1 Introduction

The Parallaxis parallel programming project started in 1987, while Thomas Bräunl stayed at the University of Southern California, Los Angeles, on a Fulbright grant. After returning to the Universität Stuttgart, Germany, he developed a model for massively parallel computing and guided a first partial implementation, performed by students Bruno Schulze and Roland Becker. Since the completion of his Ph.D. thesis on Parallaxis in 1989 [Bräunl 89d] Thomas Bräunl has been directing the Parallaxis research project with a group of currently seven students working in several areas of the project. Ingo Barth crafted the Parallaxis compiler and Frank Sembach developed the PARZ intermediate code interpreter/simulator. Both heavily participated in changes and improvements of the high level language Parallaxis, the pseudo-assembly language PARZ, the interface between them, and the debugger. They used Apollo DN3000 and Sun 4 workstations to write their programs in C with the tools lex and yacc. This source code has been ported to the IBM PC and the Apple Macintosh. Karsten Krauskopf developed a preprocessor for Parallaxis, which enables the declaration and use of physical or user-defined dimensions and units for data values. This software package may later be distributed separately as public domain. Oliver Christ and Bruno Schulze developed a graphics interface tool for Parallaxis source programs for Sun under X-Windows. It reads a Parallaxis program and interactively displays the processor connection topology on a graphics screen. Finally, Stefan Engelhardt developed the compiler for translating intermediate PARZ programs to standard C (resulting in faster "compiled" Parallaxis programs as compared to the "interpreted" mode offered by the simulator), and also the compiler for translating PARZ programs to MasPar's MPL (their own parallel C extension), allowing Parallaxis programs to be executed on a true massively parallel system. A compiler translating PARZ programs to Thinking Machine's C* is in preparation; this will extend Parallaxis to the massively parallel Connection Machine and AMT's Distributed Array Processor.

This research report is an extended and updated version of [Barth Bräunl Sembach 90], which described version 1 of the Parallaxis system. New material has been included for version 2 of the software, and this report also covers all information contained in [Barth Bräunl Sembach 90]. Further papers published on the Parallaxis project are

in English: [Bräunl 89a], [Bräunl 89b], [Bräunl 89c], [Bräunl 90], [Bräunl 91b].
in German: [Barth 90a], [Barth 90b], [Bräunl 89d], [Bräunl 91a], [Engelhardt91], [Krauskopf 90], [Liebelt 91], [Schulze Christ 90], [Sembach 90a], [Sembach 90b], [Verba 90], [Walter 91].

The authors would like to thank Prof. Dr. G. Barth, director of the German AI research institute (DFKI) in Kaiserslautern, and Prof. Dr. A. Reuter, director of the parallel and distributed supercomputer institute (IPVR) in Stuttgart, for their support. We also thank Michael Ancutici, Claus Brenner, Karsten Krauskopf, Sabine Liebelt, Harald Nebel, Reinhard Verba, Volker Walter, and all the students from the Parallel Programming course at the Universität Stuttgart, who wrote a lot of Parallaxis code and had to discover quite a number of bugs in the early versions of the Parallaxis system. Your applications make the whole system worthwhile.

What's New in Version Two

In version 2 of the Parallaxis language definition and the Parallaxis Programming System we made some major extensions, the most important being the introduction of semi-dynamic connection structures. Parallaxis programs may now have several independent or overlapping interconnection structures at a time, and different connection structures at different times (linked to procedure scopes). We tried to be upward compatible as much as possible, however, there are two minor changes you might have to make in order to run your old (version 1) Parallaxis programs:

How to upgrade old Parallaxis Programs

- change power operator symbol from `"^"` to `"**"`
- suffix `load/store` commands in procedures without selection with `"[*]"` selection

Language Changes

Data Types and Operators

- Pointers
- Variant Records
- power operator "**"

Dynamic data structures and variant record structures have been added. Syntax and Semantics are identical to Modula-2. The power operator had to be changed to double asterik, since the `"^"` symbol is now used for dereferencing pointer expressions.

Multiple Configuration and Connection Structures

Several configuation and connection structures may be defined globally for the entire program, or locally for each procedure. Procedures with configuration/connection structures may not be nested inside each other. Multiple configuration/connection structures may be defined on disjointed sets of PEs (with the possibility of interconnections between these groups), each having its individual data declaration. Several configuration/connection structures may also be defined on the same set of PEs, resulting in an overlay structure.

PE selection syntax in `PARALLEL`, `LOAD/STORE`, and `REDUCE` statements has been changed slightly because of this extension. Besides the PE-range, a selection also has to specify the name of the configuration in case there are more than one (i.e.: `PARALLEL tree[3..7]`).

Connections are no longer restricted to be *1:1* connections. They may now be arbitrary *m:n* connections (that is any input and output port may have an arbitrary number of connections). The arrival of multiple data values at a time has to be avoided. Therefore, the data exchange operations may be supplied with implicit data reduction operations.

Data Exchange

- propagate [reduce]
- send [reduce]
- receive [reduce]

All participating PEs had to be active in a `propagate` data exchange operation. For multiple independent configuation structures, this is no longer possible. Therefore, `propagate` has been given two siblings: `send`, for transferring data **to** a destination-PE that does not have to be active (i.e. being in another configuration, or also inside the same configuration), and `receive`, for transferring data **from** a source-PE that does not have to be active.

All three data exchange operations may be specified with an implicit reduction operator, in order to take care of possible *m:n* connections. Using this operator, multiple imcoming values are reduced to a single value.

Standard Functions

- parallel `read`
- parallel `write`
- in/out-connected
- `new/dispose`

Read and write may now take vector arguments, reflecting the parallel I/O technology, called "Data Vault" at Thinking Machines and "Parallel Disk Array" at MasPar. In/Out-connected functions have been changed according to the new multiple connection structures, and dynamic storage allocation functions have been added.

Graphics Interface

A number of graphics routines have been included in Parallaxis to allow the building of an interface on a workstation with the **X-window** system (simulator and executables). Multiple windows may be opened (with manually positioning and **activating via a mouse click**) for graphics output.

Reduction

User-defined reduction functions now have to have two **vector** arguments and a **vector** result.

Programming System Changes

Compiler

The compiler now optionally creates cross-reference files for the symbolic debugger of the simulator.

Simulator and Debugger

The simulator also allows variant data structures (unions) and supports dynamic connection structures. That is, the PE connections are no longer listed before the actual commands in each PARZ program, but there are vector commands `connect`, `biconnect` and `disconnect` to build and discard dynamic connection structures. As a consequence, there is now a debugger command for displaying the dynamic connections (formerly integrated in the `LIST` command).

The debugger has been extended to a symbolic debugger that operates on Parallaxis level and on PARZ level (new commands handling Parallaxis expressions are `EXAMINE` and `ASSIGN`). Parallaxis source code is always displayed, when available. All debugger commands formerly using PARZ labels only, may now also use Parallaxis source line numbers.

New Tools and Further Compilers

The visualizer tool is being discussed briefly. However, it is still not ready for version 2 distribution. There are two all new PARZ-to-C compilers. The first one is to speed up Parallaxis programs on single-processor Unix workstations, and the second compiler has been especially designed to run Parallaxis programs on the massively parallel MasPar computer system (16,384 PEs).

2 Installing the Simulation System

On a Unix system:

- transfer all files of the supplied HD floppy disk from an IBM-PC AT to the workstation in **binary mode** (via kermit, ftp, etc.)
- uncompress the files (Unix command "`uncompress`" for suffix "`.Z`")
- open archive file (Unix command "`tar`" for suffix "`.tar`")
- copy the executable files `pa` and `pz` to the local /bin directory
- copy the manual pages `pa.l` and `pz.l` to the local /man directory
- copy the example programs as read-only to a public directory

On an IBM-PC or Compatible:

- be sure to use a model AT or higher, since the supplied floppies are HD
- copy the executable files `pa.exe` and `pz.exe` to a directory searched by "path"
- copy the example programs to a subdirectory
- for the DOS operating system's limit of 640 KB, so **severe memory space restrictions** apply for both, compiler and simulator. That is only smaller application programs may be compiled and executed !
- the OS/2 version 2.0 operating system is **strongly recommended** (use the appropriate version of the Parallaxis software) !

On an Apple Macintosh:

- be sure to use a double sided disk drive (800 KB)
- copy the executable files `Compiler` and `Simulator` (if you have the version without floating point co-processor) or `Compiler` and `SimulatorFP` (if you have the version with co-processor and 68020 CPU), respectively.
- increase the programs' memory allocation depending on your Mac's memory
- copy the example programs into the same or a separate folder
(be aware of colons in *full* Mac-filenames, e.g. `:samples:find.p`)
- in case you are using a Macintosh with only 1 MB of memory, some space restrictions apply for both, compiler and simulator, that is only smaller application programs may be compiled and executed
- a memory extension to 2 MB or higher and the usage of multi-finder is highly recommended for medium size and large application programs
- the compiler has a user-interface in Macintosh style:
all options are set via **menu selection**, so there is no need for command line arguments as described in the following section
- the simulator will be launched automatically on double-clicking a compiled file
- explicit starting of the simulator brings up a Unix-like command line, where options and I/O-redirections may be entered; without options a file selector box appears

Installation of tools and the intermediate code compilers is described with the software only.

3 Using the Parallaxis Compiler

The compiler is started by typing

```
pa {options} in_filename [-o out_filename]
```

The compiler takes files ending with `.p`, `.para` or `.pre` as input files. If the filename entered has none of these extensions, the compiler adds one and tries to open this file. If no file with a standard extension can be found, the compiler tries the given filename without an extension. Options start with a '-' sign. All characters after the '-' sign up to the next white space are treated as options. A compiler call may contain more than one option.

The options are:

d ignore all DEBUG and TRACE commands.
e`num` set the number of error messages reported to the console to that number. In a listing-file all error messages are reported.
h use every temporary variable only once. This option should only be used in combination with an optimizer.
l generate a listing file. The name of the listing file is built by removing the standard-extension and adding the extension `.list` (or `.lis` on a PC). All error- and warning messages are reported in the listing file.
m don't generate PARZ code for operand-checking in mathematical operations (division0, ln(0)). These checks may be omitted when using the PARZ simulator.
n don't generate PARZ code to check for dereferencing of NIL pointers.
p use the compiler as a filter, that is reading from standard input and writing to standard output. In this case, option `l` will be ignored.
r don't generate PARZ code for range-checks. However, be aware that range-errors may **not** be detected by the PARZ simulator.
u distinguish between lower case and capital letters.
v verbose mode, give information about the compilation.
w`num` set the number of warning messages reported to the console to that number.In a listing file all warning messages are reported.
x generate a cross reference file. The xrf-file contains the additional information to use the symbolic debugger in the PARZ simulator. The name of the xrf-file is built from the output filename by removing an extension `.z` and adding the extension `.xrf`. If the compiler is used as a filter, then the xrf-file is called `$xrf`.

The generated PARZ code is normally written to a file. The name of this file is built by removing the standard extension from the infilename and adding the extension `.z` to it. If option `-o out_filename` is provided, this filename is used for the PARZ program.

If an error has occurred, no PARZ code will be generated. The compiled PARZ code contains additional information to allow the PARZ simulator to show the source lines of the program. The simulator generates the source-filename from the PARZ filename.

For testing massively parallel algorithms, the PARZ simulator may be used to report the execution of a program step by step. With a pseudo-comment `$R` {proc_ident} within a Parallaxis comment, code is generated to report these procedures. If a procedure which is reported calls a non-reported procedure, only the procedure call will appear in the reporting file.

The same pseudo-comment `$R` {proc_ident} is treated by the PARZ-to-C and PARZ-to-MPL intermediate code compilers not as a recording option, but as a timing option (which is analog and makes much more sense in this case). Measured run times (CPU-time) for all procedures specified will be printed on standard output.

The pseudo-comment `$D` {proc_ident} makes the compiler generate debugging code for the procedure names given.

4 Using the PARZ Simulator and Debugger

4.1 Using the PARZ Simulator

The PARZ simulator is started by typing

```
pz {options | filename}
```

Options and filenames are separated by blanks. If both are missing, or if a wrong option is given, PARZ writes the list of all valid options and terminates. Options are single letters preceded by a dash; some of them may be followed by a number or a filename. Options may be merged into a single option word (with some exceptions, see below):

Example: `pz -l -c -e10 myfile` is equivalent to `pz -lce10 myfile`

The arguments of the `pz` command are processed from left to right, according to the following rules: A filename identifies a program that is loaded and executed (exception: option `-l`). Filenames ending with `.p` are treated as Parallaxis programs, files with trailing `.z` are PARZ programs. An option may be overridden by another option later in the command line.

List of Command Line Options

c [0|1] compile (UNIX-systems only)
`1` : the following filenames identify Parallaxis programs that have to be compiled by the compiler `pa` to a PARZ program, prior to execution.
`0` : the following filenames identify PARZ programs.
Compilation is off by default, files ending with `.p` are always compiled.

d [0|1] debug mode
The output of `DEBUG`- and `TRACE`-commands is enabled (by `1`) or disabled (by `0`). By default, debug output is enabled.

e [0-9]+ error messages
Limit the number of error messages and warnings when reading a program. Default limit is set to 15 messages.

f filter
Read a program from standard input and execute it. No filename may appear after this option. The options `-c`, `-l` and `-m` have no effect. If a single-step mode is activated by `-s`, the single-step information is printed, but simulation does not stop after each step. The program loaded should **not** try to read from standard-input, as correct input cannot be guaranteed after reading a PARZ program from the same stream. Trying to interrupt the simulation from the terminal (Control-C), will terminate the simulator.

l load
If a filename follows in the command line, this file will be loaded, but not started. Otherwise, an already executed program will remain in memory. The simulator will enter command level, indicated by prompt " `P>` ".

m [0-9]* more - screen size (set screen size for scrolling, like `more` in UNIX)
Defines the number of lines after which output on command level is interrupted.
If the number is zero, `more`-like output is disabled. If the number is missing, `more` is reactivated. Default is screen size 23 (or window size for Macintosh).

r [0|1|2] [name] recording mode
Defines format and filename of output-recording.
`0` : no recording
`1` : short recording : command type and number of active PEs
`2` : long recording : complete command with arguments and PE-activity-status
See Appendix B for details.
`name` is the name of the recording-file. If not specified, the recording filename is derived from the program filename. If `name` is specified, this option may not be merged with other options in one word.
Default: no recording.

s [0|1|2] single-step mode
`0` : no program interrupts
`1` : program interrupted before each Parallaxis statement
`2` : program interrupted before each PARZ command
Default is: no program interrupts.
At each interruption, the simulation status is printed and the simulator enters execution level, indicated by prompt "`P* `".

w [0|1] runtime warnings
Enables (`1`) or disables (`0`) runtime warnings.
By default, runtime warnings are enabled.

If the digit in options -c, -d, -r, -s or -w is omitted, it is assumed to be '`1`'. If the last argument of `pz` is the only filename and if no option -l is given, the program is loaded and executed without further comment. Otherwise, for each file read, the simulator writes the names of the source-files and recording-files to the terminal.

4.2 Using the PARZ Debugger

The debugger has two levels: the **command level** (prompt "`P>`"), and the **execution level** (prompt "`P*`"). Command level is reached directly by option -l, while execution level is reached when a simulated program is interrupted. Both levels share the same commands as described below with a few exceptions. Commands consist of a command identifier and arguments qualifying the command. Identifier and arguments are separated by white space; each command is entered by pressing the `<RETURN>` key. Command identifiers and argument identifiers may be abbreviated. Between several argument identifiers, each abbreviated by one character, the delimiting white space may be omitted. Except in strings, lower case and capital letters are not distinguished. The command syntax is described in EBNF (Extended Backus-Naur Form) with the following additions:

X(YZ) : each abbreviation of `XYZ`, at least `X`. E.g. '`ST(EP)`' may read '`ST`', '`STE`' or '`STEP`'
's' : the string s, literally
{ x }+ : non-empty list of x.
{ x },* : list of x, separated by commas.
{ x },+ : non-empty list of x, separated by commas.
num : a positive integer number

line : a Parallaxis procedure name: the first command of the procedure is referenced.
or a num: *In Parallaxis-mode* (see `MODE`): the command according to this or the next higher sourceline is used.
In PARZ-mode: the first command with that label is referenced.
label : like line in PARZ-mode
pelist : optional list of PE-numbers
<u>Syntax:</u> [PE ({ num ['..' num] },+ | '*')]
<u>Example:</u> 'PE 1..3,10..11,5' : the PEs 1,2,3,5,10 and 11
'PE *' : all PEs
selection : Parallaxis selection without configuration name.
expr : Parallaxis expr: for syntax see Appendix A.

<u>Debugger Commands</u>

A(SSIGN) [selection] designator ":=" expr
Assigns the value of `expr` to the Parallaxis designator `designator`.
`expr` is evaluated on all currently active PEs.
If `selection` is specified, the assignment takes place on the selected PEs only.
<u>Example</u>: <u>`P*`</u> `A x := 4 * id_no` ⇒ assigns the value of expression `4 * id_no` to the vector integer variable `x`

B(REAKPOINT) [[sign] line [offset]]
The BREAKPOINT command is used to show, set, and remove breakpoints. Execution stops if a command marked by a breakpoint is reached. The simulator will be in execution mode and print the current simulation status.
sign = '+' | '-'.
offset = sign num.
If `sign` is '+' or missing, a breakpoint will be set, otherwise it will be removed.
`line` and `offset` define the PARZ command to be considered:
If `line` is a procedure name: breakpoints are set on all procedures with that name, no `offset` is allowed.
If `line` is a number: <u>in Parallaxis-mode</u>, no offset is allowed, `line` is the line number in the source file.
<u>in PARZ-mode</u>, the command with distance `offset` from the command labelled `line` is taken; an offset is only needed for commands without a label.
<u>Example</u>: <u>`P>`</u> `B 75` ⇒ set breakpoint at command on Parallaxis source line 75

C(ALLS) [A(CTIVITY)]
This command shows the current call-stack.
For each procedure on the call-stack, the following is printed:
- the name of the Parallaxis-procedure, if known
- the label of the PROC-command
- the source line of the procedure call, if known
- the label of the CALL-command that called the procedure
If A(CTIVITY) is present, the PE-activities, saved by conditional calls, are printed too.

Example: `P* C` ⇒ shows the call stack

CO(NNECTIONS) { cpat },*

Lists all or some of the established connections between ports.

cpat = enum enum TO enum enum .
enum = (num ['..' num]) | '*' .

The list of connection patterns `cpat` contains all the patterns, a connection may match for being printed. An asterisk marks an arbitrary PE-number or port-number. "x .. y" denotes a range of numbers. If no `cpat` is given, all connections are listed.

Examples: `P* CO 2 * TO * *` ⇒ all connections leaving PE 2
`P* CO 2..10 * TO 2..10 *` ⇒all connections between PEs 2 through 10.

D(EBUG) [Y(ES) | N(O) | '1' | '0'] pelist

Like option -d, this command enables or disables the commands DEBUG and TRACE.

In addition, the output produced by these commands may be limited to variables on specific PEs. Such a limitation has an effect on commands SE(T) and S(HOW) without `pelist`. The `PE` in `pelist` can be abbreviated as `P`. Y(ES) and 1 enable the output of DEBUG and TRACE, N(O) and 0 disable it. If `pelist` is specified, it limits the output of DEBUG and TRACE and installs a default `pelist` for SE(T) and S(HOW).
If the DEBUG-command is given without arguments, the present settings are printed.

Example: `P> D 1 PE 1..5` ⇒ activates DEBUG and TRACE for PEs 1 to 5 (SET and SHOW now have effect on these PEs)

E(XAMINE) [ProcPath ["!" depth]] [selection] {expr},+

The EXAMINE command is used to show the values of Parallaxis expressions.

ProcPath = {'!' Ident}+
Each `Ident` in `ProcPath` is either the system name or a procedure name of the source program.
`ProcPath` is the path to find a procedure in the procedure hierarchy, when starting on the active scope. `ProcPath` defines the desired viewpoint.
`depth` is the count of recursive activations of the procedure defined by `ProcPath`.
The Parallaxis expressions are evaluated on all actually active PEs.
If `selection` is given, the results are only shown for the selected PEs.

Example: `P* E !foo!3 i, value > limit` ⇒ shows the values of variable `i` and expression `value > limit` in the three last recursive activations of procedure `foo`.

G(O) [S(TEP) [O(VER)]] [T(O) line]

On command level, the program is started; on execution level, an interrupted program is continued.

With S(TEP), the simulation status is printed after each step. If O(VER) is also present, proper procedures are simulated in a single step, but not conditionally called procedures. In Parallaxis-mode, each step is one Parallaxis statement, marked with '!' in the PARZ program by the compiler. In PARZ-mode, a step is a single PARZ command.

With argument T(O), the simulation will be interrupted just before the command indicated by 'line' is executed. Execution will always be interrupted when a breakpoint is reached or if an interrupt is generated from the console (Control-C).
When the program is interrupted, the following is printed:

- reason of interruption (multiple reasons are possible)
- PARZ command that would be executed next
- line number in source code, if known
- part of source code containing that line, if known
- PE activity status

If simulation reaches one of the commands HALT or END, execution terminates. If execution has been started from the command level, the simulator will return to this level, while data from the program execution (e.g. variables, stacks) can still be examined with the command S(HOW). If the program was directly started from the operating system, PARZ continues to process the command line, e.g. loading and executing new programs, or terminating.

Example: `P> G` ⇒ starts simulating the loaded program

H(ELP) [command_id]
? [command_id]

This command supplies help information.

Without `command_id`, a list of all commands is given. If `command_id` is a valid command identifier, a description of syntax and functionality of that command is given.

Example: `P> H G` ⇒ explains the GO command.

L(OAD) file ['-' C(OMPILER)]

This command is used to load a program into the simulator and replace a previously loaded program.

The simulator will enter command level if it was started with command-line option `-l`; otherwise, the loaded program will be executed immediately.
`file` is an arbitrary string without white space, representing a valid filename (a standard extension will be added automatically, if it is not provided).
The Parallaxis compiler `pa` is invoked with the named file as argument, if

- the argument -C(OMPILER) is given,
- `file` is ending with `.p` or
- `file.p` exists but not `file.z`.

The file `$xrf` generated by the compiler is loaded as cross reference file.
Otherwise `file.z` or `file` is loaded as a PARZ program. If a matching file with `.xrf` ending exists, it is loaded as cross reference file.

Example: `P* L program` ⇒ substitutes the interrupted program by "`program.z`"

LI(ST) [trange]

The loaded PARZ program or parts of it are printed on the console.

If program-faults have been discovered during loading, they are marked "`@@`" in the listing. A correct program is printed the way it was read. All keywords and identifiers are converted to capital letters. Global declarations are only listed together with the first command.

trange = (label ['..']) | ([label] '..' label) .
The optional argument `trange` selects program-parts by label:

no `trange`: all commands
x : the command labelled x
x .. : from command x to the last command
.. y : from the first command to y
x .. y : from command x to y

Example: `P> LI` ⇒ lists the global declarations and all program commands.

Two derivatives of the list command and only for Unix systems are:

LI(ST) S(TRINGMATCH) "string"

All lines in the output of LIST TEXT which also contain `string` are listed.
Lower case and capital letters are not distinguished.

Example: `P> LI S "@@"` ⇒ lists all lines that contain an error marker

LI(ST) P(ATTERN) "reg_expr"

The same as LIST STRINGMATCH, but `reg_expr` is a regular expression as used by the Unix-command `grep`.

Example `P> LI P "PROC.*VECTOR"` ⇒ lists all PROC-commands that declare local vector-variables

M(ODE) [C(OMPILER) | I(NTERPRETER)]

This command switches between Parallaxis- and PARZ-mode.
Without argument the active mode is shown. The default setting at start is Parallaxis-mode. Argument C(OMPILER) switches to Parallaxis-mode: line numbers in BREAKPOINT, GO and STEP commands now define source code lines. In single-step mode, execution stops before Parallaxis statements.
Argument I(NTERPRETER) switches to PARZ-mode. Now line numbers refers to PARZ labels and each PARZ command is a single step.

Example: `P* M I` ⇒ switch to PARZ mode (interpreter)

N(OTRACE) [vardesc]

Value tracing as set for specific variables by the TRACE command in PARZ or the T(RACE)-debugger command can be deactivated for single variables or for all blocks where it was active.
The NOTRACE debugger command has the same form and effect as the NOTRACE PARZ-command. `vardesc` is a PARZ variable; if it is indirect, the index-variable mustn't be a vector. If `vardesc` is specified, tracing is disabled for all memory blocks (that is Parallaxis variables), beginning with that PARZ variable. Without `vardesc`, all tracing is disabled.

Example: `P* N SI0:1` ⇒ disables tracing on all blocks beginning with `SI0:1`

Q(UIT)

This command terminates the simulator, and returns to the operating system.

R(ECORD) [O(FF) | N(O) | '0' | S(HORT) | '1' | L(ONG) | '2']

The simulator has two forms of recording commands of PARZ programs. Similar to the command-line option -r, the RECORD-command can choose between these forms or disable recording.

O(FF), N(O), `0` : Recording is disabled.
S(HORT), `1` : Record command-character and number of active PEs (vector commands only)
L(ONG), `2` : Record each command and PE-activity status for vector commands

Every change to the record-mode and all load-activities are mentioned in the record-file. If no commands are recorded in a record-file, the file will be removed when exiting the simulator. Commands of any recording mode which are marked `R` in the label-definition will always be recorded. Without arguments, the RECORD command prints the current recording mode. See appendix B for details.

Example: `P* R 2` ⇒ activates the long recording format.

S(HOW)

This command shows the current memory contents. Several subcommands allow different views of the memory.
In the following descriptions of the sub-commands holds:
adrlist = { (adr ['..']) | ([adr] '..' adr) },* .
adr is an integer number, used as an absolute address.
`pelist` always specifies the PEs, the data of which will be shown. If a `pelist` is omitted, the default PE-list (set by the DEBUG-command) is used.

1.) Show memory contents, ordered by absolute addresses

S(HOW) (M(EMORY) | H(EAP)) [S adrlist] [V adrlist] [pelist]

Data in activation records have positive addresses, global data starts at address `1`. Data closer to the activation record of the actual procedure has a higher address. The SHOW MEMORY command shows each PARZ variable value together with its absolute address, the PARZ variable descriptor, the PROC-label of the procedure declaring the variable, and (if known) the name of the corresponding Parallaxis procedure.
Data allocated on the heap by the NEW-command have negative addresses. The block allocated first, gets `-1` as highest address. Blocks allocated later reside at lower addresses in heap space. Argument M(EMORY) shows all memory-cells, while H(EAP) accesses the heap (i.e. negative addresses) only. Arguments `S` and `V` define the address ranges on the control processor (S = scalar) and on the PE-array (V = vector) that will be shown.
`adrlist` is a list of address-ranges, the contents of which will be shown successively. If an `adrlist` is empty, all addresses of the appropriate memory type are shown. If both arguments `S` and `V` are missing, the whole memory will be shown. If only one argument is specified, that memory part is shown. If one of the address-ranges is out of range with both bounds, an error will be reported, however, if a part of a specified address range exists, it will be shown.

Examples:
`P* S H` ⇒ shows the whole scalar and vector heap.
`P* S MS` ⇒ shows the whole scalar memory.
`P* S H S V -3..-1 PE 1,4` ⇒ shows the whole scalar heap and the heap addresses -3 to -1 on PEs 1 and 4.

2) Show contents of parameter-stacks

S(HOW) P(ARSTACK) [S [depth]] [V [depth]] [pelist]

The scalar and vector parameter stacks are manipulated by the PARZ-commands PUSHS, POPS and PUSHV, POPV. The command SHOW PARSTACK shows a specific number of stack elements, beginning at the top of stack.

`depth` is an integer > 0.

Arguments `S` and `V` specify the number `depth` of elements to be shown of the scalar and vector stacks. If `depth` is missing, the whole stack is shown. If both arguments `S` and `V` are missing, all stacks are shown completely.

'`* ToS *`' in the output marks the top of a stack. If '`* BoS *`' (bottom of stack) terminates the output, the whole stack was shown, otherwise the end of printing is marked '` ... `'.

Example: `P* S P V5` ⇒ shows the top five elements of the vector parameter stacks

3) Show callstack and memory contents ordered by activation records

S(HOW) C(ALLSTACK) [C(HAIN) [A(CTIVITY)]]
[S [depth]] [V [depth]] [pelist]

Each procedure call generates an activation record on top of the callstack, which is removed by the corresponding RETURN command. The following data held in an activation record can be shown by the command SHOW CALLSTACK:

- label of the PROC command and the corresponding procedure name in the Parallaxis-program (if known)
- label of the calling command and the source-code line number (if known) with the calling Parallaxis statement
- PE-activity status saved by conditional calls
- local scalar and vector data of a procedure or global data in the activation record of the main program.

`depth` is an integer > 0.

In the output, each PARZ variable is identified by its absolute address and the PARZ variable descriptor. Without arguments, all information in the callstack is shown. By C(HAIN), the whole call-chain information is shown. With A(CTIVITY), the saved PE-activity status information is shown as well. The arguments `S` and `V` specify the number `depth` of activation records starting with the actual activation record that will show their scalar or vector data. If argument `depth` is missing, all activation record variables of the corresponding memory type are shown. If only one of the arguments `S` or `V` is present, only variables of that type are shown. If variables of an activation record are shown, the calling information is printed too (otherwise only shown with arguments CHAIN ACTIVITY).

Example: `P* S CV3 PE1..4` ⇒ shows the calling information and vector variables on. PEs 1 through 4 of the topmost 3 activation records

4) Show memory-blocks, corresponding to Parallaxis-variables

S(HOW) V(ARIABLE) vardesc [P(ROC) label [D(EPTH) depth]]
pelist [AS declaration]

Similar to the DEBUG-command, a PARZ-variable and a declaration define a memory-block, that represents a Parallaxis-variable.

`vardesc` and `declaration` define a memory block as described for the DEBUG-command. If `AS declaration` is missing, the memory block consists of `vardesc`.

If `pelist` is omitted, only variables on active PEs are shown. The output format is identical to the one used by the DEBUG command. If an indirect address points to a nonexisting vector-variable, this will be indicated by the string '`<non-ex>`', if the variable exists on other PEs.

P(ROC) changes the view on the stack to the topmost activation record of the procedure with PROC-label `label`. This may be useful if more than one procedure with the same nesting level is on the call-stack.

If the PROC with `label` has been called recursively, argument D(EPTH) defines how many different activations of the procedure are looked at. With P(ROC), for each memory block shown, the distance of the viewpoint to the top af the callstack is stated. The actual activation record has distance zero.

Example: `P* S V VI1:7 PE2..5`⇒ shows PARZ variable `VI1:7` on PEs 2 through 5

5) Show predefined variables

S(HOW) S(PECIALS){ MAXTRANS | ACTTRANS | DONE | TERMCH |
SRESULT | VRESULT },* pelist

Predefined variables, which are either read-only or write-only, are treated specially by the simulator, so a special subcommand exists to show their contents.

If a list of variable names is specified, their contents are shown, otherwise the contents of all predefined variables is printed.

`pelist` is effective only for printing VResult.

Example: `P* S S` ⇒ shows the contents of all predefined variables

SE(T) vardesc pelist ['-' | NOT] VarConst

With this command, memory contents may be manipulated from execution level. This command can be executed on both levels, but changes made on command level won't have any effect, since the starting PARZ program will destroy all previous data.

This command has the same effect as the PARZ-command: `vardesc := [-|NOT] VarConst`

`pelist` specifies the PEs that execute the assignment, if `vardesc` is a vector-variable.

If `pelist` is missing, the default PE-list is used, as defined by the DEBUG-command.

Example: `P* SE VI3:2 ID` ⇒ writes the PE-number to `VI3:2` on each PE.

ST(EP) [O(VER)] [(T(O) line) | A(LL)]

This command executes one step of the program. In PARZ-mode, each step is one PARZ command, in Parallaxis-mode, a step is a Parallaxis statement, with the first PARZ command marked '`!`'.

After each step, the following is printed:

- the PARZ command to be executed next,
- the PE-activity status,
- the Parallaxis source-line number, if known,
- three lines of the Parallaxis source code containing the current line, if known.

With O(VER), procedures that are not conditionally called, are executed in one step.

T(O) `line` defines a PARZ command that can be reached stepwise, by just pressing the `RETURN`-key. If the command, specified by `line` is reached, this will be reported and `<RETURN>` has no special meaning any more. The stepping-function of

<RETURN> will as well be stopped by another GO- or STEP-command. Argument A(LL) works like T(O), but no line is defined that will end the stepping-function of <RETURN>.

Example: `P> ST A` ⇒ goes to the first command, then executes one step, each time <RETURN> is pressed

STO(P)

On execution level, the interrupted program will be terminated. The simulator returns to command level if it was started with command-line option `-l`. Otherwise, command-line processing continues (exit, if this file was the last file in the command-line).

T(RACE) vardesc string AS declaration

This command has the same form and effect as the TRACE command in a PARZ program. A memory block is defined that will be printed each time a write-access occurs.

`vardesc` and `declaration` describe a memory block that will be traced
`string` will be printed together with the data block

Example: `P* T VI0:2 "multi1-2" AS I2` ⇒ activates tracing for the two successive variables `VI0:2` and `VI0:3` with header "multi1-2"

W(ARN) [Y(ES) | '1' | N(O) | '0']

In some situations, like uninitialized data or inactive PEs, runtime warnings are generated. The WARN command allows to suppress these warnings like the command-line option -w.

Y(ES), 1: runtime warnings are generated
N(O), 0: runtime warnings are suppressed
If no arguments are given, the present setting is printed.

Example: `P* W 0` ⇒ no runtime warnings will be generated

WI(DTH) [num]

All bitstrings, representing PE-activity are printed in a block with a default width of 50. The WIDTH command is used to adjust that block width to the structure of the PE-array.

If `num` is present, it sets the new output width for bitstrings,
otherwise the existing setting is reported.

Example: `P> WI 30` ⇒ all bitstrings are printed with width 30

The Empty Input

Normally, the empty input has the only effect of generating a new input prompt. After a command STEP TO or STEP ALL, the empty input causes the simulator to execute one program step. If a continuous simulation is interrupted from the terminal (Control-C), pressing <RETURN> has the same effect as the command STEP ALL.

MORE-like output

If MORE-like output is active on command level (see option -m), the following prompt appears after each screen full with printing:

-- MORE (help with h) --

The following inputs are possible; they must not be terminated by <RETURN>:

`<space>`	:	the next screen is printed
`num<space>`	:	screen size is set to `num`, the next screen is printed
`<RETURN>`	:	the next line is printed
`num<RETURN>`	:	the next `num` lines are printed
`q`	:	the function currently producing this output is aborted
`h` or `?`	:	help is given

If `num` is zero, it will be ignored.

5 The Model of Massive Parallelism

The Parallaxis language model was designed to allow structured and machine-independent parallel programming in a high level language, similar to sequential Pascal or Modula-2 [Wirth 83]. Each Parallaxis program contains a functional description of the parallel machine architecture which it is about to use together with the parallel algorithm that refers to the architecture specification. This *virtual* specification is at a level of abstraction, of course, and is translated by a compiler to match any given physical architecture, even if it is only a single-CPU von-Neumann simulator. We restricted our flexible model to SIMD structures with identical processing elements (PEs).

An application program may specify number and dimensional arrangement of PEs as well as the interconnections between them. For this reason, all PEs have the same number of bi-directional ports, which may be connected to ports of other PEs or remain unconnected. Homogenous, scalable networks are preferred for clarity and are easy to specify in Parallaxis. However, arbitrary interconnection structures may be specified as well.

Corresponding to the machine model of a processor network controlled by a single, central host, variables may be declared either for the controlling host (using the declaration keyword `scalar`) or for each of the PEs (using `vector`). Each variable is strictly typed, like in Modula-2, and there may be no vector variables in an expression that is to be assigned to a scalar variable. Vectors may be used inside explicitly marked parallel blocks or operations only, while scalars may also appear in parallel vector expressions (requiring a duplication or *broadcast* of that particular value).

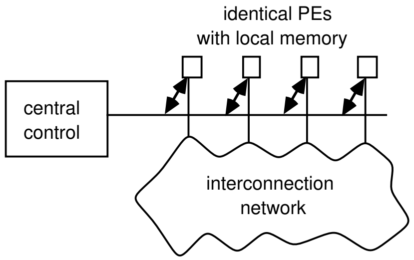


Figure 5.1: the abstract machine model

Parallaxis makes the following assumptions about its abstract parallel machine, whether it is physically present or simulated:

- SIMD structure with
  - central control unit
  - variable number of processing elements (PEs)
  - flexible (re-routable) communication network, that is it can be used as if forming any topological structure
- all operations occur synchronously
- all PEs are identical in processor and memory structure
- each PE has the same number of bi-directional ports for sending and receiving data
- one process per PE

Within the realm of this machine model, the programmer may now choose a certain topological structure for the communication network to achieve the best match possible between architecture and algorithm for a given problem. For each application, the number of PEs and the network topology is **semi-dynamic,** that is, all topologies used have to be specified prior to compilation, but the program may switch between them during execution.

Parallaxis provides a language construct for building parallel statement blocks. However, the control flow inside such a block is strictly sequential and may not be confused with the model of parallelism as used in Occam [Inmos 84]. Parallaxis executes each statement sequentially as seen from the host processor, sending (or **broadcasting**) that particular command to the selected set of PEs. According to SIMD-philosophy, for a large number of processors, each one performs the same computation on local data. There is no such thing like explicit programming of individual PEs as with MIMD-systems. Compared to MIMD-systems, there is no overhead for synchronization mechanisms, data locking, multi-process time-sharing operating system, and so on. However, not every algorithm is appropriate for the less flexible SIMD vectorization, but may be parallelizable for the more general MIMD-systems.

Parallaxis language constructs for explicit parallelism:

- `configuration / connection`
  specification of the desired network structure
- `parallel - endparallel`
  selection of processing elements for subsequent parallel processing
- `store / load / propagate / send / receive`
  data exchange to/from host, parallel data exchange among PEs
- `reduce`
  data reduction from vector to scalar
- `if / case / while / repeat / for`
  parallel branching and loops (the construct `loop` is always sequential)
- `x := e`
  vectorized assignment

Statements of a parallel block will be executed *chronologically sequential* but "spaced", that is *vectorized* for a number of processors. Following the keyword `parallel` may be an optional PE selection, which yields the same effect as a parallel branching statement. Here, only a subset of previously active processors perform the `then`-branch of an `if`-selection, while its complement set performs the `else`-branch. On real SIMD systems, these cannot be operated in parallel, so they have to be serialized by the Parallaxis system.

Data exchange among PEs using `propagate`, `send` or `receive` may only occur inside a parallel block, because here data exchange applies to groups of processors only, not to a pair of individual PEs. Which PEs participate in a data exchange operation (the so called "active" PEs) and which don't, is determined by its surrounding static and dynamic selections:

1. the comprising `parallel` block, and
2. all surrounding (and possibly nested) branching or loop constructs.

6 The Parallaxis Programming Language

6.1 Relation to Modula-2

Parallaxis was derived from Modula-2 by Thomas Bräunl [Bräunl 89d]. Language constructs identical to Modula-2 won't be explained here. Refer to the Modula-2 manual [Wirth 83] for further information. Parallaxis has the following data types:

- simple and real types
  - INTEGER
  - CARDINAL = [0 .. MAX(INTEGER)]
  - subrange
  - enumeration
  - CHAR
  - BOOLEAN = (FALSE, TRUE)
  - REAL
  - Pointer types: POINTER TO ...
- structured types
  - Arrays: ARRAY ... OF type
    a multi-dimensional array:
    ARRAY range_a, range_b OF type*is treated as*
    ARRAY range_a OF ARRAY range_b OF type
  - Records: RECORD ... END
  - Sets: SET OF stype
    Sets have no limitations in size and are treated as ARRAY stype OF BOOLEAN,
    for there are no bit-manipulating commands in PARZ.
  - Standard-Set: BITSET = SET OF [0 .. 31]

The following control structures from Modula-2 are implemented in Parallaxis:
(comments are also enclosed in "(*" and "*)")

- IF ... THEN ...{ ELSIF ...THEN ... }[ELSE ...] END
- CASE ... OF ... [ELSE ...]END
- WHILE ... DO ... END
- FOR ... TO ... [BY ...] DO ...END
- REPEAT ... UNTIL ...
- LOOP ... END
- WITH ... DO ... END

The conditions in all control structures may be scalar or vector expressions. However, `EXIT` and `RETURN` must not occur within a vector statement-block.

Parallaxis provides the following predefined constants:

- TRUE boolean constant for true
- FALSE boolean constant for false
- EOL char constant for end-of-line

- PI real constant (3.14159..)
- NIL nil pointer

and scalar read-only variables:

- termCH terminating character during a read-operation;
  it may be read only after a read-operation.
- Done boolean value set after an I/O-operation or a graphic operation
  signals success of an operation.

Concepts from Modula-2 that are **not** supported in the present implementation:

- module concept
- procedure types
- open array parameters

Features available in Parallaxis, but not in Modula-2 (see syntax in appendix A):

- multiple comparisons
  A boolean expression may consist of any number of comparisons.
  All of them have to be `TRUE` to make the whole expression `TRUE` .
  E.g.: `IF 1 < x < 100 THEN ...`
- power operator "`**`"
  The power operator has the highest priority and is right-associative.
  E.g.: `x := y**3.0;`
- constant arrays and records
  A structured constant may be expressed by its type-name followed by values
  (of appropriate type) in parentheses.
  E.g.:
```
TYPE field  = ARRAY [1..3] OF integer;
triple = RECORD x,y,z: real END;
...
a := field(1,2,3);
b := triple(2.0, PI, 6.1);
```

6.2 Specifying Processor Networks

Since in Parallaxis neither the topology nor the number of the processor elements is fixed, each program can freely choose its configuration; it will be transparently mapped to the physical hardware of the system. Two steps are necessary to specify the virtual form of the network structure: First, Parallaxis is told the number of processing elements one wants to use and how they will be arranged in dimensions (`CONFIGURATION` specification). This almost exactly takes the form of Modula-2's array declaration, with the distinction that we are dealing here with processors instead of data elements. Although we create some neighbor relation between the PEs by this declaration, this does not specify any connections between processors, which is reserved for the second step. There, we may specify a transfer-function (from a general PE position to its relative neighbor) for every processor-port in this topology (`CONNECTION` specification). Each transfer-function has a name (the exit-port's name) and also states the name of the corresponding entry port on its neighbor-PE after a period.

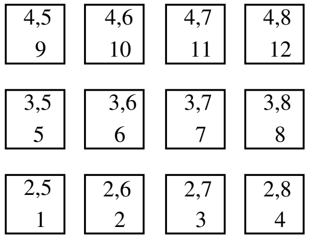
The arrangement of the processor elements can be a one or more dimensional array of PEs. A configuration specification states the configuration's name and a sequence of ranges, separated by commas. There are two possible types of ranges:

- `[ a .. b ]` forms a range from `a` to `b` (`a <= b`).
- `[ a ]` forms a range of `a` elements from `0` to `a-1` (`a > 0`).

Example 6.1:

```
CONFIGURATION example [ 2 .. 4 ], [ 5 .. 8 ];
```

defines a two-dimensional grid of 12 PEs in 3 rows by 4 columns



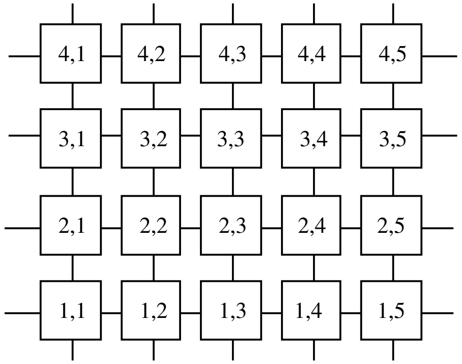
Every PE has a unique identification number. This number is in the range from 1 to the number of PEs specified and can be accessed in Parallaxis programs by the vector constant `id_no`. This number does *not* reflect the special topological arrangement of the processor elements, it is just a flat labelling. On a higher level, there are vector constants $\mathtt{DIM}_i$ to specify the position within the i-th logical dimension. The relation between `id_no` and $\mathtt{DIM}_i$ is explained in example 6.1 with an arrangement of twelve PEs in a two-dimensional array of processor elements. The lower number shows the `id_no`, while the upper numbers show `DIM1` and `DIM2`.

After the number and the topology of the processor elements are defined, the definitions of connections between the PEs follow the keyword `CONNECTION`. Each "transfer function" defines the connection of one port for **every** PE of this configuration.

Each transfer function begins with the output-port name, followed by a colon (see the complete syntax in appendix A). Now, a general source PE reference for the output direction is given in a form similar to an array access in Modula-2. There may be undeclared **free variables** inside a transfer function, defining this connection for each possible value according to the `CONFIGURATION` specification. Then follows a single arrow (`->`) denoting a one-way (unidirectional) connection or a double arrow (`<->`) denoting a two-way (bi-directional) connection. At last, the general destination PE is given in the same form as the source PE, again representing a subset of all declared PEs, followed by a period and the input-port name. While the dimensions of the source PE normally contain free variables (or constant values to build irregular topologies), the destination PE normally has an expression involving the free variables of the source PE in each dimension. Thus, a network topology is constructed by simply writing down the interconnection relation in functional form.

Example 6.2:

```
CONFIGURATION grid [1..4],[1..5];
CONNECTION    right: grid[i,j] -> grid [i  ,j+1].left;
              left : grid[i,j] -> grid [i  ,j-1].right;
              up   : grid[i,j] -> grid [i+1,j  ].down;
              down : grid[i,j] -> grid [i-1,j  ].up;
```



(* Defines a grid of 20 PEs with 4 ports/connections per PE,
connecting each PE to its 4 nearest neighbors.
The open endings at the left,right, up and down borders indicate
that all PEs have four ports, but some ports remain unconnected.*)

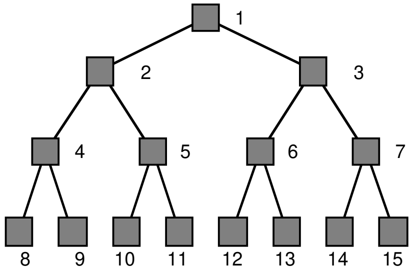
The input selection must have one expression for each dimension of the configuration.

There are three extensions to the basic form of topology specification:

- composed connections,
- parametrized connections and
- broadcasting connections.

Composed connections are essential for specifying complex topologies, like a tree structure (see example 6.3) or a perfect shuffle network. Each tree node PE has three ports: its father, its left son, and its right son. All nodes are arranged in a single dimension so that the left son of each PE has twice the position number while its right son has twice plus one. With this information, it is straightforward to specify the connections for both son-nodes, but with the father-connection we are in trouble, since we cannot tell whether a general PE is the left or the right son of a different PE.

Example 6.3: tree structure with composed transfer function



tree structure
(using a conditional function)

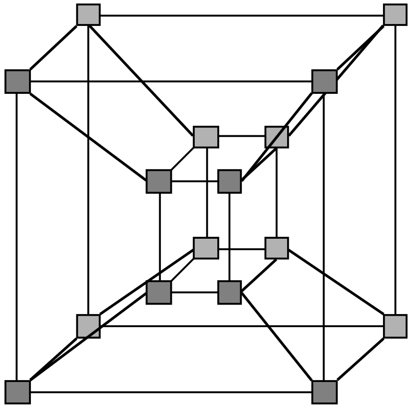
```
CONFIGURATION  tree [1..15];
CONNECTION     son_l : tree[i]  → tree[2*i].father;
               son_r : tree[i]  → tree[2*i + 1].father;
               father: tree[i]  → {even(i)} tree[i div 2].son_l,
                                  {odd(i)}  tree[i div 2].son_r;
```

We solve this problem with a composed connection, resembling composed mathematical functions. Boolean expressions (the *discriminants*) in curly brackets separate the cases from each other, separated by commas. In our tree topology, all even-numbered PEs are specified to be left sons while all odd PEs are right sons. With this topology specification we may propagate data both downward and upward the tree structure.

Composed connections are especially needed for complex topologies, in the case of our bi-directional binary tree, we could have also used the double arrow (↔) in the two son declarations to get away without defining the more difficult father relation.

Example 6.4: hypercube topology

```
CONFIGURATION  hyper [2] [2] [2] [2];
CONNECTION     go(1): hyper[i,j,k,l]    →
                      hyper[(i+1)mod 2,j,k,l].go(1);
               go(2): hyper[i,j,k,l]    →
                      hyper[i,(j+1)mod 2,k,l].go(2);
               go(3): hyper[i,j,k,l]    →
                      hyper[i,j,(k+1)mod 2,l].go(3);
               go(4): hyper[i,j,k,l]    →
                      hyper[i,j,k,(l+1)mod 2].go(4);
```



hypercube structure
(using parametrized functions)

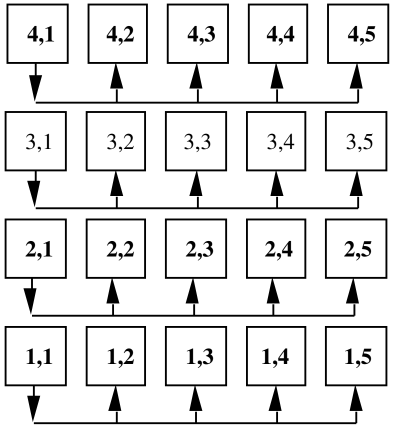
Parametrized connections do not increase the expressive power of the network specification as did the composed connections; they just simplify data exchange through highly symmetrical topologies, as the hypercube example 6.4.

Here, the port names are no proper identifiers, but parametrized names with single integer values in parentheses. Referencing directions by numbers may be much more convenient for data exchange in this case. A program can now perform a propagate operation with a computed direction (e.g. in scalar variable `i`) like the following:

```
propagate.go(i) (x);
```

Broadcasting connections are 1:n connections, that is a particular PE's output port is connected to several Pes' input ports. They do not increase the expressive power of the network specifications; but simplify data exchange for the broadcasting of values.

Example 6.5: grid structure with broadcasting transfer function



grid structure
(using broadcasting function)

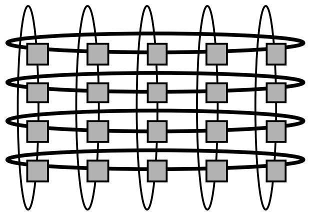
```
CONFIGURATION grid [1..4],[1..5];
CONNECTION    b_out: grid [i,1] -> grid [i,2..5].b_in;
```

The specified range must be a scalar constant range. Such a connection definition represents a list of connection definitions each with a different constant of the specified range.

Example 6.6: some more topologies and their specification in Parallaxis

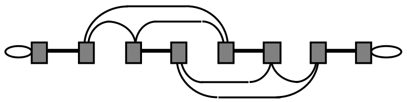
Torus

(2-dim. grid wrapped around)



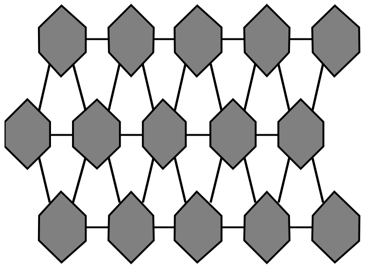
```
CONFIGURATION torus [0..3],[0..4];
CONNECTION
 right: torus[i,j]<->torus[i,(j+1)mod 5].left;
 up   : torus[i,j]<->torus[(i+1)mod 4,j].down;
```

Perfect Shuffle Network



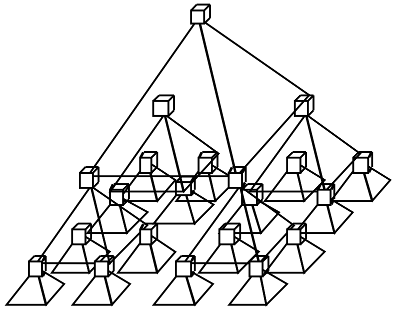
```
CONFIGURATION ps [8];
CONNECTION
 exch   :ps[i]->{even(i)} ps[i+1].exch,
                {odd(i)} ps[i-1].exch;
 shuffle:ps[i]->{i<4} ps[2*i].shuffle,
               {i>=4} ps[(2*i+1)mod 8].shuffle;
```

Hexagonal Mesh



```
CONFIGURATION hexa [3],[5];
CONNECTION
  right: hexa[i,j] <-> hexa[i  ,j+1].left;
  up_l : hexa[i,j] <-> hexa[i+1,j-i mod2].down_r;
  up_r: hexa[i,j] <-> hexa[i+1j+1-i mod2].down_l;
```

Quadtree



```
CONFIGURATION quad[0..84];
CONNECTION
   child1 : quad[i] ´ quad[4*i+1].parent;
   child2 : quad[i] ´ quad[4*i+2].parent;
   child3 : quad[i] ´ quad[4*i+3].parent;
```

Multiple Topologies

Parallaxis permits three forms of multiple topologies within a program. All three forms may be used in any combination.

a) Multiple disjointed topologies may be declared "**at different points in time**" in disjoint procedures. These different topologies cannot have any connections with each other, because

they are established at different times. This form of multiple topologies is similar to starting several Parallaxis programs after each other with a scalar data exchange. Procedures that have a topology definition must not be called from another procedure that also has a topology definition. A Parallaxis program with more than one topology will have as much PEs as the largest topology has.

Example 6.7:

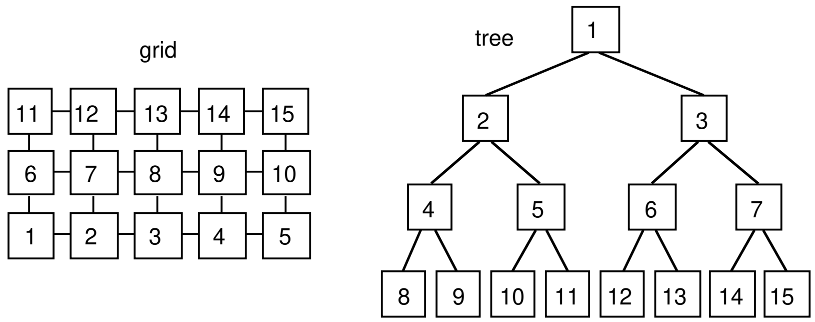
```
PROCEDURE ABC;
  CONFIGURATION ring [100];
  CONNECTION right: ring[i] <-> ring[i+1].left;
BEGIN ...
END ABC;

PROCEDURE XYZ;
  CONFIGURATION grid [10],[20];
  CONNECTION right: grid[i,j] <-> grid[i,j+1].left;
             up   : grid[i,j] <-> grid[i+1,j].down;
BEGIN ...
END XYZ;
```

b) Multiple overlay topologies may be declared **on the same set of PEs** in the same declaration part. With this form of multiple topologies, the programmer can solve different problems that need different topologies on the same set of PEs with the same data. This is like having "**different views**" of the same set of PEs. Therefore, tehy share the **same data space** and only a single vector declaration applies to all configurations. To declare multiple overlay topologies on the same PEs, the programmer has to declare each of the configurations after the same keyword `CONFIGURATION`.

The different topologies must have the same number of PEs, for they describe the same set of PEs. These topologies have the same ports and the same vector variables.

Example 6.8:

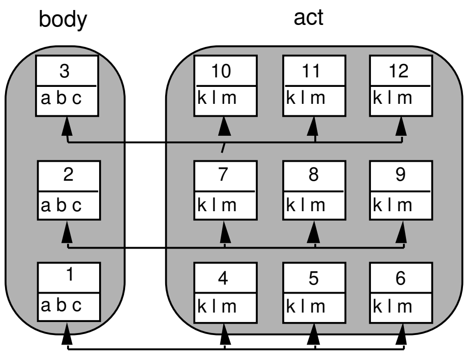
```
CONFIGURATION grid [5],[3];
              tree [1..15];
CONNECTION right: grid[i,j] <-> grid[i,j+1].left;
           up   : grid[i,j] <-> grid[i+1,j].down;
CONNECTION lchild: tree[i] <-> tree[2*i  ].parent;
           rchild: tree[i] <-> tree[2*i+1].parent;
```



c) Multiple disjointed topologies may be declared **on disjointed sets of PEs** to exist concurrently. For each topology, PEs have different ports and different vector variables. The PEs of each topology are continously labelled. All disjointed topologies are declared with separate keywords `CONFIGURATION`. All vector variables and all connections declared after the definition of a configuration are assigned to this previous topology. Each group of PEs has a **different data space** and therefore requires individual vector data declarations. Each connection definition must use at least one configuration name from the previous configuration declaration. The programmer may also specify connections between different configurations of the same declaration part.

<u>Example 6.9:</u>

```
CONFIGURATION body[1..n];
CONNECTION n: body[i] -> body[i+1].n;  (* Connection for body only *)
VECTOR a,b,c: INTEGER;                 (* Variables for body only  *)

CONFIGURATION act[1..n],[1..n];
CONNECTION r: act[i,j] -> act[i,j+1].r;  (* Connect. for act only  *)
           u: act[i,j] -> act[i+1,j].u;
           x: body <-> act[i,1..n].y; (* Con. between act and body *)
                       (* identical to: y: act[i,j] <-> body[i].x *)
VECTOR k,l,m: INTEGER;                (* Variables for act only    *)
```



6.3 Scalar and Vector Data

In sequential programming languages, all variables, parameters and function-results belong to one processor. In Parallaxis, however, the programmer has to tell the system whether a variable, parameter or function-result belongs to the scalar host or the parallel processor elements. This is done with the keywords `SCALAR` (for the control processor) and `VECTOR` (for the processor elements), replacing Modula-2's `VAR`, which is only used to mark call-by-reference parameters in Parallaxis. Each procedure parameter and a possible function-result have to be preceded by either `SCALAR` or `VECTOR` to identify whether these values belong to the host or the parallel PEs. The following rules apply for assigning values to variables or parameters:

- scalar values may be assigned to vector variables (duplicating scalar values), but
- a vector value mustn't be assigned to a scalar variable
- binary operations with at least one vector value will produce a vector-result
- there mustn't be a vector index to a scalar array
- two types are assignment compatible if
  - the types have the same name
  - both types are non-structured types and have the same definition
  - both types are subranges of the same type and the types have common elements
  - one type is a subrange of the other type
  - the assigned value is a string and the variable is a one-dimensional array of CHAR with at least (string_length + 1) elements
  - they are both pointer types and point to equal types. The constant NIL may be assigned to any pointer variable.
  - the standard types BOOLEAN, BITSET, CHAR, POINTER, INTEGER and REAL are incompatible. (CARDINAL is a subrange of INTEGER)

Vector variables that are declared in a declaration part which contains a topology defintion, can only be accessed within a parallel block, a `REDUCE` expression or a `LOAD` or `STORE` operation which use the corresponding topology. The special constants $\mathtt{DIM}_i$ can only be accessed, if the used topology can clearly be determined.

6.4 Vectorized Execution

Because of the distinction between control processor and parallel processor elements, the evaluation of expressions and the execution of statements has to be distinguished, too. Although the control processor controls the execution, there is a difference between the execution of statements which contain vector-components and the execution of statements without them. A statement without a vector-component is executed as in conventional programming languages, especially Modula-2. If all arguments are scalar, the operation is executed by the control processor, but if there is at least one vector argument, the operation is exectuted in parallel on the processor elements. To separate vectorized execution from normal execution, Parallaxis uses the keywords `PARALLEL` and `ENDPARALLEL` to embrace a parallel statement sequence. This block structure must not be nested and vector commands may only appear inside such a block. Outside a parallel block vector variables are only allowed within a `REDUCE` expression or a `LOAD` or `STORE` statement (cf. section 6.5 and 6.6). At the beginning of the parallel block it is possible to select the topology and activate a subset of PEs using a selection-expression after the keyword `PARALLEL`. To specify the used topology the PE selection may be headed by the name of the topology used. If there is more than one possible topology, the programmer must supply the configuration name.

A selection-expression is a list of activation expressions separated by commas. This list must contain as many activation-expressions as the processor configuration has dimensions. An activation-expression is a dimension-wide activation of processor elements within the processor arrangement and takes the following forms:

1. "[" const_int_expr [".." const_int_expr]
   { "," const_int_expr [".." const_int_expr] } "]"
   Static activation of all PEs inside these ranges
2. "[" "*" "]"
   Static activation of all PEs of that particular dimension
3. "[" bool_expr "]"
   Dynamic activation according to boolean vector expression
4. "[" int_expr "]"
   Dynamic activation according to integer vector expression (probably involving id_no or dim_i)
5. "[" int_expr ".." int_expr "]"
   Dynamic activation of a subrange of PEs of that particular dimension

The compiler can analyze the first two forms at compile time and produces a static activation in the PARZ-program for them. For formats 3 to 5, the compiler has to use dynamic PE activation. This is done by replacing the selection construct by "`[*]`" to select all PEs, and branching with vector conditions at run-time. The following example shows the translation of a dynamic activation:

Example 6.10: Dynamic PE selection

```
CONFIGURATION example [10],[10];
CONNECTION (* ... *);
SCALAR x,y : INTEGER;
...
PARALLEL [1..3], [x .. y];
   (* statements *)
ENDPARALLEL
```

will be transformed into:

```
PARALLEL [1..3],[*];
  IF x <= dim2 <= y THEN
    (* statements *)
  END
ENDPARALLEL
```


Control flow for branchings and loops with vector expressions differs a lot from their execution with scalar expressions. While a scalar branching executes exactly one branch, several branches might have to be executed for vector branchings, since the local results of a vector expression may be different for each PE. Each processor element is active in at most a single branch, but the control processor has to execute all branches with at least one active PE. Branches with no active processor element will be skipped.

Loops with vector control expressions will be executed until no processor element is active any more. The control expression is evaluated each time for all PEs that were active at the beginning of the loop. Thus, during each pass through the loop, some PEs may be switched off until termination of that loop pass. They may become active again, however, during the next pass through the loop, as shown in example 6.11.

Example 6.11: Alternating activation of processor elements in a loop

```
 (* at least 2 PEs *)
x := 0;
WHILE EVEN(id_no + x) DO
   (* loop body *)
   x := (x + 1) MOD 2
END;
```

6.5 Parallel Data Exchange

The following statements are used to exchange data between the control processor and the parallel PEs (`LOAD` and `STORE`), and among the PEs itself (`PROPAGATE`):

- **LOAD** "(" vector_variable "**,**" scalar_variable ")"
  puts the data of a scalar array (of appropriate size) in a vector variable, e.g.
  ```
  CONFIGURATION topo [1..20]; ...
  SCALAR s : ARRAY [1..20] OF INTEGER;
  VECTOR v : INTEGER;
  ...
  LOAD(v,s);
  ```
- **STORE** "(" vector_variable "**,**" scalar_variable ")"
  puts the data of a vector variable into a scalar array (of appropriate size), e.g.:
  ```
  CONFIGURATION topo [1..20]; ...
  SCALAR s : ARRAY [1..20] OF INTEGER;
  VECTOR v : INTEGER;
  ...
  STORE(v,s);
  ```

In `LOAD` and `STORE`, the scalar variable has to be an array of the same data type as the vector variable. It is also possible to `LOAD/STORE` structured variables. The scalar array has to be big enough to send or receive all data transferred to or from the activated PEs. If data from a single PE is transferred, the scalar variable doesn't have to be an array.

`LOAD/STORE` may be used outside a parallel block with an explicit selection (if necessary), or inside a parallel block without an explicit selection. In the latter case only the data of/for active PEs is being transferred and placed into/from the next consecutive slot of the scalar array.

There are three possibilities to transfer fewer data elements (see appendix A for details):

a) Select a subset of all PEs for the load/store operation, in analogy to the selection for a parallel block. E.g.:

```
LOAD [1..5],[3..7] (v,s);
```

b) Use a scalar length argument as third parameter in a load/store operation. If the length argument is given, the size of the scalar variable will not be checked. After the transmission, the actual number of transferred data elements is stored in the length argument (a VAR parameter). The length designator has to be a scalar variable of a data type that includes the range from 0 to the maximum of activated PEs. E.g.:

```
max_length := 100;
LOAD (v,s, max_length);
```

c) LOAD/STORE may also be used inside a parallel block without explicit PE-range-selection (but with specifying the configuration name in case there are multiple topologies in a program). In this case, only the data of/for active PEs is being transferred and placed into/from the next consecutive slot of the scalar array. E.g.:

```
PARALLEL [ odd(id_no) ]
  LOAD (v,s);
ENDPARALLEL;
```

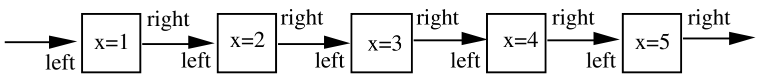
- **PROPAGATE** "." direction "(" vector_variable ")"
  shifts the local components of a vector variable through the output-port *direction* into the corresponding input-port of a neighbor PE, according to the definition of the network topology (CONNECTION specification), e.g (see example 6.12):

```
CONFIGURATION list [1..5];
CONNECTION right: list[i] <-> list[i+1].left;
VECTOR x,y : INTEGER;
...
PROPAGATE.right(x);
```

The vector variables in this parallel data exchange mustn't be a structured data type due to implementation restrictions in PARZ, but it may be a component of a structured variable. Only active PEs participate in the parallel data exchange (selected explicitly for the parallel block or implicitly by branchings or loops with vector expressions).

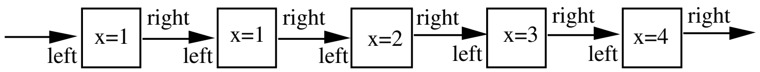
Example 6.12: propagating data through a linear list

Before the data exchange:



PROPAGATE.right(x)

After the data exchange:



The PROPAGATE-statement is, in contrast to the LOAD- and STORE-statement, a true parallel statement and, therefore, has to be used within a PARALLEL_ENDPARALLEL block. For propagating data through the network, we have to provide:

- the description of the shift-direction (separated by a period).
- the description of the shifted variable (as an argument in parentheses).

In order to define the shift-direction, we simply use a symbolic direction name as defined in the CONNECTION specification. So the data exchange statement: `propagate.right(x);` would send the component-value of x on each active PE one step to its right neighbor in a linear list topology, as shown in detail in example 6.12.

Supplying only the send-direction is sufficient for unambiguous topologies like a grid, since the compiler is able to derive the corresponding receive-direction from the CONNECTION specification. In the case of the above list, left would be automatically determined to be the corresponding receive-port for right. However, this does not hold for all topologies! So it is *not* sufficient for a tree topology, as explained in section 6.2. In this case, the send-direction *and* the receive direction have to be appended (with periods). E.g.:

```
propagate.father.son_l (x);
```

is a correct data exchange up the tree from each left son to his father. The data of all "right sons" will be discarded.

If there is no connection to an output-port for a PE, or if the corresponding processor element is not active, the send-value will not be received by any PE. This data element is literally being pushed "over the edge". In case there is no connection to the input-port of a PE, or the corresponding processor element is not activated, then this PE will not receive a new value from anywhere after sending its data element, so the propagated variable keeps its original value for this particular PE. A shift of *n* steps in an open topology will transmit the value from the border PE to the next *n* processor elements. This effect is demonstrated in example 6.13.

There are three extensions to the basic form of data propagation:

1. *propagating data several steps at once*
A scalar multiplier is added after the send-direction with the caret-character ("^").

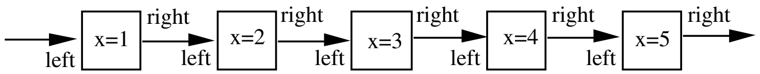
Example 6.13: multiple data propagation for the list from example 6.12

```
PROPAGATE.right^2(x);
```

is equivalent to:

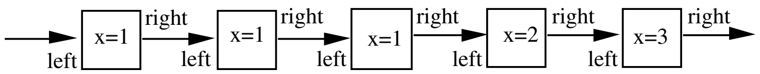
```
FOR i := 1 TO 2 DO
   PROPAGATE.right(x);
END;
```

Before the data exchange:




```
PROPAGATE.right^2(x);
```

After the data exchange:



2. *propagating data in computed directions*
For parametrized directions, as described in section 6.2, propagate operations may be used with computed directions.
E.g. for a hypercube:

```
i := 3*j - 2;
PROPAGATE.dir(i) (x);
```

3. *receiving data in a second argument (preserving the send-value)*
This form of the `PROPAGATE` statement has two arguments, the first being an expression that will be sent, the second being a variable which will receive the new value.
E.g. for a two-dimensional grid:

```
PROPAGATE.right (x,y);
```

is equivalent to:

```
y:=x;
PROPAGATE.right (y);
```

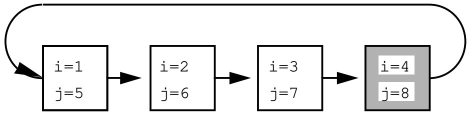
The previouly declared `PROPAGATE` operations suppose that there are only *1:1* connections. As Parallaxis provides the facility of *m:n* connections there may be a collision when two or more PEs are sending a value to the same PE. This conflict may freely be solved by specifying a `REDUCE` operator to handle the conflict. This `REDUCE` operator returns a single result combining all incoming values.The conflict solution with a `REDUCE` operator is defined by appending the keyword `REDUCE` followed by a period '`.`' and the name of the operator to the `PROPAGATE` statement. The standard `REDUCE` operators and how to define an own reduce operator is declared in the following section 6.6. If a collision is detected and no `REDUCE` operator is specified, the PARZ simulator will produce a run-time error.

As the `PROPAGATE` operation can only deal with active sending and active receiving PEs, there is no way of data exchange between different topologies on different sets of PEs. Therefore, the data exchange between different topologies would only be possible through the control processor using `STORE` and `LOAD`. This is not acceptable and therefore there are two data exchange statements which can handle idle PEs.

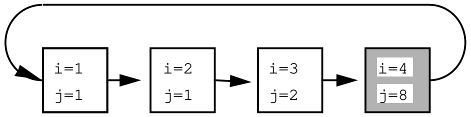
- **SEND** out_port "(" vector_expr ")" **TO** in_port "(" vector_ident ")"
  All active PEs will send the result of the vector expression through the specified out_port to all other PEs. All PEs, including the idle ones, look at their in_port if a value is received. If a value has reached the in_port the value is assigned to the vector identifier. A `REDUCE` operator may be added to handle a collision of values at the in_port in the same way as in the `PROPAGATE` statement. The out_port must belong to the active topology.

- **RECEIVE** in_port "(" vector_variable ")" **FROM** out_port "(" vector_ident ")"
  All PEs will send the value stored in the vector identifier through the given out_port. The active PEs will look at their in_ports for an incoming value. If a value is received the value is assigned to the vector variable. As for the `PROPAGATE` and the `SEND` statement there may be a `REDUCE` operator added to the statement. The in_port must belong to the active topology.

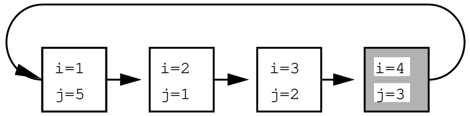
The different semantics of the three vector-vector data exchange statements for data exchange within one topology is shown in the following picture. The dark PE is assumed to be inactive. All examples start with the initial data settings of the first picture.



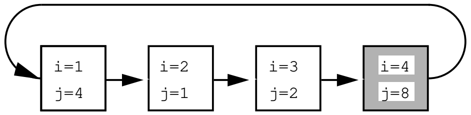

```
PROPAGATE.ring_out(i,j);
```




```
SEND ring.ring_out(i) TO ring.ring_in(j);
```




```
REICEIVE ring.ring_in(j) FROM ring.ring_out(i);
```



6.6 Data Reduction

There are a lot of problems that can be solved in parallel by using the processor elements to calculate components of the result. These partial results have to be combined somehow to give the final result (e.g. numeric integration). Completing this task for n components would take n–1 sequential operations for the controlling host to compute. Using the PEs themselves in a parallel tree-like fashion reduces the amount of time needed to $\lceil ld(n) \rceil$. This is the motivation for providing the `REDUCE` construct. In each reduction step, the number of values is divided by two, until only a single value remains. This final result is transferred as a scalar to the host. For `REDUCE` always transforms a vector into a scalar value, it mustn't be applied to structured data types. Parallaxis provides the following reduction operators as predefined functions:

- MAX — maximum of all values
- MIN — minimum of all values
- FIRST — value at the active PE with the lowest PE identification-number (id_no)
- LAST — value at the active PE with the highest PE identification-number (id_no)
- AND — logic "and" operation of all values (type boolean only)
- OR — logic "or" operation of all values (type boolean only)
- PRODUCT — product of all values (numerical types only)
- SUM — sum of all values (numerical types only)
  (e.g. the number of active processors is: `REDUCE.SUM(1)`)

The use of the `REDUCE` construct is not limited to the predefined operators mentioned above. A reduction function may be defined as a function with two arguments and a result, all of the same type. The function mustn't contain data-exchange operations or the `REDUCE` operator itself, neither direct nor indirect (within a called function). The performed operation should be an associative and commutative binary operation to prevent strange results. A function that is used as a `REDUCE` operator may also be used as a regular function. Example 6.14 shows a `REDUCE`-operator for adding numbers modulo 12.

Example 6.14:

```
SCALAR num: cardinal;

PROCEDURE add_mod(VECTOR a,b: cardinal): VECTOR cardinal;
BEGIN
  RETURN (a + b) MOD 12;
END add_mod;
...
num := REDUCE.add_mod (1);
```


6.7 Standard Procedures

In the following holds:

ordinal	denotes an enumeration-, subrange- or integer type
elem	denotes the base type for a set
settype	denotes the type `SET OF elem`
string	denotes a variable of type `ARRAY .. OF CHAR` or a constant string in quotes or apostrophes
pointertype	denotes the type `POINTER TO elemtype`

- DEC (SCALAR VAR x : ordinal)
  DEC (VECTOR VAR x : ordinal)
  decrement variable x by 1 or set the variable to predecessor of x.
- DEC (SCALAR VAR x : ordinal; SCALAR n : INTEGER)
  DEC (VECTOR VAR x : ordinal; VECTOR n : INTEGER)
  decrement variable x by n or set the variable to the n-th predecessor of x.
- INC (SCALAR VAR x : ordinal)
  INC (VECTOR VAR x : ordinal)
  increment variable x by 1 or set the variable to successor of x.
- INC (SCALAR VAR x : ordinal; SCALAR n : INTEGER)
  INC (VECTOR VAR x : ordinal; VECTOR n : INTEGER)
  increment variable x by n or set the variable to the n-th successor of x.
- EXCL (SCALAR VAR x : settype; SCALAR y : elem)
  EXCL (VECTOR VAR x : settype; VECTOR y : elem)
  remove element y from set x.
- INCL (SCALAR VAR x : settype; SCALAR y : elem)
  INCL (VECTOR VAR x : settype; VECTOR y : elem)
  put element y into set x.
- NEW (SCALAR VAR x: pointertype)
  NEW (VECTOR VAR x: pointertype)
  allocate memory for one element of type elemtype on the heap and assign the memory address to parameter x.
- DISPOSE (SCALAR x: pointertype)
  DISPOSE (VECTOR x: pointertype)
  free memory on the heap to which parameter x points.
- Write (SCALAR c : CHAR)
  Write (VECTOR c : CHAR)
  write character c to the active output channel.
- WriteInt (SCALAR i : INTEGER; SCALAR n : CARDINAL)
  WriteInt (VECTOR i : INTEGER; SCALAR n : CARDINAL)
  write integer number i with a width of n characters to the active output channel.
- WriteCard (SCALAR i, n : CARDINAL)
  WriteCard (VECTOR i: CARDINAL; SCALAR n : CARDINAL)
  write cardinal number i with a width of n characters to the active output channel.

- WriteReal (SCALAR r : REAL; SCALAR n : CARDINAL)
  WriteReal (VECTOR r : REAL; SCALAR n : CARDINAL)
  write floating-point number r with a width of n characters to the active output channel.
- WriteFixPt (SCALAR r : REAL; SCALAR n, p : CARDINAL)
  WriteFixPt (VECTOR r : REAL; SCALAR n, p : CARDINAL)
  write floating-point number r with a width of n characters and p decimals to the active output-channel.
- WriteBool (SCALAR b : BOOLEAN)
  WriteBool (VECTOR b : BOOLEAN)
  write boolean value in b to the active output-channel.
- WriteString (SCALAR s : string)
  WriteString (VECTOR s : string)
  write string s to the active output-channel. Output stops at the first occurence of chr(0) or after writing all characters of a char array.
- WriteLn
  write the end-of-line character to the active output-channel.
- Read (SCALAR VAR c : CHAR)
  Read (VECTOR VAR c : CHAR)
  read a character from the active input-channel into variable c.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- ReadInt (SCALAR VAR i : INTEGER)
  ReadInt (VECTOR VAR i : INTEGER)
  read a number from the active input-channel into variable i.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- ReadCard (SCALAR VAR c : CARDINAL)
  ReadCard (VECTOR VAR c : CARDINAL)
  read a number from the active input-channel into variable c. A run-time-error will occur if the number read is negative. This range-check **cannot** be disabled with option -r.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- ReadReal (SCALAR VAR r : REAL)
  ReadReal (VECTOR VAR r : REAL)
  read a floating-point number from the active input-channel into variable r.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- ReadBool (SCALAR VAR b : BOOLEAN)
  ReadBool (VECTOR VAR b : BOOLEAN)
  read a logical value from the active input-channel into variable b.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- ReadString (SCALAR VAR s : string)
  ReadString (VECTOR VAR s : string)
  read a string from the active input-channel into variable s.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- OpenInput (SCALAR s : string)
  open the file with name s as new input channel. If s is an empty string, the system will interactively prompt for a filename. The previous input channel is closed. If the file cannot be opened for input, standard-input becomes the active input channel.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).

- OpenOutput (SCALAR s : string)
  open the file with name s as new output channel. If s is an empty string, the system will interactively prompt for the filename. The previous output channel is closed. If the file cannot be opened for output, standard-output becomes the active output channel.
  Read-only-variable "Done" is set appropriately (cf. Modula-2).
- CloseInput
  close the active input channel. Standard input becomes active input channel.
- CloseOutput
  close the active output channel. Standard output becomes the active output channel.
- DEBUG (...)
  produce PARZ debug commands for all variables, supplied as parameters.
- TRACE (...)
  produce PARZ trace commands for all variables, supplied as parameters. Tracing vector arrays with vector indices isn't allowed. The values of the variables are printed when executing this command and every time an assignment is performed on them.
- NOTRACE (...)
  NOTRACE
  switch tracing mode off for all variables supplied as parameters, or for every variable (without arguments). It is not checked, whether a variable is currently being traced.
- HALT
  stop program execution.

Parallel write operations to a terminal device result in the PE values printed in their consecutive order. Characters appear on the same line, all other types require one line per PE. Parallel read operations from a terminal device read the input data and distribute it consecutively to the PEs in the order of their declaration. Changing between formatted (e.g. integers) and unformatted input (e.g. single characters) may result in left-over carriage returns that have to be read by a sequential read operation.

6.8 Standard Functions

In the following holds:
TYPE name denotes a type identifier.
PORT name denotes a port identifier.

- Cos (SCALAR r : REAL) : SCALAR REAL
  Cos (VECTOR r : REAL) : VECTOR REAL
  returns cos(r).
- Sin (SCALAR r : REAL) : SCALAR REAL
  Sin (VECTOR r : REAL) : VECTOR REAL
  returns sin(r).
- Tan (SCALAR r : REAL) : SCALAR REAL
  Tan (VECTOR r : REAL) : VECTOR REAL
  returns tan(r).
- ArcCos (SCALAR r : REAL) : SCALAR REAL
  ArcCos (VECTOR r : REAL) : VECTOR REAL
  returns arccos(r).

- ArcSin (SCALAR r : REAL) : SCALAR REAL
  ArcSin (VECTOR r : REAL) : VECTOR REAL
  returns arcsin(r).
- ArcTan (SCALAR r : REAL) : SCALAR REAL
  ArcTan (VECTOR r : REAL) : VECTOR REAL
  returns arctan(r) in the range of - /2 - + /2.
- ArcTan2 (SCALAR r1, r2 : REAL) : SCALAR REAL
  ArcTan2 (VECTOR r1, r2 : REAL) : VECTOR REAL
  returns p=arctan(r1/r2) in the unambiguous range of -pi - +pi.
  The signs of r1 and r2 are equivalent to the signs of sin(p) and cos(p).
- Exp (SCALAR r : REAL) : SCALAR REAL
  Exp (VECTOR r : REAL) : VECTOR REAL
  returns the exponential function e^r.
- Ln (SCALAR r : REAL) : SCALAR REAL
  Ln (VECTOR r : REAL) : VECTOR REAL
  returns the natural logarithm ln(r).
- Sqrt (SCALAR r : REAL) : SCALAR REAL
  Sqrt (VECTOR r : REAL) : VECTOR REAL
  returns the square root R(r) .
- ABS (SCALAR x : INTEGER) : SCALAR CARDINAL
  ABS (VECTOR x : INTEGER) : VECTOR CARDINAL
  ABS (SCALAR x : REAL) : SCALAR REAL
  ABS (VECTOR x : REAL) : VECTOR REAL
  returns the absolute value $|x|$.
- FLOAT (SCALAR i : INTEGER) : SCALAR REAL
  FLOAT (VECTOR i : INTEGER) : VECTOR REAL
  converts the integer argument to a real number.
- TRUNC (SCALAR r : REAL) : SCALAR INTEGER
  TRUNC (VECTOR r : REAL) : VECTOR INTEGER
  truncates a real number argument to an integer.
- CAP (SCALAR c : CHAR) : SCALAR CHAR
  CAP (VECTOR c : CHAR) : VECTOR CHAR
  converts a lower-case character to a capital letter; other characters remain unchanged.
- CHR (SCALAR i : INTEGER) : SCALAR CHAR
  CHR (VECTOR i : INTEGER) : VECTOR CHAR
  returns the character corresponding to ordinal number i.
- ORD (SCALAR x : ordinal) : SCALAR INTEGER
  ORD (VECTOR x : ordinal) : VECTOR INTEGER
  ORD (SCALAR x : CHAR) : SCALAR INTEGER
  ORD (VECTOR x : CHAR) : VECTOR INTEGER
  returns the ordinal number corresponding to the argument.
- EVEN (SCALAR i : INTEGER) : SCALAR BOOLEAN
  EVEN (VECTOR i : INTEGER) : VECTOR BOOLEAN
  returns `TRUE` if i is even, else `FALSE`.
- ODD (SCALAR i : INTEGER) : SCALAR BOOLEAN
  ODD (VECTOR i : INTEGER) : VECTOR BOOLEAN
  returns `TRUE` if i is odd, else `FALSE`.

- MAX (TYPE ordinal) : SCALAR ordinal
  returns the largest element of type ordinal.
- MIN (TYPE ordinal) : SCALAR ordinal
  returns the smallest element of type ordinal.
- SIZE (SCALAR x : type) : SCALAR INTEGER
  SIZE (VECTOR x : type) : SCALAR INTEGER
  returns the size of memory space used by variable x. This value is constant, but cannot be calculated at compile time, since it is implementation-dependent.
- STRCMP (SCALAR s1, s2 : string) : SCALAR INTEGER
  STRCMP (VECTOR s1, s2 : string) : VECTOR INTEGER
  compares the strings s1 and s2. Result is +1, 0, -1, depending on whether s1 is lexicographically greater than, equal, or less than s2. Strings must be terminated with chr(0).
- STREQ (SCALAR s1, s2 : string) : SCALAR BOOLEAN
  STREQ (VECTOR s1, s2 : string) : VECTOR BOOLEAN
  compares the strings s1 and s2. STREQ(s1,s2) ≡ STRCMP(s1,s2) = 0.
- SBRandom () : SCALAR BOOLEAN
  BRandom () : VECTOR BOOLEAN
  returns a random boolean value (vector: one for each PE).
- SCRandom () : SCALAR CHAR
  VCRandom () : VECTOR CHAR
  returns a random character (vector: one for each PE).
- SIRandom () : SCALAR INTEGER
  VIRandom () : VECTOR INTEGER
  returns a random integer (vector: one for each PE).
- SRRandom () : SCALAR REAL
  VRRandom () : VECTOR REAL
  returns a random real number (vector: one for each PE).
- VAL (TYPE ordinal; SCALAR n : INTEGER) : SCALAR ordinal
  VAL (TYPE CHAR; SCALAR n : INTEGER) : SCALAR CHAR
  VAL (TYPE ordinal; VECTOR n : INTEGER) : VECTOR ordinal
  VAL (TYPE CHAR; VECTOR n : INTEGER) : VECTOR CHAR
  reverse function to ORD: returns the element or character corresponding to that ordinal number. It holds: ORD(VAL(ordinal, n)) = n and VAL(CHAR, n) = CHR(n).
- IN_Connected (PORT in_port) : VECTOR CARDINAL
  returns the number of connections to in_port of the PE.
- OUT_Connected (PORT out_port) : VECTOR CARDINAL
  returns the number of connections from out_port of the PE.
- IN_Lineconnected (PORT out_port, in_port) : VECTOR CARDINAL
  returns the number of connections from an out_port to in_port of the PE.
- OUT_Lineconnected (PORT out_port, in_port) : VECTOR CARDINAL
  returns the number of connections from out_port of the PE to an in_port.

Operators in Expressions
(in order of priority)

unary operators:	`NOT`	`+`	`-`					
power:	`**`							
multiplication:	`*`	`/`	`DIV`	`MOD`	`AND`	`&`		
addition:	`+`	`-`	`OR`					
relation:	`=`	`<>`	`#`	`<`	`<=`	`>`	`>=`	`IN`

`NOT` is an unary operator, "`+`" and "`-`" may be unary or binary, all other operators are binary.
Power "`**`" is right-associative, while all other operators are left-associative.
`DIV` is the integer divison, `MOD` the integer remainder, while "`/`" denotes a real-number division.
Boolean *and* is expressed by either `AND` or "`&`". Not equal is expressed by either "`<>`" or "`#`".

6.9 Graphics Interface

Parallaxis provides a set of primitive procedures and functions to get a machine independent graphics interface. Although these procedures build a machine-independent graphics interface, the available graphics features depend on the machine used.

> PLEASE NOTE: The Graphics Interface is currently only implemented for workstions with X-windows (not yet for Macintosh and PC systems).

To handle the colors a new data structure is automatically declared:

```
TYPE COLOR = RECORD
               RED, GREEN, BLUE : CARDINAL;
             END;
```

The components of this record hold the intensity of the corresponding colors. All functions and procedures set the read-only-variable `Done` according to the success of the function or procedure.

- OpenWindow (SCALAR x,y : REAL) : SCALAR INTEGER
  opens a new window on the screen with the **relative** measure x and y (relative to the absolute screen coordinates). The result is the window selector which is needed to activate or to close the window. The new window will automatically be activated when it is opened. If the new window could not be opened, the previously active window will stay active. You have to **click the mouse button** with the cursor in the newly opened window, in order to **proceed**!

- OpenAbsWindow (SCALAR x,y : INTEGER) : SCALAR INTEGER
  opens a new window on the screen with the **absolute** measure of x pixels by y pixels (if it fits onto the screen).The result is the window selector which is needed to activate or to close the window. The new window will automatically be activated when it is opened. If the new window could not be opened, the previously active window will stay active. You have to **click the mouse button** with the cursor in the newly opened window, in order to **proceed**!

- CloseWindow (SCALAR handle : INTEGER)
  closes the window with the window selector handle. After execution, no window will be active, so another window has to be selected before the next graphics operation.

- WindowSize (SCALAR VAR x,y : INTEGER)
  returns the parameters x and y with the measure of the active window in pixels.

- SelectWindow (SCALAR handle : INTEGER)
  activates the window with the window selector handle for subsequent graphics commands.

- SetColor (SCALAR c : COLOR)
  SetColor (VECTOR c : COLOR)
  sets the actual painting color to the values contained in the record c. The real color depends on the hardware facilities of the system used (24 bit true color systems **are** supported).

- SetPixel (SCALAR x,y : INTEGER)
  SetPixel (VECTOR x,y : INTEGER)
  sets the point with the pixel coordinates (x,y) to the previously defined color. If SetPixel is called with vector arguments and the previous call of SetColor has been with a scalar argument, then all pixels will be displayed in the same color.

- GetPixel (SCALAR x,y : INTEGER) : SCALAR COLOR
  returns the color of the point with the pixel coordinates (x,y).

- Line (SCALAR x1,y1, x2,y2 : INTEGER)
  draws a line using the actual color from the point with the pixel coordinates (x1,y1) to the point with the pixel coordinates (x2,y2).

- MoveTo (SCALAR x,y : INTEGER)
  defines the active position at the pixel with coordinates (x,y) for subsequent drawing routines.

- Draw (SCALAR c : INTEGER)
  writes character c at the active position in the previously selected window.

- DrawInt (SCALAR i : INTEGER; SCALAR n : CARDINAL)
  writes integer number i with a width of n characters at the active position in the previously selected window.

- DrawCard (SCALAR i : INTEGER; SCALAR n : CARDINAL)
  writes cardinal number i with a width of n characters at the active position in the previously selected window.

- DrawReal (SCALAR r : REAL; SCALAR n : CARDINAL)
  writes floating point number r with a width of n characters at the active position in the previously selected window.

- DrawFixPt (SCALAR r : REAL; SCALAR n,p : CARDINAL)
  writes floating point number r with a width of n characters and p decimals at the active position in the previously selected window.

- DrawBool (SCALAR b : INTEGER)
  writes boolean value b at the active position in the previously selected window.

- DrawString (SCALAR s : string)
  writes character string s at the active position in the previously selected window.

On Unix workstations, the font of the drawing routines may be influenced by setting the environment variable PARALLAXIS_FONT to a valid X-Window string. The construction of such a string is described in the X-window documentation.

E.g.: `$ setenv PARALLAXIS_FONT "-adobe-symbol-*-*-*-*-20-*-*-*-*-*-*-*"`
defines that all output is made using the adobe-font "symbol" in a size of 20 pixels.

7 The PARZ Intermediate Language

PARZ is a parallel intermediate language. It is a machine-independent pseudo-assembler suitable to code massively parallel programs. PARZ programs are normally generated by the Parallaxis Compiler. For the PARZ language is readable as well, it is also possible to write PARZ programs manually.

A PARZ-program consists of:

- the keyword `START`,
- the number of processors (PEs) and ports per PE,
- the declaration of global variables (for control-processor and for the parallel PEs),
- a list of commands and
- the keyword `STOP`.

The formal syntax of PARZ can be found in Appendix B.

7.1 Specifying Processors and Connections

Unlike in Parallaxis, the processors in PARZ are linearly arranged and numbered consecutively, starting with `1`. To define the structure of the processor-array, the number of processor elements (PEs) and the number of ports on each PE are stated in each program after the keyword START.

Connections between PEs are established dynamically during program execution.

7.2 Data Management

To support the concept of nested procedures in Parallaxis, PARZ provides mechanisms for static and dynamic chains. Before starting the main program or a procedure, memory chunks for the control processor and the PEs are dynamically allocated. The structure of the required memory blocks is defined by declarations preceding the assembly-command part of the program, or by declarations in the procedures. Declarations for the control processor start with the keyword `SCALAR`, while `VECTOR` is used for the processor array. The keywords `SCALAR` and `VECTOR` of the global declarations have to be present in each program; declarations for local procedure variables may be omitted. The elementary data types are boolean, char, integer or real, which is defined in the declaration.

The keyword `SCALAR` or `VECTOR` of a declaration is followed by a possibly empty list of subdeclarations. This may be:

1) One of the characters B, C, I or R representing type boolean, char, integer or real. This character is followed by a positive integer number *n*. Such a declaration describes a block of length *n* with variables of the same defined type.

2) A positive integer number *m* followed by a list of sub-declarations enclosed in parentheses. This is an abbreviation for *m* repetitions of that list.

3) The character 'U' ("union") followed by a comma separated list of declarations, enclosed in parentheses. This is a variant declaration. Each declaration in the list uses the same memory locations.

Example 7.1: variable declaration

```
SCALAR I5 2( C3 2( I1 B1 ) R4 )
VECTOR I1 B1 U ( R3, I1 2( B1 C3 ) )
```

the above declarations are equivalent to:

```
SCALAR I5  C3  I1 B1  I1 B1  R4   C3  I1 B1  I1 B1  R4
VECTOR I1 B1 U ( R3, I1  B1 C3  B1 C3 )
```

and reflect the Parallaxis declaration:

```
SCALAR a, b, c, d, e : integer;
       f : ARRAY [ 1 .. 2 ] OF
             RECORD
               a, b, c : char;
               d : ARRAY boolean of RECORD i : integer;
                                           b : boolean;
                                    END;
               e, f, g, h : real;
             END;
VECTOR r : RECORD i : INTEGER;
             CASE b : BOOLEAN OF
               TRUE  : x,y,z : REAL |
               FALSE : j     : INTEGER;
                       a     : ARRAY [2] OF
                                 RECORD b : BOOLEAN;
                                        c,d,e : CHAR
                                 END
             END
           END;
```

7.3 Variables and Constants

Arguments of PARZ commands may be:

- simple variables
- indirect variables
- predefined variables for special purposes
- variables with address operator
- constants

Simple Variables

Variables defined in declarations can be identified by a variable description consisting of the following four parts:

1.) One of the characters S or V to distinguish between variables for the control processor (S = scalar) and variables for the PE-array (V = vector).

2.) One of the characters B, C, I or R defining the type of the variable.

3.) An integer >= 0 giving the nesting level of the procedure declaring the variable. Global variables have nesting level 0.

4.) An integer > 0 giving the variable number. The variables defined in a declaration are num bered separately for each type (starting from 1).

Components 3) and 4) are seperated by a colon.
Components of variants are numbered as if the character 'U', the parantheses and the commas were missing.

Example 7.2: variable access

(1)
```
SCALAR I2 R1 I1
VECTOR B2 I1
```
defines the variables
`SI0:1, SI0:2, SR0:1, SI0:3, VB0:1, VB0:2 and VI0:1`

(2)
```
PROC 3 VECTOR I1 U( I2, R1 I1, 2( U( C1, I2 ) ) )
```
defines the local procedure variables
```
VI3:1    VI3:2    VI3:3
         VR3:1    VI3:4
         VC3:1             VC3:2
         VI3:5    VI3:6    VI3:7    VI3:8
```
where vertically aligned variables occupy the same memory location!

Indirect Variables

Two kinds of indirection are possible in PARZ: In *locally* indirect variables, the variable number of a simple variable is substituted by another simple integer variable enclosed in square brackets. For *global* indirection, the first two characters also denote memory type and data type. A simple integer variable following these characters in square brackets gives the absolute address of the referenced memory cell. This memory cell must have the type determined by the second character of the variable description.

Example 7.3: indirection

1.) local indirection
SI1:[SI0:3] nd
SI1:2 define the same memory cell, if the variable SI0:3 has value 2.

VI1:[SI0:3] or
VI1:[VI0:3] are determined in analogy.
(the latter by substituting the value of VI0:3 on each PE for [VI0:3].)

2.) global indirection
if SI1:2 has value 3, then
SR[SI1:2] addresses the scalar memory cell with absolute address 3 of type real.

VR[SI1:2] or
VR[VI1:2] are handled in analogy to the scalar case.
(For the latter, evaluating VI1:2 on each PE gives the variable address.
The referenced vector components must be of type real.)

Predefined Variables

PARZ has some predefined variables for special tasks:

MaxTrans	(maximum number of data elements to be transferred) write-only, is read by LOAD and STORE commands
ActTrans	(number of actually transferred elements) read-only, is set by LOAD and STORE commands
SResult	(boolean result of a scalar block-compare) read-only, receives a value from scalar EQUAL command
VResult	(boolean result of a vector block-compare) read-only, receives a value from vector EQUAL command
Done	(denotes successful input/output operations) read-only, receives a value from I/O commands
termCH	(holds termination character after reading a string) read-only, receives a value when reading a string

Address Operator

To use the absolute address of a simple variable as argument to a command, the variable is preceded by the address operator `ADDR`. The address of a variable can be used like an integer value.

Example 7.4: addresses

```
SI0:1 := ADDR SR0:2;
VI0:3 := ADDR VC1:5 + SI3:6;
```

Constants

All constants have a definite type:

- integer constants are positive integer numbers, written as a sequence of digits.
- real constants are positive numbers in floating-point representation.
- boolean constants are `TRUE` and `FALSE`.
- a char constant is an arbitrary character enclosed in apostrophes.
  If the character is the apostrophe itself, it has to be duplicated.
- `NIL` is a non-existing absolute address used as an integer.
- on each PE the value of the vector constant `ID` is the PE-number.
- `termS` is the string terminator. It is the character with code 0.
- `EOL` stands for the end-of-line character.

- Unprintable characters may be stated as constants by using the pseudo-function `CHR` with the character-code as argument. The PARZ simulator uses ASCII to code characters. `termS` and `CHR(0)` are equivalent.
- A string is a sequence of characters enclosed in quotes. Any quotes contained in the string have to be duplicated. Strings can be arguments of output commands, assignments, and comparisons.
- The `SIZE` operator gives the size of a memory block described by a declaration.

Example 7.5: constant values

```
1234,  0,
1.3,  3.,  .5,  .7E-2,  8.2e20,
'x',  '''',  chr(255),
"PARZ simulator",
ID,  NIL,
SIZE( I1 3( R1 C2))
```

In the present implementation, the latter example is equivalent to the integer value 10, as every memory element has equal length 1. In other implementations this may be different.

7.4 Pseudo-Assembly Commands

Each command in PARZ can be preceded by a label definition optionally combined with information on the source code and a record-flag.

A label definition has the EBNF-description:
labdef = label ["!" [line]] ["R"] ":" .

label	is an integer > 0. This number is used to refer to the following command.
"!"	is generated by the compiler to mark the first PARZ command for a Parallaxis statement
line	is the line-number in the source program
"R"	marks commands that will always appear in the recording-file when simulated

Each command is followed by a semicolon. All text from the semicolon to the end of a line is a comment. Whether a command is executed by the control processor or by the PE array depends on its arguments. The control processor only works on scalar data, while scalar arguments of a command for the PE-array have the same value on all PEs.

A variable which never got a value assigned is marked as not initialized. In each variable the type of its value is stored. A read-attempt to an **uninitialized** variable or a variable with the wrong value type causes a **warning** (this may be switched off, see section 4.1).

Common non-terminals in the following EBNF descriptions of PARZ commands are:

vardesc	simple, indirect or predefined variable (with write access)
VarConst	*vardesc*, variable with address op., constant value or any predefined variable
label	label defined in a label definition
declaration	declaration as described above

Words in capital letters are keywords. They may be written in capital or lower case letters.

Simple Commands

Assignment and Type Conversion

vardesc ":=" VarConst

The value of *VarConst* is assigned to *vardesc*. If *vardesc* and *VarConst* have different types, a type conversion is performed. The assignment is the only way to do a type conversion, all other commands produce an error in case of a type conflict. If *VarConst* is a vector, *vardesc* has to be a vector variable, too.

The following type conversions are possible:

boolean	→	integer/real	: result is 0 or 1 (0.0 or 1.0)
char	→	integer	: result is the code of *VarConst* (0 .. 255)
integer/real	→	boolean	: result is `TRUE`, if *VarConst* <> 0 (0.0)
integer	→	char	: result is character with code *VarConst*
integer	→	real	: result is *VarConst* as a real-value
real	→	integer	: result is trunc(*VarConst*)
string	→	char	: the string-constant *VarConst* is copied to a char-array with *vardesc* as first element (a string is terminated by `termS`)

Unary Operators

vardesc ":=" ("-" | NOT) VarConst

If *VarConst* is vector, *vardesc* has to be a vector-variable, too.

"-" is the negation operator for integer and real values.

NOT is used to negate booleans.

Binary Operators

vardesc ":=" VarConst1 op VarConst2

If one of the right-hand side arguments is a vector, *vardesc* has to be a vector-variable too. Arithmetic and logical operators to be inserted for *op* are:

"+"	: addition of integer- or real-values
"-"	: subtraction of integer- or real-values
"*"	: multiplication of integer- or real-values
"/"	: division of integer- or real-values (= DIV-operator with integers)
"^"	: power for the type combinations: *integer := integer ^ integer, real := real ^ integer, real := real ^ real*
MOD	: remainder of integer-division
AND	: logical *and* for booleans
OR	: logical *or* for booleans

The following relational operators for *op* need a boolean *vardesc*. The operators are defined for all types, but the right-hand side arguments must be of the same type. Boolean values are ordered `FALSE < TRUE`. Characters are ordered according to the character code.

"="	: equal
"<>"	: not equal
"<"	: less than
"<="	: less than or equal
">"	: greater than
">="	: greater than or equal

Unconditional Jump

GOTO label

Execution continues with the command denoted by *label*. This command mustn't be a `PROC`.

Conditional Jump

IF VarConst1 [op VarConst2] GOTO label

op is a relational operator. If *VarConst1* is the only argument, it has to be boolean, otherwise *'VarConst1 op VarConst2'* is a condition resulting in a boolean value. If the condition is `TRUE`, execution will continue with the command defined by *label*. Otherwise the program continues with the next command. All arguments have to be scalars. For vector conditions the conditional procedure call has to be used.

Block Copy

MOVE vardesc_s TO vardesc_d AS declaration

vardesc_s and *vardesc_d* are the first variables of two memory blocks, whose structure is defined by *declaration*. The contents of the source block beginning with *vardesc_s* is copied to the destination block beginning with *vardesc_d*. No warning results from uninitialized variables in the source block, but the corresponding destination variables will also become uninitialized. The types in both blocks have to match *declaration*, otherwise execution will abort. In case *vardesc_s* is a vector, *vardesc_d* has to be a vector as well.

Block Comparison

EQUAL vardesc1 vardesc2 AS declaration

This command compares the two memory blocks defined by two variable descriptions and a declaration. If both blocks are scalar, the result is written to `SResult`, otherwise to `VResult`. Two blocks are equal, if every two corresponding memory cells have the same value. No warning is generated, if the blocks contain uninitialized values. Two uninitialized values are regarded equal, but an uninitialized value does not match any initialized value.

Program Halt

HALT

This command terminates program execution.

Program End

END

This has to be the last command preceding the keyword `STOP`. It has to appear exactly once in a program. `END` and `HALT` have the same effect.

No Operation

NOP

This command has no effect.

Procedures

Procedure Call
CALL label
A new activation record containing return address and procedure level is generated on the call-stack. Program execution is continued with the `PROC`-command defined by the label. If the command marked with *label* is no `PROC`, an error is generated when reading the program. After returning from the called procedure, execution continues at the command following `CALL`.

Conditional Procedure Call
IF VarConst1 [op VarConst2] CALL label
VarConst1, *op* and *VarConst2* form a condition similar to the conditional jump command. But here, *VarConst1* and *VarConst2* may be **vector values**. The procedure defined by *label* is executed with all active PEs for which the condition is true. The activation record additionally contains the processor status which will be restored after the return. The calling mechanism is the same as for the `CALL` command. If the condition is false for **all** PEs, the procedure is **not called** at all!

While Loop
WHILE VarConst1 [op VarConst2] CALL label
This command is similar to an `IF-CALL`, but execution loops after returning from the procedure call. Only if the condition is false for **all PEs**, execution will resume with the next command. This command may also be expressed by the `REDUCE` and the `IF-CALL` command, which is especially necessary in case of more complex conditions:

Example 7.6: constructing a `WHILE` loop with `REDUCE` and `IF-CALL`

```
WHILE c CALL p;          c is a condition, p is a procedure label
```
is equivalent to:
```
1: vb := c;              vb is a temporary boolean vector
   sb := REDUCE OR OF vb;sb is a temporary boolean scalar
   IF sb = FALSE GOTO 2; done if c = FALSE on all PEs
   IF vb CALL p;
   GOTO 1;
2: ...
```

Procedure Head
PROC level [SCALAR s_declaration] [VECTOR v_declaration]
The procedure level of the static link is expressed by an integer > 0; this reflects the nesting level of a procedure in Parallaxis. *s_declaration* and *v_declaration* are the declarations of scalar and vector variables local to the procedure. These variables have the same level number *level*. They are allocated in the activation record generated by the calling `CALL` or `REDUCE` command. They may be accessed from the procedure and all nested subroutines. Entering a `PROC` command without a call will cause an error. A procedure may only call procedures with a nesting level less or equal to its own level plus 1, and it may only reference variables at the same or a lower nesting level.

Procedure Return
RETURN
Execution continues at the command following the procedure call (except `while`-call) and with the procedure level stored in the activation record after it is removed from the call-stack. A possibly saved processor status is restored.

Global and Local Stacks

Procedure parameters and results are saved on separate parameter stacks. Therefore, procedure parameters and function results may easily be passed via push and pop on the data stack. The control processor and each of the PEs have a separate data stack.

Push Data Stack
PUSHS VarConst
PUSHV VarConst
`PUSHS` puts a scalar value on top of the control stack. `PUSHV` puts a local value on top of each PEs local data stack.

Pop Data Stack
POPS vardesc
POPV vardesc
`POPS` takes a scalar value off the stack and assigns it to a variable (duplication if this variable is a vector). `POPV` takes a vector component off each active PE's data stack and assigns it to a local vector component.

Vector Commands

Set Processor Status
PARALLEL (bits | vardesc)
The `PARALLEL` command can activate and deactivate any PE (the same happens implicitly for a vector `CALL` and `RETURN`). PEs which were inactive on entering a procedure and therefore don't know any local variables, **cannot** be activated in that procedure. Trying to do this will cause a warning with the PE remaining inactive.

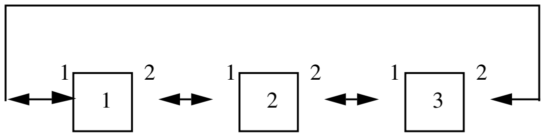
The desired processor status may be set by a bitstring (*bits*) or a variable. The bitstring contains the digits '1' (active) or '0' (passive) for each PE. The string may contain white space and may stretch over several lines. In case of a variable, it has to be the begin of a scalar boolean array. Each component of the array holds the status for one PE. An error occurs if the array does not contain enough elements.

Get Processor Status
vardesc ":=" STATUS
vardesc is the begin of a scalar boolean array. Each component of the array receives the status of one PE. An error occurs if the array is too small.

Adding connections
CONNECT o_port TO i_port AT VarConst
BICONNECT o_port TO i_port AT VarConst
On each PE these commands connect port *o_port* to port *i_port* of the PE specified by *VarConst*.
In the second form, the reverse connection is built up as well.

<u>Example 7.7:</u> establishing a ring of three processors with bidirectional connections

```
START
3 PE
2 PORTS
...
100 : VI0:10 := ID MOD 3;
101 : VI0:10 := VI0:10 + 1;
102 : BICONNECT 2 TO 1 AT VI0:10;
...
```



<u>Removing connections</u>
DISCONNECT [o_port]
If *o_port* is specified, connections leaving *o_port* on any PE are removed. Otherwise all connections are removed.

<u>Existence of Connections</u>
vardesc ":=" CONNECTED IN i_port
vardesc ":=" CONNECTED OUT o_port
vardesc ":=" CONNECTED IN i_port OUT o_port
vardesc ":=" CONNECTED OUT o_port IN i_port
This command has four forms. *vardesc* always is an integer vector-variable where the result is stored. Ports are scalar integer variables or constants that have to be valid port numbers. On each PE the commands count the number of connections ending in that port (`IN`, or `IN-OUT`) or leaving that port (`OUT`, or `OUT-IN`).

Data Exchange

<u>Value Propagation between Active Processors</u>

1.) PROPAGATE vardesc OUT o_port IN i_port
Each active PE puts the (vector-) value of *vardesc* on its port *o_port* (an integer with appropriate value), if there is an outward connection. In the second step, each active PE tries to read a value from its port *i_port*. If a connection ends there and a value arrives, this is the new value to be assigned to *vardesc*, otherwise *vardesc* remains unchanged.

2.) PUSHS o_port;
PUSHS i_port;
PROPAGATE VarConst TO v_vardesc [REDUCE (func | label)];
Each active PE puts its value of *VarConst* on their port *o_port* and then stores the values coming in on their port *i_port* in variable *v_vardesc*. If multiple values are received, a reduce operator must be specified by the optional argument (see REDUCE command).-
PROPAGATE removes the port parameters from the scalar stack.

<u>Value Propagation between Active and Inactive Processors</u>

1.) PUSHS o_port;
PUSHS i_port;
SEND VarConst TO v_vardesc [REDUCE (func | label)];

2.) PUSHS o_port;
PUSHS i_port;
RECEIVE v_vardesc FROM VarConst [REDUCE (func | label)];
The arguments have the same meaning as in the second form of the PROPAGATE command.
In the SEND command, active PEs are sending *VarConst*, all PEs are receiving values in *v_vardesc*.
In the RECEIVE command, all PEs are sending *VarConst*, active PEs are receiving values in *v_vardesc*.

<u>Data Transfer between Host and PEs</u>
LOAD v_vardesc WITH s_vardesc [(PE num) | (AS declaration)]
STORE v_vardesc TO s_vardesc [(PE num) | (AS declaration)]

v_vardesc	is a vector-variable
s_vardesc	is the first variable of a scalar array of the same type as *v_vardesc*.

`LOAD` moves scalar data (from the host) to a vector (to the PEs), while `STORE` moves vector data to a scalar, which usually is an array. With suffix "`PE` num", only a specific PE is affected, and *s_vardesc* is a single variable. With suffix "`AS` declaration", the vector variable is the beginning of a block as described by the declaration; the scalar variable is the beginning of an array of such blocks. `LOAD` copies the components of the scalar array to *v_vardesc* on all active PEs. The first active PE gets the first array component, the second active PE the second component, and so on. If `MaxTrans` is set to a value, it determines the maximum number of components to be transferred. After `LOAD` has been executed, `MaxTrans` becomes uninitialized and `ActTrans` contains the number of actually transferred variables or blocks. The reverse operation `STORE` works in analogy.

Vector Reduction

<u>Reduce Vector to Scalar</u>
vardesc ":=" REDUCE (func | label) OF VarConst
Reduces all components of a vector to a single scalar value by repeatedly applying a reduction function.

vardesc	is a scalar variable.
VarConst	is a vector value with the same type as *vardesc*. It has to be a vector variable or the vector constant `ID`.
func	is one of the following predefined function names:

<u>Predefined Reduce Functions:</u>

`FIRST`	for all types: returns the value on the first active PE. An error is reported if none of the PEs is active (exception: `ID` is reduced to number of PEs + 1 in this case).
`LAST`:	for all types: returns the value on the first active PE. An error is reported i none of the PEs is active (exception: `ID` is reduced to 0 in this case).

`SUM`: for integer/real: returns the sum of the values of all active PEs. Returns 0 if none of the PEs is active.
`PRODUCT` for integer/real: returns the product of the values of all active PEs. Returns 1 if none of the PEs is active.
`AND` for boolean: returns the logical conjunction of the values of all active PEs. Returns TRUE if none of the PEs is active.
`OR` for boolean: returns the logical disjunction of the values of all active PEs. Returns FALSE if none of the PEs is active.
`MAX` for all types: returns the maximum value of all active PEs. An error is reported if none of the PEs is active.
`MIN` for all types: returns the minimum value of all active PEs. An error is reported if none of the PEs is active.

If a label is used instead of one of the predefined functions, this determines a procedure to be used as a binary function for vector reduction. The function should be commutative and associative, as the result is computed by applying the operator to two values at a time, accumulating the final result. In order to cooperate with the `REDUCE` command, the function-procedure has to pop two values off the vector stack and push the result there upon leaving. The procedure is executed in a context as if called instead of the `REDUCE` command. The procedure mustn't contain any data exchanges or reductions.

7.5 Standard Functions

The predefined functions are normally applied in the following form with 0, 1, or 2 arguments:
vardesc ":=" func { VarConst }
All types are real, except where indicated. All angles are in radians.

Arithmetic Functions
ABS absolute value of argument, integer arguments are also allowed
SQRT square root
EXP exponential function (e^{argument})
LN natural logarithm
SIN sine
COS cosine
TAN tangent
ARCSIN arcus sine
ARCCOS arcus cosine
ARCTAN arcus tangent, result is in range - /2..+ /2
ARCTANT VarConst1 VarConst2 takes **two** arguments for unambiguous reverse function:
arctant (VarConst1/VarConst2)
the result is in range - ..+ with sign(arg1) = sign(sin), sign(arg2) = sign(cos)

String Functions
STRCMP VarConst1 VarConst2 string comparison, returns integer result: -1, 0 or 1, if the first string is lexically less than, equal to or greater than the second string. Arguments may be string constants or character variables denoting the begin of a character array, where the string is terminated by char `termS`.

Special Functions
RANDOM returns a random value of the type of the left-hand-side argument. If the variable is a vector, each PE gets its own random value.

NEW declaration a memory block with the structure defined by declaration is allocated on the heap. The returned integer value represents its absolute address. If *varesc* is a vector variable, a block is allocated in the heap of each PE. In the present implementation, heap addresses are negative integers.

INITSET bits The left hand side argument is the first element of an ARRAY OF BOOLEAN implementing a SET variable. 'bits' contains a character '0' or '1' for each possible member of the SET.

7.6 Standard Procedures

Predefined procedures are treated like pseudo-assembler commands and do not have to be executed by explicitly using `CALL`.

Input / Output Procedures

READ vardesc [VarConst]

The scalar or vector variable *vardesc* receives the value read from the active input stream. If a second argument is supplied, it is a scalar integer value representing the maximum length of a string to be read into the character array beginning with `vardesc`. To read a string, leading white space and control characters are skipped. The white space or control character following the string is copied to the variable `termCH`, but not removed from the input stream. If the type of *vardesc* is boolean, integer or real, a string is read as described above and converted to a value of the desired type. The only strings that can be converted to boolean, are `TRUE` and `FALSE` in capital or lower-case letters. If *vardesc* is a character variable and no second argument is supplied, exactly one character is read into *vardesc* without skipping white space. In this case, `termCH` isn't modified. After execution, `Done` reports the success of the operation. Read operations may fail, if no conversion is possible or if end-of-file is reached. The specialties of parallel reading from a terminal device are described in chapter 6.7 .

WRITE VarConst1 [VarConst2 [VarConst3]] or
WRITELN VarConst1 [VarConst2 [VarConst3]]

Writes the scalar or vector value of *VarConst1* to the current output stream; `WRITELN` prints an end-of-line character in addition. With only one argument, a standard format is used for values of each type (boolean, char, integer, real, string). With two arguments, *VarConst2* is the minimal output length for integers or reals. If *VarConst1* is a character variable, it is the first of a character array containing a string of maximal length *VarConst2*. Three arguments are used to write real-values in a fix-point representation. Here, *VarConst2* is the minimal output width and *VarConst3* is the fraction width. The specialties of parallel writing to a terminal device are described in chapter 6.7 .