mutable-declaration }
mutable-declaration      ::= identifier [ "[" [ unsigned-integer | identifier ] "]" ]

program-body             ::= block
```

Block

```
block ::= assignment-statement |
          definition-statement |
          program-call |
          sequential-composition |
          parallel-composition |
          choice-composition


assignment-statement ::= variable ":=" expression


definition-statement ::= variable "=" term


program-call         ::= local-program-call |
                         remote-program-call |
                         system-program-call |
                         meta-program-call

local-program-call   ::= simple-program-call

remote-program-call ::= simple-program-call "@" annotation

system-program-call ::= "!" simple-program-call
```

```
meta-program-call   ::= identifier

simple-program-call ::= program-specifier "(" actual-parameters ")"
program-specifier   ::= [ module-name ":" ] program-name
module-name         ::= quoted-identifier
program-name        ::= quoted-identifier
actual-parameters   ::= [ actual-parameter { "," actual-parameter } ]
actual-parameter    ::= term

annotation          ::= unsigned-integer | character-string | quoted-identifier

quoted-identifier   ::= ' identifier ' | identifier
```

N.B. The single quote characters in the previous line indicate literally that character.

Sequential Composition

```
sequential-composition ::= "{" ";" block { "," block } "}"
```

Parallel Composition

```
parallel-composition ::= "{" "||" block { "," block } "}"
```

Choice Composition

```
choice-composition ::= guarded-block |
                       "{" "?" guarded-block { "," guarded-block } "}"

guarded-block      ::= guards → block
guards             ::= guard-list | default
guard-list         ::= guard { conditional-and guard }
conditional-and    ::= ","
guard              ::= pattern-match | equality-test | relational-test | data-test
```

```
pattern-match       ::= identifier "?=" pattern
pattern             ::= tuple-pattern | list-pattern
tuple-pattern       ::= "{" pattern-elements "}" |
                        identifier "(" pattern-elements ")"
list-pattern        ::= "[" pattern-elements "]" |
                        "[" pattern-element-list "|" pattern-element "]"
pattern-elements    ::= [ pattern-element-list ]
pattern-element-list ::= pattern-element { "," pattern-element }
pattern-element     ::= signed-number | character-string | identifier | pattern
```

```
equality-test       ::= equality-operand "==" equality-operand |
                        equality-operand "!=" equality-operand
equality-operand    ::= expression | character-string | empty-tuple | empty-list
empty-tuple         ::= "{" "}"
empty-list          ::= "[" "]"
```

```
relational-test     ::= relational-operand "<" relational-operand |
                        relational-operand ">" relational-operand |
                        relational-operand "<=" relational-operand |
                        relational-operand ">=" relational-operand
relational-operand  ::= expression
```

```
data-test ::= int "(" term ")" |
              double "(" term ")" |
              char "(" term ")" |
              tuple "(" term ")" |
              data "(" term ")"
```

Variable

```
variable ::= identifier [ "[" index "]" ]
index    ::= unsigned-integer | identifier
```

Expression

```
expression              ::= adding-expression

adding-expression       ::= multiplying-expression |
                            adding-expression "+" multiplying-expression |
                            adding-expression "−" multiplying-expression

multiplying-expression  ::= primary-expression |
                            multiplying-expression "*" primary-expression |
                            multiplying-expression "/" primary-expression |
                            multiplying-expression "%" primary-expression

primary-expression      ::= signed-number |
                            variable |
                            length "(" identifier ")" |
                            "(" expression ")"

signed-number           ::= [ "−" ] unsigned-integer |
                            [ "−" ] unsigned-real
```

Term

```
term                ::= expression |
                        character-string |
                        tuple-constructor |
                        list-constructor

tuple-constructor   ::= "{" elements "}" |
                        identifier "(" elements ")"
list-constructor    ::= "[" elements "]" |
                        "[" element-list "|" element "]"
elements            ::= [ element-list ]
element-list        ::= element { "," element }
element             ::= signed-number | character-string | variable |
                        tuple-constructor | list-constructor
```

Index

Distribution for ANL-91/32, Revision 1

Internal:

J. M. Beumer (20)
I. T. Foster (100)
F. Y. Fradin
G. W. Pieper
R. L. Stevens
S. J. Tuecke (100)
D. P. Weber
C. L. Wilkinson

ANL Patent Department
ANL Contract File
TIS Files (3)

External:

DOE-OSTI, for distribution per UC-405 (58)
ANL Libraries
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
W. W. Bledsoe, The University of Texas, Austin
P. Concus, Lawrence Berkeley Laboratory
E. F. Infante, University of Minnesota
M. J. O'Donnell, University of Chicago
D. O'Leary, University of Maryland
R. E. O'Malley, Rensselaer Polytechnic Institute
M. H. Schultz, Yale University
J. Cavallini, Department of Energy - Energy Research
F. Howes, Department of Energy - Energy Research