The CM5 *Lisp Course

Zdzislaw Meglicki
Centre for Information Science Research,
The Australian National University,
18th & 19th January 1994

These notes were prepared with Andrew `ez`, Lucid Common Lisp, and with the Connection Machine CM5 *Lisp F7600 system on the computer facilities of the Centre for Information Science Research, The Australian National University. The notes are available in the `ez` format via `ftp-anonymous` from `arp.anu.edu.au`. Directory: `~ftp/ARP/papers/starlisp`.

Lesson 1
Introduction

Basic Common Lisp literature

1) ‘‘Lisp’’, 3rd Edition, Patrick Henry Winston (MIT) and Berthold Klaus Paul Horn (MIT), Addison-Wesley Publishing Company, 1989, ISBN 0-201-08319-1
2) ‘‘Structure and Interpretation of Computer Programs’’, Harold Abelson (MIT), Gerald Jay Sussman (MIT), Julie Sussman (MIT), The MIT Press, Eleventh Printing, 1990, ISBN 0-262-01077-1 (MIT Press), ISBN 0-07-000-422-6 (McGraw-Hill)
3) ‘‘Common Lisp The Language’’, 2nd Edition (CLtL2), Guy L. Steele Jr. (TMC), Digital Press, 1990, ISBN 1-55558-041-6

What is Common Lisp used for, examples

1) Hubble (Space) Telescope scheduling software (developed with Allegro Common Lisp),
2) ‘‘Cortex’’, expert system for predicting properties of chemicals in various circumstances developed by Molecular Knowledge Systems with Allegro Common Lisp,
3) Development of MS Windows applications,
4) Symbolic manipulation programs, e.g., Macsyma,
5) Automated reasoning systems, e.g., NQTHM by Boyer and Moore,
6) Numerical aerodynamic tunnel developed by Lockheed with *Lisp on the CM2,
7) Robot control software developed by MIT AI Labs with *Lisp on the CM2,
8) Software prototyping - both numerical and AI

Lisp on the Connection Machine

1) Originally built by MIT researchers as a massively parallel machine for AI,
2) Initially *Lisp was the only high level CM2 language available,
3) CM2 and CM5 are used at MIT AI labs for robot control,
4) Lockheed was one of the first companies to make use of the CM2 for numerical applications, their software was developed under *Lisp,
5) The University of Maryland developed an enemy ship recognition software for the US Navy using *Lisp on the CM2,
6) *Lisp is still the only interactive prototyping environment available on the CM5

Prototyping versus production code development

1) The role of experimentation while developing difficult algorithms
2) Experimentation is greatly hindered when working with non-interactive programming environments:

```
edit
        compile
                run
                        invoke external debugger
re-edit
        re-compile
                re-run
                        re-invoke external debugger
...
```

3) *Lisp codes can be compiled after the prototyping stage. Because compiled codes invoke basic CM-functions on the machine level, they should run as fast as CMF or C* codes.

Working with Epoch and Lisp

1) Interactive programming environments such as Lisp or Prolog require buffering for
   automatic formatting
   saving the worksheet
   editing the worksheet
2) Emacs and Epoch are traditionally used for this purpose
3) Epoch knows better how to work with X11 than Emacs. The new emacs-19 is still dangerously buggy
4) The usual way to work with Lisp through Epoch is to have two windows: one for worksheet editing and the other one for talking to Lisp. Lisp clauses can be selected in the worksheet window and transferred in various ways to the Lisp window. Text can be entered in Lisp window directly. It will be automatically formatted by Epoch during insertion showing logical structure of typed in programs.

Interactive session with Epoch

1) Login on the system
2) Select Epoch from the root menu

Conversing with Epoch Lisp

When Epoch is brought up without any arguments, it provides a window marked "`(Lisp Interaction)`". From within this window you can talk directly to Epoch Lisp.

1) Your first Lisp program: type `(print "hello world")` in the Lisp Interaction window and press Line-Feed. Note, Line-Feed is the little key under Return. Instead of Line-Feed you can also press `C-j`. "`C-j`" means "Control-j".
2) Epoch/Emacs Lisp is similar to Common Lisp, but it is not exactly the same. We will not dwell on Epoch Lisp any more.

Conversing with Common Lisp through Epoch

1) To edit a file with Epoch type `C-x C-f`. Note that after you type `C-x`, the cursor jumps to the Minibuffer window. Then when you type `C-f`, you see in the Minibuffer

   ```
   Find File: ~/
   ```

   type the name of the file you want to edit or create, e.g., `worksheet.l`
2) Epoch window will clear and you will see that the editing mode now becomes "`(Lisp)`". This is not the real Lisp window yet. This is a window in which you will be editing your Lisp worksheet. Epoch will help you edit the program - but you will have to invoke yet another window to bring up real Common Lisp in it.
3) Type "`C-z 2`". This will bring up another window with the cursor positioned in it. Now in this second window type "`M-x run-lisp`" - this will bring up Lucid Common Lisp in the second window. The abbreviation "`M-x`" means Meta-x. Meta key is the small key labeled with a diamond to the left and to the right of the space-bar key. The Epoch mode line says "`(Inferior Lisp: run)`"
4) Type in your first Lisp program into the Common Lisp window: `(print "hello world")` and press the Return key this time, not the Line-Feed key. You should see:

   ```
   > (print "hello world")

   "hello world"
   "hello world"
   >
   ```

   Observe that when you have typed in the second bracket the cursor briefly jumps to the first bracket in order to show you which bracket you are closing.

5) Now type "`C-z o`" in the Lisp window, the cursor will jump back to the worksheet window. In that window type again your first Lisp program `(print "hello world")`. Observe that again the cursor will show you which bracket you are closing when you type the second bracket. After you type in the line, press "`C-M-x`". Observe that Epoch will have transferred the program to the Lisp window and executed it there.
6) You can also select the text in the worksheet window by pressing the first button on the mouse and dragging the mouse across text. Then click on the Lisp window, position the mouse pointer in the shape of the pencil exactly where you want the text to go, and press the middle button on the mouse. This will transfer the text from the worksheet window to the Lisp window. Now press the Return key and execute the program.

Summary

1) When Epoch is invoked from the menu or without arguments in brings up the Epoch-Lisp interaction window, in which Epoch-Lisp commands can be executed by pressing the Line-Feed key.
2) To create a new file or to edit an existing one type `C-x C-f file-name` in the Epoch window. File names with a suffix "`.l`" will automatically invoke the Lisp editing mode.
3) To create another window type "`C-z 2`".
4) To invoke external Lisp process in that window, position the cursor in that window and type "`M-x run-lisp`"
5) Programs can be typed and executed directly in the Lisp window. Epoch will automatically show closing brackets and indent typed in text.
6) You can switch from one window to another by typing "`C-z o`" or by clicking with the mouse on the appropriate window.
7) You can transfer Lisp clauses from Lisp window to the inferior Lisp process by typing "`C-M-x`" or by copying and pasting with the mouse.

Lesson 2
Basic Common Lisp

Using Common Lisp as a calculator

1) Switch back to the Common Lisp window and try the following:

```
> (+ 2 2)
4
> (* 2 3)
6
```
Here is how you can combine various mathematical operations
```
> (+ (* 2 3) 4)
10
```
When performing operations on integers Lisp will try to be as accurate as possible to the very end
```
> (/ 3 7)
3/7
```
Common Lisp knows the value of `pi`
```
> pi
3.141592653589793
```
It also knows about trigonometric functions
```
> (sin (/ pi 2.0))
1.0
> (cos (/ pi 2.0))
6.123031769111886E-17
```
This should be zero, but we are not so accurate
```
> (tan (/ pi 2.0))
1.6331778728383844E16
```
This should be infinity, but, thanks God, we are not sufficiently accurate either.
Lisp functions can sometimes take arbitrary number of arguments
```
> (max 3 7 2 1 6)
7
> (min 3 7 2 1 6)
1
> (+ 3 7 2 1 6)
19
> (- 3 7 2 1 6)
-13
> (1+ 3)
4
> (1- 3)
```

```
2
> (/ 8)
1/8
> (/ -8)
-1/8
```
Common Lisp also knows about the greatest common divisor
```
> (gcd 8 4)
4
> (gcd 236 111)
1
```
It can also exponentiate
```
> (expt 2 3)
8
```
Note the difference between 2^3 and e^2
```
> (exp 2)
7.38905609893065
> (log 2)
0.6931471805599453
```
If you want a logarithm of 2 in the base 2 use two arguments. The default base is e
```
> (log 2 2)
1.0
```
Square root is also there
```
> (sqrt 2)
1.4142135623730952
```
Common Lisp knows about complex numbers
```
> (log -1.0)
#C(0.0 3.141592653589793)
> (sqrt -2)
#C(0.0 1.4142135623730952)
```
This is an integer square root
```
> (isqrt 9)
3
> (isqrt 10)
3
```
and this is e^{ix}
```
> (cis 0.0)
#C(1.0 0.0)
> (cis pi)
#C(-1.0 1.2246063538223773E-16)
> (cis (/ pi 2))
#C(6.123031769111886E-17 1.0)
```
Common Lisp understands fractions
```
> (numerator (/ 3 4))
```

```
3
> (denominator (/ 3 4))
4
```
It can also truncate and round numbers
```
> (truncate pi)
3
0.14159265358979312
> (round pi)
3
0.14159265358979312
> (truncate (+ pi (/ 1 2)))
3
0.6415926535897931
> (round (+ pi (/ 1 2)))
4
-0.3584073464102069
```
or cast them onto other types
```
> (float 2)
2.0
> (rational 2.0)
2
```
Here is how you can find the division remainder
```
> (rem 13 4)
1
```
To generate random numbers use
```
> (random 10.0)
2.061467316924359
> (random 10.0)
3.7399819979840676
>
```

There are many more useful mathematical functions in Common Lisp. These are discussed in detail in Chapter 12, "Numbers" of the CLtL2.

Basic iterations in Common Lisp

1) Still in Lisp window type the following text. With the exception of the first line start every following line with a tab to get the right indentation.

```
> (dotimes (i 5 nil)
    (print i))

0
1
```

```
2
3
4
NIL
>
```

Variables and more iterations in Common Lisp

1) Try the following

```
> a
>>Error: The symbol A has no global value

SYMBOL-VALUE:
   Required arg 0 (S): A
:C  0: Try evaluating A again
:A  1: Abort to Lisp Top Level

-> 1
Abort to Lisp Top Level
Back to Lisp Top Level

> (setf a 3)
3
> a
3
>
```

In Common Lisp parlance we say that symbol A has been bound to 3.

2) Variables can be also created locally using the `let` form:

```
> (let ((result 1))
    (dotimes (i 5 result)
      (setf result (* (1+ i) result))))
120
> (* 1 2 3 4 5)
120
>
```

3) This `(1+ i)` construct within the body of the loop looks ungainly. A more general iteration facility called "`do`" can be used to fix this:

```
> (let ((result 1))
```

```
    (do ((i 1 (1+ i)))
        ((< 5 i) result)
      (setf result (* i result))))
120
>
```

The first argument of `do` creates local variables, initialises them and tells `do` how to increment them on every iteration. Several local variables can be created. We could use this feature of `do` instead of the external `let`:

```
> (do ((i 1 (1+ i))
       (result 1))
      ((< 5 i) result)
    (setf result (* i result)))
120
>
```

Defining your own functions

1) The macro `defun` is used to define functions. Macros is Common Lisp are a bit like macros in C. They are far more sophisticated though and, like Common Lisp functions, they can be compiled too. Here is a simple use of `defun`:

```
> (defun factorial (n)
    (do ((i 1)
         (result 1))
     ((< n i) result)
      (setf result (* i result))
      (setf i (1+ i))))
FACTORIAL
> (factorial 5)
120
> (factorial 1)
1
> (factorial 3)
6
> (factorial 7)
5040
> (factorial 0)
1
>
```

2) When it comes to defining long functions it is no longer practical to work in the Lisp window. For this it is better to switch to the worksheet window so that you can re-edit and modify existing definitions. If you have managed to type in the definition of the factorial function without mistakes into the Lisp window, copy and paste this definition using mouse to the worksheet window.

3) The copied text may not be properly indented. To fix the problems try the following
   a) position the cursor at the beginning of the definition and type `C-space-bar`, the minibuffer should say "`Mark set`"
   b) now type `C-M-f`, the cursor should move to the end of the function definition. You can go back to the beginning by typing `C-M-b`, try it
   c) when you are now positioned at the end of the function definition type `M-x indent-region`.
4) Modify the function definition by adding the comment:

```
(defun factorial (n)
  "this function computes the factorial of its
integer argument"
  (do ((i 1)
       (result 1))
      ((< n i) result)
    (setf result (* i result))
    (setf i (1+ i))))
```

5) Transfer this new function definition to Common Lisp by positioning the cursor at the end of the function and typing `C-M-x`. Lisp should give you a warning that you are redefining the function:

```
;;; Warning: Redefining FUNCTION FACTORIAL which used
to be defined at top level
```

6) Now change back to the Lisp window and type:

```
> (describe 'factorial)
FACTORIAL is a symbol.  Its home package is USER.
Its global function definition is #<Interpreted-
Function (NAMED-LAMBDA FACTORIAL (N) (BLOCK FACTORIAL
(DO # # # #))) 157A406>.
#<Interpreted-Function (NAMED-LAMBDA FACTORIAL (N)
(BLOCK FACTORIAL (DO # # # #))) 157A406> is an
interpreted function.
Its source code is


(NAMED-LAMBDA FACTORIAL
              (N)
              (BLOCK FACTORIAL
                (DO ((I 1)
                     (RESULT 1))
                    ((< N I)
```

```
                    RESULT)
                  (SETF RESULT (* I RESULT)) (SETF I
(1+ I)))))
The function definition for FACTORIAL is in the file
/tmp/emlisp3773.
It has this function documentation:
"this function computes the factorial of its integer
argument"
>
```

Common Lisp is a self-documenting programming environment. You can find much information about any function, macro, or a variable by asking Common Lisp to `describe` it.

7) You can easily time every function which executes in the Common Lisp environment. Try

```
> (time (factorial 180))
Elapsed Real Time = 0.03 seconds
Total Run Time    = 0.03 seconds
User Run Time     = 0.03 seconds
System Run Time   = 0.00 seconds
Process Page Faults    =          0
Dynamic Bytes Consed   =          0
Ephemeral Bytes Consed =     15,728
2008960624991342996569513368984668389175
4034079886777794043533516004486095339598
0941180138112097309735631594101037399609
6710321321863314952736095985319667309729
4565355881980647506435385685815744504080
9209560358463319644664891114256430017824
1417967538181923386423026933278187319860
3960320000000000000000000000000000000000
0000000000
```

Here you can also see that integer arithmetics in Common Lisp is not restricted to 32 or 64 bits.

8) The function `factorial` defined above is interpreted. You can speed up the execution by compiling it:

```
> (compile 'factorial)
;;; You are using the compiler in development mode
(compilation-speed = 3)
```

```
;;; If you want faster code at the expense of longer
compile time,
;;; you should use the production mode of the compiler,
which can be obtained
;;; by evaluating (proclaim '(optimize (compilation-
speed 0)))
;;; Generation of full safety checking code is enabled
(safety = 3)
;;; Optimization of tail calls is disabled (speed = 2)
FACTORIAL
> (time (factorial 180))
Elapsed Real Time = 0.01 seconds
Total Run Time    = 0.01 seconds
User Run Time     = 0.01 seconds
System Run Time   = 0.00 seconds
Process Page Faults    =          0
Dynamic Bytes Consed   =          0
Ephemeral Bytes Consed =     15,640
2008960624991342996569513368984668389175
4034079886777794043533516004486095339598
0941180138112097309735631594101037399609
6710321321863314952736095985319667309729
4565355881980647506435385685815744504080
9209560358463319644664891114256430017824
1417967538181923386423026933278187319860
3960320000000000000000000000000000000000
0000000000
>
```

Summary

1) Common Lisp provides a great variety of mathematical functions. For example:
   `+, *, -, /, sin, cos, tan, max, min, gcd, expt, exp, log, sqrt, isqrt, cis, 1+, 1-, numerator, denominator, truncate, round, float, rational, rem, random`
   and many others.
2) Common Lisp knows about fractions and complex numbers. It also knows about `pi`.
3) Basic iterations in Common Lisp can be carried out using `dotimes` or `do`.
4) Variables are created using `setf`, `let`, or let-like constructs within `do`. There are also many other ways to create and bind variables in Common Lisp.
5) Functions are defined using the `defun` macro.
6) Long functions are easier to define and modify in the worksheet window.
7) You can mark a position in a text in Epoch by `C-space-bar`.
8) `C-M-f` will move the cursor forward by the whole Lisp clause and `C-M-b` will move the cursor backward by the whole Lisp clause.

9) The whole function can be automatically indented by marking the region and typing `M-x indent-region.`
10) Common Lisp functions can be annotated by inserting a comment following the list of parameters.
11) The documentation about every Common Lisp function is stored by the Lisp system and can be invoked by calling the `describe` function.
12) Common Lisp functions can be timed by using the `time` function.
13) Common Lisp functions can be compiled by using the `compile` function.
14) Common Lisp integer arithmetics is not restricted to 32 or 64 bits.

Lesson 3
Arrays

Common Lisp arrays

1) Variables in Common Lisp are dynamically constructed as needed. `setf` makes a simple variable if such doesn't exist yet. To make an array you have to use a function `make-array`.

```
> (make-array '(4))
#<Simple-Vector T 4 1537E9E>
> (setf *print-array* t)
T
> (make-array '(4))
#(NIL NIL NIL NIL)
>
```

Unless a predefined global variable `*print-array*` is set to `t` (true) array contents are not printed. This can be useful if the program works on very long arrays. By default newly created arrays contain `nil` (false) in every slot.

2) Arrays can be initialised using a function switch "`:initial-element`", or "`:initial-contents`":

```
> (make-array '(4) :initial-element 0.0)
#(0.0 0.0 0.0 0.0)
> (make-array '(4)
    :initial-contents '(1.0 2.0 3.0 4.0))
#(1.0 2.0 3.0 4.0)
>
```

3) Arrays can be made multidimensional:

```
> (make-array '(3 3) :initial-element 0.0)
#2A((0.0 0.0 0.0) (0.0 0.0 0.0) (0.0 0.0 0.0))
> (make-array '(3 3)
    :initial-contents '((1.0 2.0 3.0)
                        (4.0 5.0 6.0)
                        (7.0 8.0 9.0)))
#2A((1.0 2.0 3.0) (4.0 5.0 6.0) (7.0 8.0 9.0))
>
```

4) Once made, arrays can be bound to various symbols using `setf`:

```
> (setf a (make-array '(3)
           :initial-contents '(1.0 0.0 0.0)))
#(1.0 0.0 0.0)
> (setf b (make-array '(3)
           :initial-contents '(0.0 1.0 0.0)))
#(0.0 1.0 0.0)
> a
#(1.0 0.0 0.0)
> b
#(0.0 1.0 0.0)
>
```

5) We can retrieve selected elements of an array using the function `aref`:

```
> (aref a 0)
1.0
> (aref a 1)
0.0
> (aref a 2)
0.0
>
```

6) With multidimensional arrays function `aref` takes two or three indexes depending on the dimension of the array. Let's define as an exercise Pauli matrices:

```
> (setf i (complex 0 1))
#C(0 1)
> (* i i)
-1
> (setf -i (complex 0 -1))
#C(0 -1)
```

Observe that variable name may begin with "-". This can be a very useful feature. Also, note that we have used a new function `complex` *which is used to make complex numbers.*

```
> (* -i -i)
-1
> (* i -i)
1
> (setf sigma-x (make-array '(2 2)
                   :initial-contents '((0 1)
                                       (1 0))))
#2A((0 1) (1 0))
> (setf sigma-y (make-array '(2 2)
                   :initial-contents (list
```

```
                                          (list 0 -i)
                                          (list i 0))))
#2A((0 #C(0 -1)) (#C(0 1) 0))
```
Observe that we have to use a new function `list`*, which makes a list of its arguments, because* `i` *and* `-i` *are now symbols which must not be quoted.*
```
> (setf sigma-z (make-array '(2 2)
                   :initial-contents '((1 0)
                                       (0 -1))))
#2A((1 0) (0 -1))
>
```
Now, in order to extract the elements of those arrays do:
```
> (aref sigma-x 0 1)
1
> (aref sigma-x 1 0)
1
> (aref sigma-y 0 1)
-I
> (aref sigma-y 1 0)
I
>
```

7) Common Lisp provides array information functions:

```
> (array-rank sigma-y)
2
> (array-dimension sigma-y 0)
2
> (array-dimension sigma-y 1)
2
> (array-dimensions sigma-y)
(2 2)
> (array-total-size sigma-y)
4
>
```

8) Let us define a simple function for multiplication of 2x2 matrices:

```
(defun matrix-multiply (a b)
  (let ((a00 (aref a 0 0))
     (a01 (aref a 0 1))
     (a10 (aref a 1 0))
     (a11 (aref a 1 1))
     (b00 (aref b 0 0))
     (b01 (aref b 0 1))
```

```
     (b10 (aref b 1 0))
     (b11 (aref b 1 1)))
    (make-array '(2 2)
          :initial-contents
          (list
           (list (+ (* a00 b00) (* a01 b10))
                 (+ (* a00 b01) (* a01 b11)))
           (list (+ (* a10 b00) (* a11 b10))
                 (+ (* a10 b01) (* a11 b11)))))))
```

9) We can now use this function in order to check if various properties of Pauli matrices are satisfied:

```
> (matrix-multiply sigma-x sigma-x)
#2A((1 0) (0 1))
> (matrix-multiply sigma-y sigma-y)
#2A((1 0) (0 1))
> (matrix-multiply sigma-z sigma-z)
#2A((1 0) (0 1))
> (matrix-multiply sigma-x sigma-y)
#2A((#C(0 1) 0) (0 #C(0 -1)))
> (matrix-multiply sigma-y sigma-x)
#2A((#C(0 -1) 0) (0 #C(0 1)))
> (matrix-multiply sigma-y sigma-z)
#2A((0 #C(0 1)) (#C(0 1) 0))
> (matrix-multiply sigma-z sigma-y)
#2A((0 #C(0 -1)) (#C(0 -1) 0))
> (matrix-multiply sigma-z sigma-x)
#2A((0 1) (-1 0))
> (matrix-multiply sigma-x sigma-z)
#2A((0 -1) (1 0))
> (matrix-multiply (matrix-multiply sigma-x sigma-y)
sigma-z)
#2A((#C(0 1) 0) (0 #C(0 1)))
>
```

10) Specific entries in the arrays can not only be read using `aref` but also changed using the combination of `aref` and `setf`:

```
>  (setf a (make-array '(4)
             :initial-contents '(1 2 3 4)))
#(1 2 3 4)
> (setf (aref a 2) 55)
55
```

```
> a
#(1 2 55 4)
>
```

11) We can now write a more universal function which multiplies two matrices of sizes `m x n` and `n x p`:

```
(defun matrix-multiply (a b)
  "multiply matrices a (m x n) and b (n x p)"
  (let ((na (array-dimension a 1))
        (nb (array-dimension b 0)))
    (if (/= na nb)
        (error
          "Error matrices have incompatible dimensions")
      (let* ((n na)
             (m (array-dimension a 0))
             (p (array-dimension b 1))
             (answer-array
               (make-array (list m p)
                           :initial-element 0.0)))
        (dotimes (i m answer-array)
          (dotimes (j p)
            (let ((sum 0.0))
              (dotimes (k n)
                (setf sum
                      (+ sum
                         (* (aref a i k)
                            (aref b k j)))))
              (setf (aref answer-array i j)
                    sum)))))))))
```

We have used here two new functions: `if` and `let*`. The meaning of `if` is the same as in other languages. Note that `/=` means *not-equal*. `let*` is a function which is very similar to `let`. The difference is that `let` performs assignments in parallel, whereas `let*` performs assignments sequentially.

12) First, observe how the error condition works:

```
> (matrix-multiply #2A((1 2) (2 3))
                   #2A((1 2 3) (3 4 5) (5 6 7)))
>>Error: Error matrices have incompatible dimensions

MATRIX-MULTIPLY:
Original code: (NAMED-LAMBDA MATRIX-MULTIPLY (A B)
(BLOCK MATRIX-MULTIPLY (LET # #)))
```

```
   Required arg 0 (A): #2A((1 2) (2 3))
   Required arg 1 (B): #2A((1 2 3) (3 4 5) (5 6 7))
:A  0: Abort to Lisp Top Level

-> 0
Abort to Lisp Top Level
Back to Lisp Top Level

>
```

13) Now you can also check that the Pauli matrices, sigma-x, sigma-y, and sigma-z multiply by one another as before.

Summary

1) Arrays in Common Lisp are made using the function `make-array`.
2) Arrays can be initialised using the `:initial-element` or the `:initial-contents` switch to `make-array`.
3) Arrays can be bound to symbols (variables) with `setf`.
4) Array entries can be retrieved using the function `aref`. A combination of `setf` and `aref` can be used to write onto selected array entries.
5) Complex numbers can be made using the function `complex`.
6) Variable names can begin with "`-`".
7) The function `list` is used to construct a list from elements that require further evaluation by Lisp.
8) Common Lisp provides various array information functions, e.g., `array-rank`, `array-dimension`, `array-dimensions`, `array-total-size`.
9) "`if`" is available in Common Lisp and it works as in other languages
10) The difference between `let*` and `let` is that `let*` carries out its assignments sequentially, whereas `let` carries out its assignments in parallel.
11) The function `error` can be used to break the execution of a function and write an error message.

Lesson 4
Structures

Structures in Common Lisp

A modern computer language is unthinkable without structures. In Lisp structures, arrays, objects, etc, can be easily made by hand from basic primitives (see, e.g., Abelson & Sussmans). But the idea behind Common Lisp is to save the programmer the trouble of having to reinvent the wheel all the time. Structures, objects, generic functions, series, fancy iterative facilities, sophisticated string and list processing functions are all provided.

1) First let us define a number of special variables. It is a Lisp convention that the names of special variables begin and end with an asterisk. The function used to define special variables is `defvar`:

```
(defvar *qe* 1.60210E-19 "Charge of positron in C")
(defvar *-qe* (- *qe*) "Charge of electron in C")
(defvar *me* 9.1091E-31 "Mass of electron in kg")
(defvar *mp* 1.67252E-27 "Mass of proton in kg")
(defvar *h-bar* (/ 6.6256E-34 2 pi)
        "Reduced Planck constant in J s")
(defvar *1/2*h-bar* (* (/ 2) *h-bar*)
        "Quantum of spin in J s")
(defvar *-1/2*h-bar* (- *1/2*h-bar*)
        "Minus quantum of spin in J s")
(defvar *mue* (* (/ *-qe* *me*) *1/2*h-bar*)
        "Magnetic moment of electron in A m^2")
```

The commenting strings will be displayed by Lisp when you ask it to `(describe '*qe*)`.

2) Now let us use the macro `defstruct` to define a structure called `classical-particle`:

```
(defstruct classical-particle
  "A structure which defines a classical particle
with mass, charge magnetic-moment, position, and
velocity"
  (mass *me*)
  (charge *-qe*)
  (magnetic-moment *mue*)
  (position #(0.0 0.0 0.0))
```

```
  (velocity #(0.0 0.0 0.0)))
```

3) Invoking the macro `defstruct` generates automagically instance creation function `make-classical-particle` and slot accessor functions `classical-particle-mass`, `classical-particle-charge`, etc:

```
> (setf *print-array* t)
T
> (make-classical-particle)
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(0.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
>
```

3) The default parameters used in the definition can be changed during instantiation of the structure by using the corresponding switch:

```
> (setf electron-1 (make-classical-particle
                     :position #(1.0 0.0 0.0)))
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(1.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
> (setf electron-2 (make-classical-particle
                     :position #(-1.0 0.0 0.0)))
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(-1.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
> (setf proton-1 (make-classical-particle
            :mass *mp* :charge *qe*
            :magnetic-moment (/ (* 2.79 *qe* *1/2*h-
bar*) 2 *mp*)))
#S(CLASSICAL-PARTICLE MASS 1.67252E-27 CHARGE
1.6021E-19 MAGNETIC-MOMENT 7.045435727927414E-27
POSITION #(0.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
>
```

4) The accessor functions are used to extract values of a particular slot:

```
> (classical-particle-position electron-1)
#(1.0 0.0 0.0)
> (classical-particle-position electron-2)
#(-1.0 0.0 0.0)
> (classical-particle-position proton-1)
#(0.0 0.0 0.0)
```

```
> (classical-particle-mass proton-1)
1.67252E-27
>
```

5) We can use the combination of `setf` and accessor functions to change values of specified slots in already existing structures:

```
> (setf (classical-particle-position electron-2)
        #(-0.8 0.0 0.0))
#(-0.8 0.0 0.0)
> electron-2
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(-0.8 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
>
```

6) Because `:position` is an array we can operate also on the selected slot of that array in the following way:

```
> (setf
    (aref
      (classical-particle-position electron-2) 2)
    0.2)
0.2
> electron-2
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(-0.8 0.0 0.2) VELOCITY #(0.0 0.0 0.0))
>
```

`setf` is a very powerful facility. It accesses and modifies values of specified data items in situ.

7) Existing definitions of structures can be reused in defining larger structures which incorporate the already existing ones:

```
(defstruct (quantum-particle
             (:include classical-particle))
  (flavour 'electron)
  (colour 'white)
  (spin 'up))
```

8) When a quantum particle is made, it incorporates the properties of a classical particle and adds some new properties to it:

```
> (make-quantum-particle)
#S(QUANTUM-PARTICLE MASS 9.1091E-31 CHARGE -1.6021E-19
MAGNETIC-MOMENT -9.273197292819559E-24 POSITION #(0.0
0.0 0.0) VELOCITY #(0.0 0.0 0.0) FLAVOUR ELECTRON
COLOUR WHITE SPIN UP)
>
```

9) Structures are the precursors of objects. Common Lisp provides the most elaborate and powerful object oriented programming facility of all object oriented languages. It is called CLOS (Common Lisp Object System). Consequently, structures in Common Lisp are used rather infrequently nowadays. However *Lisp does not implement CLOS in its parallel part. For this reason we won't go beyond structures this time.

Summary

1) Special variables in Common Lisp programs are defined using `defvar`.
2) Traditionally the names of Common Lisp special variables begin and end with an asterisk.
3) Structure types are defined using the macro `defstruct`. The macro automatically generates and compiles creation and accessor functions.
4) To create a structure instance use a `make-<`*`structure-name`*`>` function.
5) The default values of slots can be altered during structure instantiation.
6) To access a particular slot in a structure use a function `<`*`structure-name`*`>-<`*`slot-name`*`>`.
7) The accessor functions can be used to modify the values of the slots if combined with `setf`.
8) Existing structures can be used to define new larger structures, which incorporate the already existing ones.

Lesson 5
Input and Output

Format

Like Fortan and C, Common Lisp has a very elaborate `format` function. It has also a variety of simpler functions for I/O:

1) Let's try some simple functions first. Compare:

```
> (print electron-1)

#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(0.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
#S(CLASSICAL-PARTICLE MASS 9.1091E-31 CHARGE
-1.6021E-19 MAGNETIC-MOMENT -9.273197292819559E-24
POSITION #(0.0 0.0 0.0) VELOCITY #(0.0 0.0 0.0))
> (pprint electron-1)

#S(CLASSICAL-PARTICLE MASS 9.1091E-31
                      CHARGE -1.6021E-19
                      MAGNETIC-MOMENT
-9.273197292819559E-24
                      POSITION #(0.0 0.0 0.0)
                      VELOCITY #(0.0 0.0 0.0))
>
```

2) Function `format` allows more elaborate operations on output strings. First compare:

```
> (format t "Hello World")
Hello World
NIL
> (format nil "Hello World")
"Hello World"
>
```

In the first case the function evaluates a side action (printing) and returns `nil`. In the second case the function returns the formatted string.

3) Like in C or in Fortran, the string can contain a variety of format directives. The directives begin with tilde "~". Compare:

```
> (format nil "~&Mrs Barling is ~D years old" 40)
```

```
"Mrs Barling is 40 years old"
> (format nil "~&Mrs Barling is ~B years old" 40)
"Mrs Barling is 101000 years old"
> (format nil "~&Mrs Barling is ~O years old" 40)
"Mrs Barling is 50 years old"
> (format nil "~&Mrs Barling is ~X years old" 40)
"Mrs Barling is 28 years old"
> (format nil "~&Mrs Barling is ~R years old" 40)
"Mrs Barling is forty years old"
> (format nil "~&Mrs Barling is ~@R years old" 40)
"Mrs Barling is XL years old"
>
```

4) Floating point numbers can be formatted with `~F` or `~E`:

```
> (format nil "~&I paid $~F for this car" 35725.23)
"I paid $35725.23 for this car"
> (format nil "~&I paid $~E for this car" 35725.23)
"I paid $35725.23 for this car"
> (format nil "~&I paid $~8,5E for this car" 35725.23)
"I paid $3.57252E+4 for this car"
>
```

5) The function `format` can take care of pluralising nouns. The directives `~:@P` and `~:P` are used for that:

```
> (format nil "~D tr~:@P/~D win~:P" 7 1)
"7 tries/1 win"
> (format nil "~D tr~:@P/~D win~:P" 1 0)
"1 try/0 wins"
> (format nil "~D tr~:@P/~D win~:P" 1 3)
"1 try/3 wins"
>
```

6) The directives `~A` and `~S` take any symbolic expression including numbers:

```
> (format nil "The first list is ~A" '(a b c d))
"The first list is (A B C D)"
> (format nil "The first list is ~S" '(a b c d))
"The first list is (A B C D)"
>
```

7) The function `format` has great many other features. Its discussion takes 29 pages in CLtL2.

Writing data on files

There are various ways to write and read files in Common Lisp. The easiest way is to use the function `with-open-file`, which takes care of most other details. The file remains open only within the body of that function.

1) Try the following:

```
> (with-open-file (roman-numerals "roman.dat"
                                  :direction :output)
    (dotimes (i 20)
      (format roman-numerals "~D = ~@R~%" i i)))
NIL
>
```

Observe several new points: the name of the output stream inside the Lisp program is "`roman-numerals`". This output stream corresponds to UNIX file "`roman.dat`". The same function `with-open-file` can be used both for reading and for writing depending on the value of the key `:direction`. The `~%` format directive prints new-line at the end of the string. If you don't want `dotimes` to return anything you can skip the third argument in the list - `dotimes` will then return `nil` as the default. The first argument to `format` function is the name of the destination stream. "`t`" is taken to mean "standard output", and "`nil`" means "no output".

2) Now go to the UNIX shell and inspect the file "`roman.dat`" which Lisp created in your current directory:

```
gustav@arp:~/Lispcourse 206 $ cat roman.dat
0 = 0
1 = I
2 = II
3 = III
4 = IV
5 = V
6 = VI
7 = VII
8 = VIII
9 = IX
10 = X
11 = XI
12 = XII
13 = XIII
14 = XIV
15 = XV
```

```
16 = XVI
17 = XVII
18 = XVIII
19 = XIX
gustav@arp:~/Lispcourse 207 $
```

3) Common Lisp has provisions only for byte or character streams. There are no streams of reals, integers, etc. The programmer is responsible for coding reals into bytes. This is an obvious hassle and a shortcoming. The reason for this is that at the time Common Lisp was originally conceived there were no agreed on IEEE standards for floating point representations available yet. Even nowadays integer or floating point number files are seldom portable across various architectures, unless the architectures support the IEEE floating point representation - not all systems do! If instead you take care of representing floating point numbers and integers yourself you'll be always portable. Common Lisp, like Ada is paranoid about portability and operating system independence.

4) Writing binary files is as easy as writing files of characters:

```
> (with-open-file (binary-numerals "binary.dat"
                     :direction :output
                     :element-type 'unsigned-byte)
                   (dotimes (i 20)
                      (write-byte i binary-numerals)))
NIL
>
```

Here we had to use a special function `write-byte` which converts an integer to a byte and writes it on the output stream.

5) We can view this newly created file of bytes with UNIX program `od`:

```
gustav@arp:~/Lispcourse 214 $ od -b binary.dat
0000000  000 001 002 003 004 005 006 007 010 011 012
013 014 015 016 017
0000020  020 021 022 023
0000024
gustav@arp:~/Lispcourse 215 $
```

6) The function `with-open-file` is a syntactic sugar masking a slightly more complex operation:

```
> (with-open-stream
    (binary-numerals
      (open "binary.dat" :direction :output
            :element-type 'unsigned-byte))
```

```
    (dotimes (i 20)
      (write-byte i binary-numerals))
    (close binary-numerals))
NIL
>
```

The function `with-open-stream` can operate on more broadly defined Common Lisp streams: those can be special devices, strings, etc. The functions `open` and `close` do much the same as their Fortran and C relatives.

Reading data from external files

1) The simple reading from the keyboard can be accomplished simply by using the functions `read` and `read-line`:

```
> (read)
twas
TWAS
> (read)
´twas
(QUOTE TWAS)
> (read)
(twas brillig and the slithy toves)
(TWAS BRILLIG AND THE SLITHY TOVES)
> (read)
twas brillig and the slithy toves

TWAS
>
Clearing input from *debug-io*
>>Error: The symbol BRILLIG has no global value

SYMBOL-VALUE:
   Required arg 0 (S): BRILLIG
:C  0: Try evaluating BRILLIG again
:A  1: Abort to Lisp Top Level

-> 1
Abort to Lisp Top Level
Back to Lisp Top Level

> (read-line)
twas brillig and the slithy toves
"twas brillig and the slithy toves"
```

```
NIL
>
```

2) The functions `read` and `read-line` can take up to 4 optional arguments (*optional* means that they don't have to be used).

```
> (with-open-file (roman-numerals "roman.dat"
                                  :direction :input)
            (do ((line (read-line roman-numerals nil)
                       (read-line roman-numerals nil)))
                ((not line))
              (print line)))

"0 = 0"
"1 = I"
"2 = II"
"3 = III"
"4 = IV"
"5 = V"
"6 = VI"
"7 = VII"
"8 = VIII"
"9 = IX"
"10 = X"
"11 = XI"
"12 = XII"
"13 = XIII"
"14 = XIV"
"15 = XV"
"16 = XVI"
"17 = XVII"
"18 = XVIII"
"19 = XIX"
NIL
>
```

Observe that here we have uncovered yet another syntactic variant of `do`. This new feature is easier to see on the following example:

```
> (do ((i 0
        (1+ i)))
      ((> i 10))
    (print i))

0
```

```
1
2
3
4
5
6
7
8
9
10
NIL
>
```

The list of bindings may contain three elements, with the third one telling `do` how to ‘‘increment’’ this variable during iterations. In our file reading example we ‘‘increment’’ `line` by reading it again. We continue doing so until `read-line` returns `nil`.

3) Similarly we can read our binary data file:

```
> (with-open-file (binary-numerals "binary.dat"
                        :direction :input
                        :element-type 'unsigned-byte)
            (do ((byte
                  (read-byte binary-numerals nil)
                  (read-byte binary-numerals nil)))
                ((not byte))
              (print byte)))

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
15
16
17
18
19
NIL
>
```

Summary

1) The simplest way to print a Common Lisp datum is to use `print` or `pprint`. `pprint` stands for ‘‘pretty print’’.
2) `format` is a more elaborate facility for printing output similar to corresponding facilities in C and in Fortran. There are formatting directives for decimal, digital, octal, hexadecimal, literal, and roman outputs for integers. Formatting directives are also available for floating point numbers. Nouns can be automatically pluralised. More general formatting directives allow to print any Common Lisp symbols and data structures.
3) To create a data file and write on it use the `with-open-file` function
4) Common Lisp allows only character (text) and byte files.
5) `with-open-file` can be also used to write byte files. The `write-byte` function takes an integer as an argument and writes a byte on the output stream.
6) `with-open-file` is a syntactic sugar for the combination of `with-open-stream`, `open`, and `close`.
7) The functions `read` and `read-line` can be used to read data either from the keyboard or from files.
8) The initialisation clause of `do` can be used to tell `do` how to increment the initialised variable during iterations.
9) The functions `read` and `read-line` have facilities for detecting the end of file condition.
10) The function `read-byte` reads a byte from an input stream and returns the corresponding integer.

The closing remark

We have now covered some of the most basic features of Common Lisp. In languages such as Fortran 77, or Pascal, even C, that would be almost all that the language offers. In Common Lisp, this is only a very tiny part of the language. The real strength of Common Lisp is in symbol manipulation. We skipped entirely symbols, lists, conses. We only touched strings. We said nothing about vectors, hash tables, packages, objects, macros, methods, predicates (of which there are great many), numerous control structures apart from `if`, `do`, and `dotimes`, declarations, property lists, general sequences, error signaling. We only mentioned the compilation, we haven’t mentioned at all the CLtL2 ‘‘Loop Facility’’, conditions, series, generators, and gatherers.

But with the small part of Common Lisp presented in the first 5 lessons we are sufficiently well equipped to begin working with the Connection Machine *Lisp system.

*Lisp still suffers from many limitations of the CM2 architecture. For this reason the only parallel data structures that are allowed at present are based on numbers or characters. You cannot have `pvars` of symbols, or `pvars` of lists, or `pvars` of trees of objects. This limitation derives from the fact that the CM2 had a very large number of very slow 1-bit processors and only limited amount of memory available to each processor. The CM5 architecture is largely free of such limitations - although the presence of vector units restricts the type of data that can be placed in their slots. Still the TMC software engineers have not, as yet, released a version of *Lisp (or of any other programming language) which would be able to take a full advantage of this new architecture. On the other hand, *Lisp provides all that is available under CM Fortran and C* - largely because much less is expected of those languages - plus an interactive programming environment which those languages don't have.

The new exciting architecture of CM5 allows, in principle, for an unlimited richness of data types and structures to be deposited on every node and an unlimited number of communication styles and processes between those nodes. But, all this is much easier said than done. It is a common feature of present day MPPs that software generally lags far behind the available hardware. This is very well illustrated by the CM5, which, although of MIMD architecture offers only SIMD high level languages, with MIMD programming restricted to PVM (very highly portable) or CMMD (completely non-portable).

In the following lessons we will learn how SIMD programming differs from conventional sequential programming. The SIMD programming model is not as restrictive as it is often claimed. In the last 3 lessons of the course we will learn how quite an unstructured system of randomly scattered interacting particles can be handled within the SIMD programming paradigm.

Lesson 6
Connecting to the Connection Machine *Lisp

Defining the DISPLAY, using .Xauthority, and invoking jrun

1) In order to use *Lisp you must first login on octavia. Octavia serves as the front end to the ANU Connection Machine CM5. Octavia is a Sun 690 MP with 4 processors, 256 MB memory and a relatively modest disk space (as far as supercomputers are concerned) of some 14 GB. But Octavia doesn't talk directly to the CM5. It provides user disk space, compilation software, *Lisp simulator, CMF simulator, and job queue software. The latter is called DJM. It's been written at the Minnesota Supercomputer Center specifically for the Connection Machine. It is not a general applicability queuing software. It works only on the Connection Machine.
2) The computer which talks directly to the CM5 is caesar. It is a small SPARCstation 2 (a pizza box) hidden inside the larger Connection Machine box. That machine has very limited resources (only 64 MB of memory and 2.5 GB of disk space) and should not be accessed directly (although it can be).
2) In order to obtain a window on octavia select "`Connections`" and then "`octavia`" from the root menu on the ARP workstations - from other systems you will have to use `telnet` or `rlogin`.
3) Once you are on octavia, transfer a file `.Xauthority` from your home directory on arp, for example using `ftp` in a binary mode or `rcp` (from arp's side). Make sure that file has protection `rw------- (600).`
4) Define your X11 `DISPLAY` on octavia:

```
% setenv DISPLAY <my_machine>:0.0
```
if you use csh or tcsh, or
```
% export DISPLAY=<my_machine>:0.0
```
if you use bash, or
```
% DISPLAY=<my_machine>:0.0; export DISPLAY
```
if you use sh

5) Now you are ready to invoke `Epoch` on caesar. In order to do that you must submit an interactive job through the DJM:

```
gustav@octavia:/home/id005c/gustav 206 $ jrun -export
/usr/new/bin/epoch
Number of processors (32)?
Estimated cpu time (5min)? 30min
Estimated memory (64M)?
Job id is 46580.
[Type ~^B to disconnect, ~^C to kill.]
```

```
[Connected to caesar.anu.edu.au /dev/ttyp0.]
*** Job epoch (46580) procs 32, mem 64M, cpu 30min,
server caesar:0
*** gustav /usr/new/bin/epoch started on Sun Jan 16
10:40:36 1994
```

Notes:

a) `jrun` does not execute jobs on the Connection Machine. It executes jobs on caesar, but now every time you access the Connection Machine itself, your Connection Machine job will be accounted for.

b) you must type the full pathname of the command you want to execute on caesar

c) the switch "-export" will export all your environmental variables on octavia to caesar

Establishing the session with the Connection Machine

1) By now you should get the caesar `Epoch` window on your ARP workstation. Invoke, as before, a second window with `C-z 2`. Use your first window as your worksheet, and the second window as your Lisp window.

2) Invoke the Connection Machine *Lisp by typing `M-x run-lisp` in your second window. This time Epoch will ask you which Lisp you want to run. Answer:

```
/usr/bin/starlisp-vu-dev-f7600
```

3) At this stage you have already started *Lisp, but you are still not connected to the CM5. Type

```
> (cm:finger)
32 PN System, 28164K mem. free, 7040K VU mem. free, 1
procs, TS-10/19/93-7:41 (CMOST 7.2) Daemon up: 2 days,
1:24

USER      PID   CMPID  TIME   TEXT    ILH     ILS
IGS    IGH    VUS    VUH COMMAND
daveh    *10531 2     11:10  1360K   604K    48K
0K     0K   128K    12K  gel2
>
```

4) In order to attach yourself to the Connection Machine issue the command

```
> (*cold-boot)

;;; Not attached.  Attaching...
```

```
1024
(32 32)
10624
>
```

This command attaches you to the Connection Machine in the default mode. The *quantum* size of the machine is 1024 virtual processors. Your default configuration for `pvars` is a parallel array 32 x 32 and your process ID is 10624.

5) Now issue the command `(cm:finger)` again:

```
> (cm:finger)
32 PN System, 26888K mem. free, 6720K VU mem. free, 2
procs, TS-10/19/93-7:41 (CMOST 7.2) Daemon up: 2 days,
1:26

USER      PID   CMPID  TIME   TEXT    ILH     ILS
IGS    IGH    VUS    VUH COMMAND
daveh     10531 2     13:30  1360K   604K    48K
0K     0K   128K    12K  gel2
gustav   *10624 1      0:01   792K   144K    48K
0K     4K    64K     8K  starlisp-dhod-f7600
>
```

You should be able to see yourself attached to the Connection Machine.

6) Because of a bug in the DJM software, if you are the only user on the system, you will be charged for the wall-clock connection time, even if you don't do anything. This is a serious handicap for all users of interactive programs (this includes also the CM5 `Prism` debugger used with CMF and C*). The *Lisp remedy is to detach yourself from the machine:

```
> (cm:detach)
T
> (cm:finger)
32 PN System, 30736K mem. free, 7180K VU mem. free, 0
procs, TS-10/19/93-7:41 (CMOST 7.2) Daemon up:  0:19

>
```

7) You can define an alternative default geometry when attaching to the Connection Machine. But your new default geometry must be always a multiple of 1024 which is a quantum of the `pvar` size on our system. To request an alternative default geometry `*cold-boot` in the following way:

```
> (*cold-boot :initial-dimensions '(64 64 64))
```

```
;;; Not attached.  Attaching...

1024
(64 64 64)
11044
>
```

8) After a finished session with *Lisp on the Connection Machine you should `(quit)` *Lisp and quit your `Epoch` session. This will return you back to octavia. The DJM will print some statistics referring to your job:

```
*** Job epoch (46590) procs 32, mem 64M, cpu 5min,
server caesar:0
*** gustav /usr/new/bin/epoch started on Sun Jan 16
11:18:09 1994
*** Job terminated at Sun Jan 16 11:50:06 1994
*** exitcode = 0000
*** procs 32, mem 40.9M, cpu 5.27min
***
*** You requested 64M of memory but only used 40.9M.
*** The next time you run this job, you should request
*** less memory.  This will let your job run earlier
*** and you will get your results sooner.
gustav@octavia:/home/id005c/gustav 222 $
```

Summary

1) Octavia is the user front end to the ANU CM5
2) When connecting to octavia you should properly set up your X11 environment (`.Xauthority` and `DISPLAY`)
3) caesar is the machine which talks directly to the Connection Machine
4) To start an accounted job on caesar use `jrun`. Only accounted jobs are allowed the access to the Connection Machine.
5) The version of *Lisp to use on our Connection Machine is `/usr/bin/starlisp-vu-dev-f7600.`
6) Having started *Lisp does not imply having made a connection with the CM5. The connection is established using the command `(*cold-boot).`
7) You can see processes attached to the Connection Machine with `(cm:finger).`
8) To detach yourself from the Connection Machine use `(cm:detach).`
9) The quantum `pvar` size on the ANU CM5 is 1024. You can request an alternative default geometry by using the `:initial-dimensions` switch to the `*cold-boot` function.
10) Use `(quit)` to quit the *Lisp session (it will also detach you from the CM5 if you are still attached) and quit `Epoch` with `C-x C-c` to come back to Octavia.

11) If you are a single user on the CM5 you will be charged for the wall-clock time of the connection. This is caused by a bug in the DJM accounting system.

Lesson 7
Working with pvars

Pvars, the Connection Machine parallel data structures

1) `pvars` are like Common Lisp arrays, but this time every `pvar` element sits on its own virtual processor. There should be a *Lisp function *`make-pvar`* for creating `pvars`, to comply with Common Lisp tradition. But the part of *Lisp that lives on the Connection Machine is more like Fortran than like Lisp. It doesn't have a proper Lisp-like stack management facility and a garbage collection. Instead, *Lisp functions which operate on the CM5 use straightforward allocation and deallocation of data on the stack in the C style. To create a permanent `pvar` on the stack use:

```
> (defpvar *the-grid* 0)
*THE-GRID*
>
```

The second argument to function `defpvar` initialises every element of `*the-grid*` to `0`.

2) We can view various elements of this `pvar` with the function `ppp`:

```
> (ppp *the-grid* :mode :grid :end '(10 10))

     DIMENSION 0 (X)  ----->

0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
>
```

3) A `pvar` variable with zeroes all over is a boring thing to look at. There is a function which creates a `pvar` which has a number of the corresponding virtual processor in every slot. This function is called `self-address!!`:

```
> (ppp (self-address!!) :mode :grid :end '(10 10))
```

```
     DIMENSION 0 (X)  ----->

0 2 4 6 128 130 132 134 256 258
1 3 5 7 129 131 133 135 257 259
8 10 12 14 136 138 140 142 264 266
9 11 13 15 137 139 141 143 265 267
16 18 20 22 144 146 148 150 272 274
17 19 21 23 145 147 149 151 273 275
24 26 28 30 152 154 156 158 280 282
25 27 29 31 153 155 157 159 281 283
32 34 36 38 160 162 164 166 288 290
33 35 37 39 161 163 165 167 289 291
>
```

4) Observe that the array generated by `(self-address!!)` is laid out on the virtual processors in a somewhat peculiar manner: in blocks of 4 x 2 integers. It is very seldom that you have to be concerned with the details of this lay-out.
5) The contents of a `pvar` can be modified with the `*setf` function:

```
> (*setf *the-grid* (self-address!!))
NIL
> (ppp *the-grid* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 4 6 128
1 3 5 7 129
8 10 12 14 136
9 11 13 15 137
16 18 20 22 144
>
```

6) You may have already noticed that there are two types of *Lisp functions. The first type are functions whose names end with "`!!`". They return `pvars`. The second type are functions whose names begin with "`*`". They return scalar front end variables, but their side effects are operations performed on `pvars`.
7) There is an equivalent of Common Lisp `aref` function called `pref`, which can be used to access a specified slot in a `pvar`:

```
> (pref *the-grid* 1)
1
> (pref *the-grid* 14)
14
>
```

8) The contents of the selected slot in a `pvar` can be modified using `*setf` and `pref`:

```
> (*setf (pref *the-grid* 14) 7)
NIL
> (ppp *the-grid* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 4 6 128
1 3 5 7 129
8 10 12 7 136
9 11 13 15 137
16 18 20 22 144
>
```

9) The function `cube-from-grid-address` can be used to translate the default grid coordinates to the corresponding processor number:

```
> (cube-from-grid-address 0 0)
0
> (cube-from-grid-address 0 1)
1
> (cube-from-grid-address 1 0)
2
> (cube-from-grid-address 1 1)
3
>
```

10) In combination with `pref` and `*setf` the function `cube-from-grid-address` allows us to modify any entry in `*the-grid*` specified using grid coordinates:

```
> (*setf (pref *the-grid* (cube-from-grid-address 3 2))
0)
NIL
> (ppp *the-grid* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 4 6 128
1 3 5 7 129
8 10 12 0 136
9 11 13 15 137
16 18 20 22 144
>
```

11) It is very easy to carry out arithmetic operations on `pvars`:

```
> (defpvar lots-of-sevens 7)
LOTS-OF-SEVENS
> (defpvar lots-of-threes 3)
LOTS-OF-THREES
> (ppp lots-of-sevens :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

7 7 7 7 7
7 7 7 7 7
7 7 7 7 7
7 7 7 7 7
7 7 7 7 7
> (ppp (+ lots-of-sevens lots-of-threes) :mode :grid
:end '(5 5))

     DIMENSION 0 (X)  ----->

10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
10 10 10 10 10
> (defpvar proc-address (self-address!!))
PROC-ADDRESS
> (defpvar -proc-address (* -1 proc-address))
-PROC-ADDRESS
> (ppp (+ proc-address -proc-address) :mode :grid :end
'(5 5))

     DIMENSION 0 (X)  ----->

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
>
```

12) The function `random!!` generates a random `pvar`:

```
> (ppp (random!! 10) :mode :grid :end '(5 5))
```

```
     DIMENSION 0 (X)  ----->

0 4 3 8 3
8 6 4 5 4
9 5 6 0 5
0 6 6 9 8
4 1 8 0 4
> (ppp (random!! 10) :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

8 4 9 3 3
8 2 2 4 1
8 8 9 9 1
2 3 2 8 2
1 2 5 8 8
>
```

13) Many other mathematical functions are provided in a parallelised form: `sin!!, sinh!!, cos!!, tan!!, exp!!, expt!!, sqrt!!:`

```
> (ppp (sin!! (random!! 10)) :mode :grid :end '(3 3))

     DIMENSION 0 (X)  ----->

0.8414709568023682 0.9893582463264465
0.8414709568023682
-0.756802499294281 0.14112000167369843
-0.279415488243103
0.6569865942001343 -0.9589242935180664
-0.9589242935180664
>
```

14) With the function `cm:time` you can time your CM5 operations:

```
> (cm:time (sqrt!! (random!! 100.0)))

 Warning: Turning off Paris safety for CM:TIME.
 Warning: Turning off *Lisp Interpreter Safety for
CM:TIME.
Evaluation of (SQRT!! (RANDOM!! 100.0)) took 0.057098
seconds of elapsed time,
during which the CM was active for 0.002021 seconds or
3.00% of the total elapsed time.
```

```
>
```

Here you can see that it took only 0.002 seconds to generate 1024 random number and take a square root from each of them. For all practical purposes you can assume that the Connection Machine carries out up to 1024 operations simultaneously.

The NEWS operations

1) The `news!!` function is used to shift `pvars` in the main directions. On the borders of the `pvars` `news!!` will wrap data around:

```
> (ppp *the-grid* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 4 6 128
1 3 5 7 129
8 10 12 14 136
9 11 13 15 137
16 18 20 22 144
> (ppp (news!! *the-grid* 1 0) :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

2 4 6 128 130
3 5 7 129 131
10 12 14 136 138
11 13 15 137 139
18 20 22 144 146
> (ppp (news!! *the-grid* -1 0) :mode :grid :end '(5
5))

     DIMENSION 0 (X)  ----->

902 0 2 4 6
903 1 3 5 7
910 8 10 12 14
911 9 11 13 15
918 16 18 20 22
> (ppp (news!! *the-grid* 0 1) :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1 3 5 7 129
```

```
8 10 12 14 136
9 11 13 15 137
16 18 20 22 144
17 19 21 23 145
> (ppp (news!! *the-grid* 0 -1) :mode :grid :end '(5
5))

     DIMENSION 0 (X)  ----->

121 123 125 127 249
0 2 4 6 128
1 3 5 7 129
8 10 12 14 136
9 11 13 15 137
>
```

2) On one-dimensional `pvars` `news!!` can be used too. In that case it will merely shift data left or right:

```
> (*cold-boot :initial-dimensions '(1024))

;;; Not attached.  Attaching...

1024
(1024)
12083
> (defpvar *the-row* (self-address!!))
*THE-ROW*
> (ppp *the-row* :end 10)
0 1 2 3 4 5 6 7 8 9
> (ppp (news!! *the-row* 1) :end 10)
1 2 3 4 5 6 7 8 9 10
> (ppp (news!! *the-row* -1) :end 10)
1023 0 1 2 3 4 5 6 7 8
> (ppp (news!! *the-row* 3) :end 10)
3 4 5 6 7 8 9 10 11 12
>
```

3) Using the wrap-around NEWS facility is not very useful if you must take care of fixed boundary conditions. There is another function which is very useful in that context: `news-border!!` Here is how it works:

```
> (defpvar *my-grid* (self-address!!))
*MY-GRID*
```

```
> (ppp *my-grid* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 4 6 128
1 3 5 7 129
8 10 12 14 136
9 11 13 15 137
16 18 20 22 144
> (ppp (news-border!! *my-grid* (!! -1) 0 -1) :mode
:grid :end '(5 5))

     DIMENSION 0 (X)  ----->

-1 -1 -1 -1 -1
0 2 4 6 128
1 3 5 7 129
8 10 12 14 136
9 11 13 15 137
> (ppp (news-border!! *my-grid* (!! -1) -1 0) :mode
:grid :end '(5 5))

     DIMENSION 0 (X)  ----->

-1 0 2 4 6
-1 1 3 5 7
-1 8 10 12 14
-1 9 11 13 15
-1 16 18 20 22
>
```

The second argument to `news-border!!` is called a `border-pvar`. The `news-border!!` function slips in the values from the `border-pvar` in places where NEWS would make references off the grid.

4) How to construct a `border-pvar` which would be more elaborate than just a one number matrix? The most general although not necessarily the simplest way to make such a `pvar` is to make it first on the front end and then to transfer it onto the Connection Machine. This is a slow operation, but if you need to make a particular `border-pvar` only once, the cost will be affordable. Even the most complex `border-pvars` can be made like that.

```
> (setf *print-array* nil)
NIL
> (setf the-front-end-array
```

```
    (make-array (list (* 32 32)) :initial-element 0))
#<Simple-Vector T 1024 1B9B3D6>
> (dotimes (i 1024)
    (let ((row (grid-from-cube-address i 1))
          (column (grid-from-cube-address i 0)))
      (if (= row 0)
          (setf (aref the-front-end-array i) -1))))
NIL
> (defpvar *my-mask* 0)
*MY-MASK*
> (array-to-pvar the-front-end-array *my-mask*)
#<FIXNUM-PVAR *MY-MASK*, 7340032 Heap, 4 bytes,
*DEFAULT-VP-SET* (32 32) (mutable)>
> (ppp *my-mask* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

-1 -1 -1 -1 -1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
>
```

5) If the `border-pvar` has a simple structures as in the example above, then it can be constructed more efficiently using the `news-border!!` function itself.

```
> (ppp (news-border!! (!! 0) (!! -1) 0 -1) :mode :grid
:end '(5 5))

     DIMENSION 0 (X)  ----->

-1 -1 -1 -1 -1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
>
```

Solving the Laplace equation in 2 dimensions on the Connection Machine

1) Consider the following problem. A square metal plate has one of its edges kept at 100 K whereas all other edges are cooled to 0 K. Evaluate the final temperature distribution in the plate. This is a very simple problem which is described by the Laplace equation. The solution can be obtained by the following iteration procedure. Start with any sensible initial condition for the whole plate. In each iterative step we replace the value at each point by the average over its

neighbourhood. We repeat this procedure as long as we observe changes in temperature distribution.

2) Type the following short program in your worksheet window.

```
(setq *compilep* nil)

(defpvar *border-pvar* 0.0)
(*setf *border-pvar* (news-border!!
                        *border-pvar* (!! 1.0) 0 -1))
(*setf *border-pvar*
   (*!! (news-border!! *border-pvar* (!! 0.0) 1 0)
        (news-border!! *border-pvar* (!! 0.0) -1 0)
        100.0))
(defpvar *the-plate* (!! 0.0))

(defun iteration-step ()
  "perform averaging over the neighbourhood
for each point of *the-plate*"
  (*setf *the-plate*
     (/!!
       (+!! (news-border!! *the-plate* *border-pvar*
                           1 0)
            (news-border!! *the-plate* *border-pvar*
                           0 1)
            (news-border!! *the-plate* *border-pvar*
                           -1 0)
            (news-border!! *the-plate* *border-pvar*
                           0 -1))
       (!! 4.0)))
  (ppp *the-plate* :mode :grid :end '(5 5)))
```

The first command `(set *compilep* nil)` turns off the automatic *Lisp compiler. The compiler is useful only when you work on the final production code. If it is not switched off during prototyping it can result in incorrect arithmetics. The compiler must be used only with full type declarations for all variables. This can be done in *Lisp in a way similar to Fortran or C, but this is best done at the end when the algorithm has been thoroughly prototyped and tested. We will not talk about *Lisp compiler in this course.

3) To run the program switch to the Lisp window and type:

```
> (*cold-boot)
1024
(32 32)
```

```
12633
> (load "laplace.l")
;;; Loading source file "laplace.l"
;;; Warning: File "laplace.l" does not begin with IN-
PACKAGE.  Loading into package "*LISP5"
>  (iteration-step)

     DIMENSION 0 (X)  ----->

0.0 25.0 25.0 25.0 25.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
> (iteration-step)

     DIMENSION 0 (X)  ----->

6.25 31.25 37.5 37.5 37.5
0.0 6.25 6.25 6.25 6.25
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
> (iteration-step)

     DIMENSION 0 (X)  ----->

7.8125 37.5 43.75 45.3125 45.3125
3.125 9.375 12.5 12.5 12.5
0.0 1.5625 1.5625 1.5625 1.5625
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
>
```

4) If you become bored with typing `(iteration-step)` by hand, use `dotimes`:

```
> (dotimes (i 30) (iteration-step))
...
```

This will produce a rather voluminous output, which will show you how the boundary condition spreads through the plate.

Summary

1) Permanent `pvars` are allocated on the CM5 stack by the function `defpvar`. This function can be also used to initialiase `pvars`.

2) `Pvars` can be viewed with function `ppp`.
3) The function `self-address!!` generates a `pvar` which has the number of the corresponding virtual processor in each slot.
4) Values can be assigned to allocated `pvars` by using the function `*setf`. This function, unlike its Common Lisp counterpart, will not create a variable if such doesn't exist yet.
5) There are two types of `*Lisp` functions: functions whose names end with "`!!`" return `pvars`; functions whose names begin with "`*`" modify `pvars` as a side effect, but return scalar quantities.
6) Specified entries in `pvars` can be accessed with function `pref`.
7) The function `cube-from-grid-address` converts grid coordinates into appropriate virtual processor number
8) Common Lisp arithmetic operators have their "`!!`" equivalents in *Lisp.
9) The execution of a *Lisp function can be timed with `cm:time`. For most practical purposes you can assume that the ANU Connection Machine carries out up to 1024 operations simultaneously at each execution step.
10) The `news!!` function is used to shift a `pvar` in basic directions of the grid.
11) The function `news-border!!` shifts a `pvar` like the function `news!!` but instead of wrapping a `pvar` around it slips values from a `border-pvar` at the boundary.
12) Arrays made on the front end can be transferred onto the Connection Machine with the function `array-to-pvar`.
13) The function `grid-from-cube-address` can be used to find the location in grid coordinates of a specified virtual processor.
14) The function `news-border!!` itself can be used effectively to produce simple `border-pvars`.
15) The predefined global variable `*compilep*` should be set to `nil` during prototyping. The compiled code will return incorrect results unless accompanied by correct variable type definitions.

Lesson 8

*Graphics

This is a very short lesson, because *Lisp's graphics are very easy to use. There is only one caveat. You must not trust unreservedly the old documentation. Once upon a time there was a separate *Graphics package which had to be loaded explicitly. This package is still around, for some reason, but it can crash the system. If the automatic compilation is activated then *Graphics functions will be automatically converted to *LISP5-I:: functions. The latter can be also called by hand and they work just fine.

1) The program below shows the simplest way to display the images of `pvars` on an X11 display.

```
;;
;; (setq *compilep* nil)
;;
(*cold-boot :initial-dimensions '(128 128))

(defpvar *border-pvar* 0.0 single-float)
(*setf *border-pvar*
   (news-border!! *border-pvar* (!! 1.0) 0 -1))
(*setf *border-pvar*
   (*!! (news-border!! *border-pvar* (!! 0.0) 1 0)
        (news-border!! *border-pvar* (!! 0.0) -1 0)
        100.0))
(defpvar *the-plate* 0.0 single-float)

(defun iteration-step (&key (silent nil))
  "perform averaging over the neighbourhood for each
point of *the-plate*"
  (*setf *the-plate*
     (/!! (+!! (news-border!!
                  *the-plate* *border-pvar* 1 0)
               (news-border!!
                  *the-plate* *border-pvar* 0 1)
               (news-border!!
                  *the-plate* *border-pvar* -1 0)
               (news-border!!
                  *the-plate* *border-pvar* 0 -1))
          (!! 4.0)))
  (if (not silent)
```

```
      (ppp *the-plate* :mode :grid :end '(5 5))))

(setf *current-display-window*
  (*lisp5-i::create-display-window
    :display-padding 0
    :desired-width 128
    :desired-height 128
    :color-map :gray-and-rainbow))

(defun view-heat-spread (number-of-steps)
  (declare (type fixnum number-of-steps))
  (dotimes (i number-of-steps)
    (iteration-step :silent t)
    (*lisp5-i::display-image *the-plate*
       :overlay (list nil!!)
       :color-range :rainbow)))
```

There are several new elements in this program so we have to stop for a while and discuss those.

a) This time we are activating the automatic *Lisp compiler (this is the *Lisp default). In order to get correct results we must declare the types of the variables. For example:

```
(defpvar *border-pvar* 0.0 single-float)
```

b) We have introduced also the `&key` parameter in the definition of the `iteration-step`:

```
(defun iteration-step (&key (silent nil))
```

When a function argument is declared this way, it's value is by default "`nil`", and it can be changed by calling

```
(iteration-step :silent t)
```

c) The function `*lisp5-i::create-display-window` creates the display window. It can be called without any arguments at all, in which case it would use some default values and enquire interactively about the rest.

d) Once the window is up we can begin sending images of `pvars` to it. This is done with the function `*lisp5-i::display-image`. The function `view-heat-spread`, which invokes `*lisp5-i::display-image` after every call to the `iteration-step` has one argument. The type of this argument is declared by

```
(declare (type fixnum number-of-steps))
```

2) Load this program into your *Lisp process. Note that the program `*cold-boots` automatically. You should also get the empty *Graphics window. Now try the following:

```
> (view-heat-spread 10)
NIL
> (dotimes (i 500)
    (iteration-step :silent t))
NIL
> (view-heat-spread 5)
NIL
>
```

3) The window can be cleaned with

```
> (*lisp5-i::clear-display-window)
NIL
>
```

and deleted with

```
> (*lisp5-i::delete-display-window *current-display-
window*)
T
>
```

Remember that the `*current-display-window*` is the name of the variable which was bound to the window structure by the invocation to `setf` and `*lisp5-i::create-display-window`.

You must admit that it is hard to think of a more programmer friendly system for displaying data.

4) It is easy to make composite pictures. The following function will create three snapshots and place them side by side within the display window.

```
(defun three-snapshots ()
  (if (*lisp5-i::valid-display-window-p
         *current-display-window*)
      (*lisp5-i::delete-display-window
         *current-display-window*))
  (setf *current-display-window*
     (*lisp5-i::create-display-window
```

```
        :display-padding 0
        :desired-width (* 3 128)
        :desired-height (* 128)
        :color-map :gray-and-rainbow))
    (dotimes (i 3)
      (dotimes (j 300)
        (iteration-step :silent t))
      (*lisp5-i::display-image *the-plate*
                               :color-range :rainbow)))
```

We have used here one more new function `*lisp5-i::valid-display-window-p`. This function returns "`t`" (true) if there exist an already created window. If such a window exists, we delete it and then create a window with sufficient size for three snapshots.

5) *Graphics package allows for more elaborate operations. For example you can manually assemble several displays in the display window. In the section above, function `*lisp5-i::display-image` did it automatically. You can manipulate and define your own colour maps (remember, they are called ‘‘color maps’’ in American English), use three dimensional rendering functions, some simple geometrical transformations, etc. But in general terms the *Graphics package is very simple. The number of facilities made available to the researcher are just sufficient to view the data without having to worry about X11 details, mouse manipulations, etc. If you need to do something much more complex than that, e.g., define push-buttons, scrollbars, etc, you may have to combine front-end Common Lisp operations and some Common Lisp X11-toolkit packages with your program. There is a number of those available. On Suns, LispView is the easiest to use and the fastest. But you should remember that Lucid Common Lisp is not very good for doing such things: it lacks dynamic libraries with the result that Lucid CL processes tend to use an insane amount of memory. They also tend to be unresponsive if they grow too large.

Summary

1) The use of automatic *Lisp compiler requires type declarations for all `pvars`.
2) Types can be declared for `pvars` during creation and initialisation with `defpvar`.
3) Common Lisp allows to define some function arguments as `&key` parameters.
4) An X11 *Graphics display window is created with `*lisp5-i::create-display-window`.
5) If you have automatic *Lisp compilation turn on, you can also use abbreviation "`*g:`" in place of "`*lisp5-i::`". But this may lead to hanging *Lisp if automatic compilation is turned off.
6) Variable types in Common Lisp function definitions are declared with the `declare` statement which should follow the list of function variables.
7) `pvar` images can be displayed on *Graphics display windows with the `*lisp5-i::display-image` function.

8) The display window can be cleared with the `*lisp5-i::clear-display-window` function and deleted with the `*lisp5-i::delete-display-window` function.
9) The predicate (a function which returns a logical value) `*lisp5-i::valid-display-window-p` can be used to check if a valid display window already exists.
10) The function `*lisp5-i::display-image` can be also used to display images side by side.

Lesson 9

Masking

Masking operations allow to enable and disable groups of virtual processors. Masking is performed with boolean `pvars`. These can be generated using a variety of *Lisp forms which yield boolean `pvars` on evaluation. Most predicates in Common Lisp end with letter "`p`", e.g., `evenp, oddp, listp`, etc. Arithmetic predicates have their "`!!`" equivalents in *Lisp.

Parallel predicate functions

1) The function `*when` allows to carry out operations in parallel on selected groups of processors:

```
> (defpvar *procnum-pvar* (self-address!!))
*PROCNUM-PVAR*
> (defpvar *even-mask* (evenp!! *procnum-pvar*))
*EVEN-MASK*
> (ppp *even-mask* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

T T T T T
NIL NIL NIL NIL NIL
T T T T T
NIL NIL NIL NIL NIL
T T T T T
> (defpvar *some-such-pvar* 0)
*SOME-SUCH-PVAR*
> (*when *even-mask*
    (*setf *some-such-pvar*
           (sin!! (/!! (!! pi) (!! 2)))))
NIL
> (ppp *some-such-pvar* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1.0 1.0 1.0 1.0 1.0
0 0 0 0 0
1.0 1.0 1.0 1.0 1.0
```

```
0 0 0 0 0
1.0 1.0 1.0 1.0 1.0
>
```

The Common Lisp "`when`" function is like "`if`" without the "`else`" clause. `*when` capitalises on this property and provides the most straightforward selection and deselection of processors. Here is the demonstration of the Common Lisp `when` function.

```
> (when t
    (print "hello world"))

"hello world"
"hello world"
> (when nil
    (print "do not print anything"))
NIL
>
```

2) `*if` is like `*when` but it has the `else` clause too:

```
> (*if *even-mask*
       (*setf *some-such-pvar*
              (sin!! (/!! (!! pi) (!! 2))))
     (*setf *some-such-pvar*
            (sin!! (/!! (!! (- pi)) (!! 4)))))
NIL
> (ppp *some-such-pvar* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1.0 1.0 1.0 1.0 1.0
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304
1.0 1.0 1.0 1.0 1.0
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304
1.0 1.0 1.0 1.0 1.0
>
```

3) There is also the `if!!` version which can be used as follows:

```
> (ppp
```

```
    (if!! *even-mask*
          (sin!! (/!! (!! pi) (!! 2)))
       (sin!! (/!! (!! (- pi)) (!! 4))))
    :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1.0 1.0 1.0 1.0 1.0
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304
1.0 1.0 1.0 1.0 1.0
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304 -0.7071067690849304
-0.7071067690849304
1.0 1.0 1.0 1.0 1.0
>
```

4) The function `*unless` is the opposite of `*when`.
5) There is also the `*` version of the Common Lisp `case` function:

```
> (defpvar *another-pvar* 1)
*ANOTHER-PVAR*
>  (*case (mod!! (self-address!!) (!! 4))
      (0     (*setf *another-pvar* (!! 0)))
      ((1 2) (*setf *another-pvar* (self-address!!)))
      (3     (*setf *another-pvar* (!! -1))))
NIL
> (ppp *another-pvar* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 0 6 0
1 -1 5 -1 129
0 10 0 14 0
9 -1 13 -1 137
0 18 0 22 0
>
```

and its `!!` equivalent:

```
> (ppp
    (case!! (mod!! (self-address!!) (!! 4))
       (0     (!! 0))
```

```
        ((1 2) (self-address!!))
        (3     (!! -1)))
    :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0 2 0 6 0
1 -1 5 -1 129
0 10 0 14 0
9 -1 13 -1 137
0 18 0 22 0
>
```

6) *when can be used in combination with grid-from-cube-address!! and self-address!! to set boundary conditions:

```
> (*setf *some-such-pvar* (!! 0))
NIL
> (*when (=!! (grid-from-cube-address!!
                (self-address!!) (!! 1)) (!! 0))
    (*setf *some-such-pvar* (!! 1)))
NIL
> (ppp *some-such-pvar* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1 1 1 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
>
```

7) In order to define different boundary conditions at two boundaries in one evaluation one can use *case combined with grid-from-cube-address!! and self-address!!

```
> (*case (grid-from-cube-address!!
           (self-address!!) (!! 1))
      (0  (*setf *some-such-pvar* (!! 1)))
      (31 (*setf *some-such-pvar* (!! -1))))
NIL
> (ppp *some-such-pvar* :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->
```

```
1 1 1 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
> (ppp *some-such-pvar* :mode :grid
       :start '(0 27) :end '(5 32))

     DIMENSION 0 (X)  ----->

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
-1 -1 -1 -1 -1
>
```

8) Finally, there is the parallel equivalent of the Common Lisp `cond` function. In the example below we use `cond!!` combined with `random!!` and `sin!!` to generate a random sequence of heads and tails:

```
> (defpvar *random-pvar* (random!! (* 2 pi)))
*RANDOM-PVAR*
> (*setf *random-pvar* (sin!! *random-pvar*))
NIL
> (ppp *random-pvar* :end 10)
-0.6397389769554138 -0.21332737803459168
0.3066985011100769 -0.9545149207115173
-0.32147568464279175 -0.39945465326309204
0.5027628540992737 0.4262485206127167
-0.11847128719091416 0.9911273121833801
> (ppp
    (cond!!
      ((minusp!! *random-pvar*) (!! -1))
      ((plusp!! *random-pvar*) (!! 1))
      (t!! (!! 0)))
    :end 20)
-1 -1 1 -1 -1 -1 1 1 -1 1 1 -1 1 1 -1 -1 1 -1 -1 1
>
```

Summary

1) There are several functions for selecting and deselecting groups of processors. These are parallel generalisations of Common Lisp predicates. The most important ones are: `*when`, `*unless`, `*if`, `if!!`, `*case`, `case!!`, `*cond`, and `cond!!`

2) `*cond` or `*case` in combination with `grid-from-cube-address!!` can be used to operate on selected rows or columns of rectangular `pvars`.
3) Other useful predicates are `evenp!!`, `oddp!!`, `minusp!!`, `plusp!!`. A combination of `random!!`, `sin!!`, `minusp!!`, and `plusp!!` was used in the last example to generate a random sequence of heads and tails.

Lesson 10
Segmented Scans

What is a segmented scan operation?

1) Connect to the CM-5 and define the default geometry: one-line, 2^{10} entries long

```
> (*cold-boot :initial-dimensions
              (list (expt 2 10)))

;;; Not attached.  Attaching...

1024
(1024)
25554
>
```

2) Define a simple one dimensional parallel variable:

```
> (defpvar *lots-of-ones* (!! 1) fixnum)
*LOTS-OF-ONES*
>
```

3) We will now attempt to chop that variable into segments and perform operations on it which will be localised within the segments. Begin with defining the segmentation mask:

```
> (defpvar *segment-mask* (!! nil) boolean)
*SEGMENT-MASK*
> (ppp *segment-mask* :start 3 :end 10)
NIL NIL NIL NIL NIL NIL NIL
>
```

This is not much of a segmentation mask since it doesn't have any tees `(T)` in it. We can insert some `tees` using the `mod!!` operation:

```
> (ppp (mod!! (self-address!!) (!! 7)) :end 20)
0 1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2 3 4 5
> (ppp (=!! (mod!! (self-address!!) (!! 7)) (!! 0))
       :end 20)
T NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL T
NIL NIL NIL NIL NIL
>
```

4) The latter looks like a good segmentation mask. We will save it on our boolean *segment-mask*:

```
> (*setf *segment-mask*
         (=!! (mod!! (self-address!!) (!! 7))
              (!! 0)))
NIL
> (ppp *segment-mask* :end 20)
T NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL T
NIL NIL NIL NIL NIL
>
```

5) With this segmentation mask we can chop our fixnum pvar * lots-of-ones* to pieces and perform various operations on the chunks of it. For example we can sum up the contents of the segments.

```
> (ppp (scan!! *lots-of-ones* '+!!
               :segment-pvar *segment-mask*
               :include-self t) :end 20)
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6
>
```

6) We can also perform the same operation backwards:

```
> (ppp (scan!! *lots-of-ones* '+!!
               :segment-pvar *segment-mask*
               :direction :backward
               :include-self t)
       :end 20)
1 7 6 5 4 3 2 1 7 6 5 4 3 2 1 7 6 5 4 3
>
```

7) What is the difference between the above and

```
> (ppp (scan!! *lots-of-ones* '+!!
               :segment-pvar *segment-mask*
               :direction :backward
               :include-self t
               :segment-mode :segment) :end 20)
7 6 5 4 3 2 1 7 6 5 4 3 2 1 7 6 5 4 3 2
>
```

Have a close look, again at our segmentation mask

```
> (ppp *segment-mask* :end 20)
```

```
T NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL T
NIL NIL NIL NIL NIL
>
```

8) Let us save the result of summing up the numbers in the segments on a new variable

```
> (defpvar *one-to-seven*
      (scan!! *lots-of-ones* '+!!
              :segment-pvar *segment-mask*
              :include-self t) fixnum)
*ONE-TO-SEVEN*
> (ppp *one-to-seven* :end 20)
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6
>
```

9) The segmented scan operation can be also used to spread elements from the processor marked with `T` over the remaining processors belonging to the segment. In this example we spread `1` over whole segments:

```
> (ppp (scan!! *one-to-seven* 'copy!! :segment-pvar
*segment-mask*
            :include-self t) :end 20)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
>
```

10) We can also spread in the backward direction:

```
> (ppp (scan!! *one-to-seven* 'copy!!
               :segment-pvar *segment-mask*
               :include-self t
               :direction :backward
               :segment-mode :segment) :end 20)
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
>
```

11) By introducing a new very finely grained segmentation mask (`t!!`, i.e. the Tees all over the place) we can immitate the NEWS operations with `scan!!`

```
> (ppp (scan!! *one-to-seven* 'copy!!
               :segment-pvar t!! :include-self t)
       :end 20)
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6
> (ppp (scan!! *one-to-seven* 'copy!!
               :segment-pvar t!!
```

```
                :include-self nil) :end 20)
0 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5
>
```

Observe that the first attempt didn't work, whereas the second one did. Explain. Also observe how in on the second attempt `scan!!` automatically rolled in 0 "from beyond the edge of the Universe". We can also do NEWS to the left:

```
> (ppp (scan!! *one-to-seven* 'copy!!
                :segment-pvar t!!
                :include-self nil
                :direction :backward)
       :end 20)
2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7
>
```

The segmented-news!! operation

1) The operation demonstrated above shifted the whole `pvar` to the right or to the left. How can we define a segmented NEWS operation, i.e., an operation which would perform *rotations* with wrap-around within segments? Let us concentrate on forward NEWS first. To achieve a segmented rotation we need to pick the item which ends every segment, save it, and insert back at the beginning of every segment after rotation of the whole `pvar` was accomplished.
2) We can spread that last item in each segment using `:backward 'copy!!` and `:segment-mode :segment`:

```
> (defpvar *proc-numbers* (self-address!!) fixnum)
*PROC-NUMBERS*
> (ppp (scan!! *proc-numbers* 'copy!!
                :segment-pvar *segment-mask*
                :direction :backward
                :segment-mode :segment)
       :end 20)
6 6 6 6 6 6 6 13 13 13 13 13 13 13 20 20 20 20 20 20
> (ppp *proc-numbers* :end 20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
> (ppp *segment-mask* :end 20)
T NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL NIL T
     NIL NIL NIL NIL NIL
>
```

Since this works, it is worth saving:

```
> (defpvar *temp-store* 0 fixnum)
```

```
*TEMP-STORE*
> (*setf *temp-store*
      (scan!! *proc-numbers* 'copy!!
           :segment-pvar *segment-mask*
           :direction :backward :segment-mode
:segment))
NIL
> (ppp *temp-store* :end 20)
6 6 6 6 6 6 6 13 13 13 13 13 13 13 20 20 20 20 20 20
>
```

3) Now we can rotate the whole `pvar *proc-numbers*` to the right as before

```
> (defpvar *shifted-proc-numbers* 0 fixnum)
*SHIFTED-PROC-NUMBERS*
>
> (*setf *shifted-proc-numbers*
     (scan!! *proc-numbers* 'copy!!
             :segment-pvar t!!
             :include-self nil))
NIL
> (ppp *shifted-proc-numbers* :end 20)
0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
>
```

4) and finally we insert numbers from `*temp-store*` into `*shifted-proc-number*` in places marked by `tees` of the `*segment-mask*`:

```
> (*when *segment-mask*
    (*setf *shifted-proc-numbers* *temp-store*))
NIL
> (ppp *proc-numbers* :end 20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
> (ppp *shifted-proc-numbers* :end 20)
6 0 1 2 3 4 5 13 7 8 9 10 11 12 20 14 15 16 17 18
>
```

5) Observe that somehow, quite automatically, the last segment which is shorter than other segments also took care of itself correctly:

```
> (ppp *shifted-proc-numbers* :start 1010 :end 1024)
1009 1010 1011 1012 1013 1021 1015 1016 1017 1018 1019
1020 1023 1022
>
```

6) Above we experimented with the ways of implementing rotation within segments. Now it's time to wrap it all up into a function. And here it is:

```
(defun segmented-news!! (data-pvar segment-mask)
  (*let ((temp-store (scan!! data-pvar 'copy!!
                        :segment-pvar segment-mask
                        :direction :backward
                        :segment-mode :segment))
         (shifted-pvar (scan!! data-pvar 'copy!!
                          :segment-pvar t!!
                          :include-self nil)))
    (if!! segment-mask
          temp-store
        shifted-pvar)))
```

By the way, looking at the `if!!` clause, can you explain why there is no `when!!` function in *Lisp? (In fact there is almost a `when!!` function. The function `if!!` can be used without the `else` clause. In that case that clause defaults to `nil!!`.)

7) Here is how it works:

```
> (ppp *segment-mask* :end 20)
T NIL NIL NIL NIL NIL NIL T NIL NIL NIL NIL NIL
NIL T NIL NIL NIL NIL NIL
> (ppp *proc-numbers* :end 20)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
> (ppp (segmented-news!! *proc-numbers* *segment-mask*)
     :end 20)
6 0 1 2 3 4 5 13 7 8 9 10 11 12 20 14 15 16 17 18
>
```

8) We can add a key-word `:border-pvar` to our `segmented-news!!` definition. When used with the `:border-pvar` switch the function would slip in the values from the `border-pvar` into locations marked by "`T`" in the `segment-mask` instead of wrapping around the segments. In this way we also add `news-border!!` functionality to our `segmented-news!!`:

```
(defun segmented-news!! (data-pvar segment-mask
                         &key (border-pvar nil))
  (*let ((temp-store (if!! border-pvar
                           border-pvar
                         (scan!! data-pvar 'copy!!
                           :segment-pvar segment-mask
                           :direction :backward
                           :segment-mode :segment)))
```

```
        (shifted-pvar (scan!! data-pvar 'copy!!
                              :segment-pvar t!!
                              :include-self nil)))
    (if!! segment-mask
          temp-store
        shifted-pvar)))
```

9) Observe that this new function works as before if used *without* the `:border-pvar` switch, and works as we have planned when the `:border-pvar` switch *is* used:

```
> (ppp (segmented-news!! *proc-numbers* *segment-mask*)
:end 20)
6 0 1 2 3 4 5 13 7 8 9 10 11 12 20 14 15 16 17 18
> (ppp (segmented-news!! *proc-numbers* *segment-mask*
:border-pvar (!! -1))
       :end 20)
-1 0 1 2 3 4 5 -1 7 8 9 10 11 12 -1 14 15 16 17 18
> (ppp (segmented-news!!
         *proc-numbers* *segment-mask*
         :border-pvar (!! -1))
       :start 1010 :end 1024)
1009 1010 1011 1012 1013 -1 1015 1016 1017 1018 1019
1020 -1 1022
>
```

10) **Exercise**: expand the definition of `segmented-news!!` further to allow also for backward rotation.

Summary

1) Function `scan!!` can perform accumulative operations on segmented linear `pvars` with segments defined by a segmentation mask.
2) `scan!!` operations can be performed in `:forward` (default) and `:backward :directions`.
3) the first element of the segment can be included or dropped.
4) For `:backward` scans the `:segment-mode :segment` applies segment marking compliant with the `:forward` direction.
5) Apart from accumulative arithmetic operations function `scan!!` can also perform the `copy!!` operation. This operation spreads data from the beginning of the segment over the whole segment.
6) The `scan!!` operation can be also used to simulate the NEWS shifts on 1D `pvars`.
7) The `segmented-news!!` function which we have defined ourselves performs rotations with wrap-around within segments or slips in the desired border `pvar` into locations marked by `T` in the segmentation mask.

Lesson 11
*Lisp Structures

Modeling and generating particles

1) We begin by introducing a particle structure

```
> (*defstruct particle
  (x 0.0 :type single-float)
  (y 0.0 :type single-float)
  (x0 0.0 :type single-float)
  (y0 0.0 :type single-float)
  (work1 0.0 :type single-float)
  (work2 0.0 :type single-float)
  (x-bin 0 :type fixnum)
  (y-bin 0 :type fixnum)
  (box-mark nil :type boolean)
  (id 0 :type fixnum)
  (processor 0 :type fixnum))
;;; You are using the compiler in DEVELOPMENT mode
(compilation-speed = 3)
;;; If you want faster code at the expense of longer
compile time,
;;; you should use the production mode of the compiler,
which can be obtained
;;; by evaluating (lisp:proclaim '(optimize
(compilation-speed 0)))
;;; Generation of full safety checking code is enabled
(safety = 3)
;;; Optimization of tail calls is disabled (speed = 2)
PARTICLE
>
```

The call to `*defstruct` automatically triggers the compiler which compiles dynamically generated accessor and structure creation functions.

2) Define a parallel variable which has the structure `particle` sitting on every (virtual) processor.

```
> (defpvar particles (make-particle!!) particle)
PARTICLES
> (pref particles 173)
#S(PARTICLE X 0.0 Y 0.0 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
```

```
> (ppp particles :start 111 :end 115)
#S(PARTICLE X 0.0 Y 0.0 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0
Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 0.0 Y 0.0 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0
Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 0.0 Y 0.0 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0
Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 0.0 Y 0.0 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0
Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
>
```

3) We can easily access any slot of this structure by invoking the corresponding accessor function

```
> (pref (particle-x0!! particles) 125)
0.0
```

4) Let us now scatter all particles at random within a square 100 x 100

```
> (setq x-width 100.0)
100.0
> (setq y-width 100.0)
100.0
> (*setf (particle-x!! particles)
         (random!! (!! x-width)))
NIL
> (*setf (particle-y!! particles)
         (random!! (!! y-width)))
NIL
> (ppp particles :start 111 :end 115)
#S(PARTICLE X 16.562498092651367 Y 69.52366638183594 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 3.6271214485168457 Y 64.846923828125 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 52.86708068847656 Y 26.74432945251465 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 32.238006591796875 Y 95.36654663085938 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)

>
```

5) We should also assign a distinct id number to each particle

```
> (*setf (particle-id!! particles)
         (self-address!!))
NIL
```

```
> (ppp particles :end 5)
#S(PARTICLE X 6.005513668060303 Y 19.47820281982422 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 0 PROCESSOR 0)
#S(PARTICLE X 96.10951232910156 Y 56.32259750366211 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 1 PROCESSOR 0)
#S(PARTICLE X 13.404273986816407 Y 14.564407348632813 X0 0.0 Y0
0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 2
PROCESSOR 0)
#S(PARTICLE X 65.82643127441406 Y 24.56744956970215 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 3 PROCESSOR 0)
#S(PARTICLE X 99.88583374023438 Y 11.52757453918457 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 4 PROCESSOR 0)

>
```

Sorting particles

1) Our first task in doing anything with those particles is to sort them and insert into appropriate `xy-bins` for quick location, interpolation, etc.
2) Before we can sort the particles we must first decide what it is that we want to sort them into:

```
> (setq number-of-x-bins 32)
32
> (setq number-of-y-bins 32)
32
> (setq x-bin-width (/ x-width number-of-x-bins))
3.125
> (setq y-bin-width (/ y-width number-of-y-bins))
3.125
```

3) The first step is to sort the particles in one direction. This is accomplished by calling the `sort!!` function with an appropriate key

```
> (*setf particles (sort!! particles '<=!!
                               :key 'particle-x!!))
NIL
> (ppp particles :end 5)
#S(PARTICLE X 0.056815147399902344 Y 73.61520385742188 X0 0.0 Y0
0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 330
PROCESSOR 0)
#S(PARTICLE X 0.2538442611694336 Y 82.6822509765625 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 56 PROCESSOR
0)
#S(PARTICLE X 0.4670143127441406 Y 20.36408233642578 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 243 PROCESSOR
0)
```

```
#S(PARTICLE X 0.5941152572631836 Y 88.83160400390625 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 657 PROCESSOR
0)
#S(PARTICLE X 0.6241321563720703 Y 70.14872741699219 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 411 PROCESSOR
0)
>
```

4) Now we can assign the corresponding x-bin number to each particle

```
> (*setf (particle-x-bin!! particles)
         (truncate!! (/!! (particle-x!! particles)
                          (!! x-bin-width))))
NIL
> (ppp (particle-x-bin!! particles) :end 60)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1
> (ppp particles :start 40 :end 44)
#S(PARTICLE X 3.070080280303955 Y 3.162992000579834 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 629 PROCESSOR
0)
#S(PARTICLE X 3.1198859214782715 Y 21.63579559326172 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 205 PROCESSOR
0)
#S(PARTICLE X 3.1573891639709473 Y 19.115447998046875 X0 0.0 Y0
0.0 WORK1 0.0 WORK2 0.0 X-BIN 1 Y-BIN 0 BOX-MARK NIL ID 643
PROCESSOR 0)
#S(PARTICLE X 3.1862616539001465 Y 47.397422790527344 X0 0.0 Y0
0.0 WORK1 0.0 WORK2 0.0 X-BIN 1 Y-BIN 0 BOX-MARK NIL ID 186
PROCESSOR 0)
>
```

5) This stream of zeroes and ones can now be converted into a suitable segmentation mask as follows. First produce a new stream of zeroes and ones which is shifted by one place to the right. We can use our `segmented-news!!` or standard NEWS to achieve that

```
> (defpvar shifted-x-bin
           (segmented-news!!
           (particle-x-bin!! particles)
           (=!! (self-address!!) (!! 0))
           :border-pvar (!! -1)))
SHIFTED-X-BIN
> (ppp shifted-x-bin :end 5)
-1 0 0 0 0
```

```
> (ppp shifted-x-bin :start 40 :end 44)
0 0 0 1
>
```

6) The mask which marks the first particle in each `x-bin` can now be obtained by comparing `shifted-x-bin` with the contents of the `x-bin` slot:

```
> (defpvar x-bin-mark
           (/=!! (particle-x-bin!! particles)
                 shifted-x-bin))
X-BIN-MARK
> (ppp (particle-x-bin!! particles)
       :start 40 :end 44)
0 0 1 1
> (ppp x-bin-mark :start 40 :end 44)
NIL NIL T NIL
>
```

7) Now we can sort again particles in y-direction, but this time the sort will be restricted to the `x-bin` segments

```
> (*setf particles
      (sort!! particles '<=!! :key 'particle-y!!
             :segment-pvar x-bin-mark))
```

```
;;; Looking for file-set starlisp-bignums-f7600 directory definition in /usr/starlisp/starlisp-
f7600/dfs/starlisp-bignums-f7600.dfs
;;; Loading source file "/usr/starlisp/starlisp-f7600/dfs/starlisp-bignums-f7600.dfs"
;;; File-set directory for starlisp-bignums-f7600 defined.
;;; Looking for def-file-set starlisp-bignums-f7600 in /usr/starlisp/starlisp-f7600/lib/starlisp/bignums/def-
file-set.lisp
;;; Loading source file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/def-file-set.lisp"
;;; File-set starlisp-bignums-f7600 defined.
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/compiler-settings.sbin41x"

;;; Compiler settings: Safety 0, Warning Level :HIGH
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-utilities.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-globals.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-arithmetic.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-predicates.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-logic.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-comparisons.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-send.sbin41x"
;;; Loading binary file "/usr/starlisp/starlisp-f7600/lib/starlisp/bignums/bignum-scan.sbin41x"
NIL
>
```

Observe that this time *Lisp dynamically loaded additional libraries. This time we run `sort!!` with the segmentation mask for the first time. Like `scan!!` `sort!!` can also carry out operations restricted to the segments of a `pvar`.

8) After sorting our particles look as follows:

```
> (ppp particles :end 5)
#S(PARTICLE X 3.070080280303955 Y 3.162992000579834 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 629 PROCESSOR
0)
#S(PARTICLE X 1.9780397415161133 Y 4.0868401527404785 X0 0.0 Y0
0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 978
PROCESSOR 0)
#S(PARTICLE X 0.8625507354736328 Y 5.395853519439697 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 280 PROCESSOR
0)
#S(PARTICLE X 1.62581205368042 Y 5.515110492706299 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 839 PROCESSOR
0)
#S(PARTICLE X 2.370011806488037 Y 5.751502513885498 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 390 PROCESSOR
0)
>
```

They have been now sorted in the y direction, but they became again disorganised in the x direction. However, they still remain within their original x bins:

```
> (ppp particles :start 40 :end 44)
#S(PARTICLE X 0.5941152572631836 Y 88.83160400390625 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 657 PROCESSOR
0)
#S(PARTICLE X 0.8997678756713867 Y 95.01298522949219 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 0 BOX-MARK NIL ID 136 PROCESSOR
0)
#S(PARTICLE X 4.607486724853516 Y 1.3324618339538575 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 1 Y-BIN 0 BOX-MARK NIL ID 930 PROCESSOR
0)
#S(PARTICLE X 3.5369038581848145 Y 10.4466552734375 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 1 Y-BIN 0 BOX-MARK NIL ID 812 PROCESSOR
0)
>
```

9) Now we can assign the `y-bin` numbers to all particles knowing that they already have their `x-bin` numbers in place

```
> (*setf (particle-y-bin!! particles)
         (truncate!!
           (/!! (particle-y!! particles)
                (!! y-bin-width))))
NIL
> (ppp (particle-y-bin!! particles) :end 80)
1 1 1 1 1 2 3 6 6 7 8 11 11 12 12 13 15 18 20 20 21 21
22 22 22 23 23 23 23 24 24 25 26 26 26 26 26 26 28 28
28 30 0 3 3 4 4 4 5 6 6 6 6 8 9 10 11 11 15 16 17 18 18
18 19 20 22 22 24 24 24 26 27 31 31 1 2 2 2 3
```

```
>
```

10) We can now again produce `T` marks for the `y-bins` in the same way we have already done it for the `x-bins:`

```
> (defpvar y-bin-mark
    (*let ((shifted-y-bin-mark
             (segmented-news!!
               (particle-y-bin!! particles)
               x-bin-mark
               :border-pvar (!! -1))))
       (/=!! (particle-y-bin!! particles)
             shifted-y-bin-mark))
    boolean)
Y-BIN-MARK
> (ppp y-bin-mark :end 60)
T NIL NIL NIL NIL T T T NIL T T T NIL T NIL T T T T NIL
T NIL T NIL NIL T NIL NIL NIL T NIL T T NIL NIL NIL NIL
NIL T NIL NIL T T T NIL T NIL NIL T T NIL NIL NIL T T T
T NIL T T
>
```

Because we have created `y-bin-mark` on top of the `x-bin-mark` (which was used to roll-in -1 in its own locations `y-bin-mark` in fact marks the first particle in an xy-bin. Hence

```
> (*setf (particle-box-mark!! particles) y-bin-mark)
NIL
> (ppp particles :end 10)
#S(PARTICLE X 3.070080280303955 Y 3.162992000579834 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
T ID 629 PROCESSOR 0)
#S(PARTICLE X 1.9780397415161133 Y 4.0868401527404785 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-
MARK NIL ID 978 PROCESSOR 0)
#S(PARTICLE X 0.8625507354736328 Y 5.395853519439697 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-
MARK NIL ID 280 PROCESSOR 0)
#S(PARTICLE X 1.62581205368042 Y 5.515110492706299 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
NIL ID 839 PROCESSOR 0)
#S(PARTICLE X 2.370011806488037 Y 5.751502513885498 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
NIL ID 390 PROCESSOR 0)
#S(PARTICLE X 2.957797050476074 Y 7.583916187286377 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 2 BOX-MARK
T ID 338 PROCESSOR 0)
#S(PARTICLE X 2.962791919708252 Y 12.109363555908204 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 3 BOX-
MARK T ID 248 PROCESSOR 0)
#S(PARTICLE X 0.4670143127441406 Y 20.36408233642578 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 6 BOX-
MARK T ID 243 PROCESSOR 0)
#S(PARTICLE X 3.1198859214782715 Y 21.63579559326172 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 6 BOX-
MARK NIL ID 205 PROCESSOR 0)
#S(PARTICLE X 2.2101998329162598 Y 22.547542572021485 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 7 BOX-
MARK T ID 372 PROCESSOR 0)
>
```

11) We finish sorting particles by assigning the current processor number to each particle.

```
> (*setf (particle-processor!! particles)
         (self-address!!))
NIL
> (ppp particles :end 5)
#S(PARTICLE X 3.070080280303955 Y 3.162992000579834 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
T ID 629 PROCESSOR 0)
#S(PARTICLE X 1.9780397415161133 Y 4.0868401527404785 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-
MARK NIL ID 978 PROCESSOR 1)
#S(PARTICLE X 0.8625507354736328 Y 5.395853519439697 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-
MARK NIL ID 280 PROCESSOR 2)
#S(PARTICLE X 1.62581205368042 Y 5.515110492706299 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
NIL ID 839 PROCESSOR 3)
#S(PARTICLE X 2.370011806488037 Y 5.751502513885498 X0 0.0 Y0 0.0 WORK1 0.0 WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK
NIL ID 390 PROCESSOR 4)
>
```

Summary

1) Parallel structures on the CM are defined using `*defstruct` macro.
2) The creator and accessor functions for parallel structures have the suffix "`!!`". The macro `*defstruct` also generates creation and accessor functions on the front end.
3) Function `sort!!` can sort on relation and on a specified structure slot.
4) Function `sort!!` can be restricted to work within segments only.

Lesson 12
Using the Router

Mapping data from particles to boxes

1) Once we have the particles sorted it would be nice to be able to map them onto the 2D grid which corresponds to our xy-bins. In order to do that we have to introduce the new geometry of the processor lay-out. The current geometry is linear. What we want is the 2D geometry of a 2D grid.

```
> (def-vp-set box-geometry (list number-of-x-bins
                                 number-of-y-bins))
NIL
```

2) Now let us define two variables which will have this particular geometry

```
> (*proclaim '(type (pvar fixnum box-geometry)
            *first-particle-in-box*
            *number-of-particles-in-box*))
NIL
> (*defvar *first-particle-in-box*
          (!! -1) nil box-geometry)
*FIRST-PARTICLE-IN-BOX*
> (*defvar *number-of-particles-in-box*
          (!! 0) nil box-geometry)
*NUMBER-OF-PARTICLES-IN-BOX*
>
```

These two variables are 2D

```
> (ppp *first-particle-in-box*
       :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
>
```

3) Now we can transfer the processor number which resides in `(particle-processor!! particles)` for particles marked with `T` in `(particle-box-mark!! particles)` to `*first-particle-in-box*`. This transfer is difficult because we have to move data between two

data sets which are layed-out in different ways. This operation involves using the router.

```
> (*when (particle-box-mark!! particles)
    (*let ((address-pvar
             (cube-from-vp-grid-address!!
                box-geometry
                (particle-x-bin!! particles)
                (particle-y-bin!! particles))))
      (*pset
         :no-collisions
         (particle-processor!! particles)
         *first-particle-in-box* address-pvar)))
NIL
> (ppp *first-particle-in-box* :mode :grid
       :end '(5 5))

     DIMENSION 0 (X)  ----->

-1 42 -1 112 -1
0 -1 75 115 148
5 -1 76 117 150
6 43 79 -1 151
-1 45 81 120 152
> (ppp particles :end 5)
#S(PARTICLE X 3.070080280303955 Y 3.162992000579834 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 0 Y-BIN 1 BOX-MARK T ID 629 PROCESSOR 0)
#S(PARTICLE X 1.9780397415161133 Y 4.0868401527404785 X0 0.0 Y0 0.0 WORK1 0.0
WORK2 0.0 X-BIN 0 Y-BIN 1 BOX-MARK NIL ID 978 PROCESSOR 1)
#S(PARTICLE X 0.8625507354736328 Y 5.395853519439697 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 0 Y-BIN 1 BOX-MARK NIL ID 280 PROCESSOR 2)
#S(PARTICLE X 1.62581205368042 Y 5.515110492706299 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 0 Y-BIN 1 BOX-MARK NIL ID 839 PROCESSOR 3)
#S(PARTICLE X 2.370011806488037 Y 5.751502513885498 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 0 Y-BIN 1 BOX-MARK NIL ID 390 PROCESSOR 4)
>
```

4) In order to put data in `*number-of-particles-in-box*` we must first calculate how many particles there are in every box. This is easily achieved by summing up "`1`" within segments in the backward mode (so that the result sits in the location marked with "`T`") for all particles:

```
> (defpvar segmented-sums (!! 0) fixnum)
SEGMENTED-SUMS
> (*setf segmented-sums
         (scan!! (!! 1) '+!! :segment-pvar
                 (particle-box-mark!! particles)
```

```
                    :direction :backward
                    :segment-mode :segment
                    :include-self t))
NIL
> (ppp segmented-sums :end 60)
5 4 3 2 1 1 1 2 1 1 1 2 1 2 1 1 1 1 2 1 2 1 3 2 1 4 3 2
1 2 1 1 6 5 4 3 2 1 3 2 1 1 1 2 1 3 2 1 1 4 3 2 1 1 1 1
2 1 1 1
> (ppp (particle-box-mark!! particles) :end 60)
T NIL NIL NIL NIL T T T NIL T T T NIL T NIL T T T T NIL
T NIL T NIL NIL T NIL NIL NIL T NIL T T NIL NIL NIL NIL
NIL T NIL NIL T T T NIL T NIL NIL T T NIL NIL NIL T T T
T NIL T T
>
```

5) Now we can transfer the lengths from `segmented-sums` to `*number-of-particles-in-box*` the same way we've done it for `*first-particle-in-box*`

```
> (*when (particle-box-mark!! particles)
    (*let ((address-pvar (cube-from-vp-grid-address!!
                  box-geometry
                  (particle-x-bin!! particles)
                  (particle-y-bin!! particles))))
      (*pset :no-collisions segmented-sums *number-of-
particles-in-box*
          address-pvar)))
NIL
>
```

Printing the contents of a box

1) At this stage we can easily print particles which reside in any box. Below is a function which finds about the first particle in a given box, then finds about the number of particles in that box and finally prints all particles in that box:

```
(defun print-particles (particle-pvar x-bin y-bin)
  (let ((length
          (pref *number-of-particles-in-box*
                (cube-from-vp-grid-address
                  box-geometry x-bin y-bin)))
        (first-particle
          (pref *first-particle-in-box*
                (cube-from-vp-grid-address
```

```
                box-geometry x-bin y-bin))))
    (dotimes (i length)
      (format t "~&~a"
              (pref particle-pvar
                    (+ first-particle i))))))
```

```
> (ppp *number-of-particles-in-box* :mode :grid :end '(10 10))

    DIMENSION 0 (X)  ----->

0 3 0 1 1 1 1 1 1 0
1 0 1 0 1 3 2 1 2 0
2 1 1 1 2 1 3 0 1 0
1 0 3 0 0 0 2 0 1 1
2 0 0 0 0 1 1 1 3 1
1 2 0 0 1 0 0 0 0 0
2 0 1 0 1 3 0 3 2 0
0 1 3 1 0 2 0 0 2 0
2 2 3 1 0 2 0 0 0 1
1 0 1 1 0 1 0 0 1 1
> (print-particles particles 5 1)
#S(PARTICLE X 15.642749786376954 Y 3.9786696434020996 X0 0.0 Y0 0.0 WORK1 0.0
WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK T ID 31 PROCESSOR 171)
#S(PARTICLE X 16.999805450439453 Y 4.175364971160889 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 473 PROCESSOR 172)
#S(PARTICLE X 15.692747116088868 Y 5.684483051300049 X0 0.0 Y0 0.0 WORK1 0.0 WORK2
0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 676 PROCESSOR 173)
NIL
> (print-particles particles 6 3)
#S(PARTICLE X 18.824827194213867 Y 10.890901565551758 X0 0.0 Y0 0.0 WORK1 0.0
WORK2 0.0 X-BIN 6 Y-BIN 3 BOX-MARK T ID 702 PROCESSOR 212)
#S(PARTICLE X 21.394872665405274 Y 10.925054550170899 X0 0.0 Y0 0.0 WORK1 0.0
WORK2 0.0 X-BIN 6 Y-BIN 3 BOX-MARK NIL ID 1015 PROCESSOR 213)
NIL
```

Calculating density using the Monaghan-Lattanzio weighting function

1) Below is a definition of a parallel function called `joes-kernel!!` which computes the value of the Monaghan-Lattanzio weighting function for each particle:

```
(defun joes-kernel!! (r h)
  "Compute Monaghan-Lattanzio kernel. r is the radius,
and h is the smoothing length."
  (*let ((r-by-h (/!! r h))
         (one-by-pi-h-cube (/!! (!! 1.0)
                                (!! pi) h h h)))
    (cond!!((<!! r h)
            (*let ((second-term
                    (-!! (*!! (!! 1.5)
                              r-by-h r-by-h)))
                   (third-term
```

```
                    (*!! (!! 0.75) r-by-h
                         r-by-h r-by-h)))
               (*!! one-by-pi-h-cube
                    (+!! (!! 1.0)
                         second-term third-term))))
          ((<!! r (*!! (!! 2.0) h))
           (*let ((two-less-r-by-h
                    (-!! (!! 2.0) r-by-h)))
             (*!! one-by-pi-h-cube
                  (!! 0.25) two-less-r-by-h
                  two-less-r-by-h two-less-r-by-h)))
          (t (!! 0.0)))))
```

Here is how this function works

```
> (ppp (joes-kernel!! (!! 0.0) (!! 1.0)) :end 3)
0.31830987334251404 0.31830987334251404
0.31830987334251404
> (ppp (joes-kernel!! (!! 1.0) (!! 1.0)) :end 3)
0.07957746833562851 0.07957746833562851
0.07957746833562851
> (ppp (joes-kernel!! (!! 2.0) (!! 1.0)) :end 3)
0.0 0.0 0.0
```

2) We can use this function in order to compute contributions the particles will make to a point located exactly in the middle of each box. Let us first define that point:

```
(*proclaim '(type (pvar single-float box-geometry)
                  centres-of-boxes-x))
NIL
> (*defvar centres-of-boxes-x 0.0)
CENTRES-OF-BOXES-X
> (*proclaim '(type (pvar single-float box-geometry)
                  centres-of-boxes-y))
NIL
> (*defvar centres-of-boxes-y 0.0)
CENTRES-OF-BOXES-Y
```

At this stage we just have the variables in place but nothing interesting in them yet. Now, hold on to your seats...

```
> (*with-vp-set box-geometry
  (*set centres-of-boxes-x
     (+!! (*!!
```

```
            (grid-from-vp-cube-address!!
               box-geometry (self-address!!) (!! 0))
            (!! x-bin-width))
          (/!! (!! x-bin-width) (!! 2.0))))
  (*set centres-of-boxes-y
     (+!! (*!!
            (grid-from-vp-cube-address!!
               box-geometry (self-address!!) (!! 1))
            (!! y-bin-width))
          (/!! (!! y-bin-width) (!! 2.0)))))
NIL
> (ppp centres-of-boxes-x :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1.5625 4.6875 7.8125 10.9375 14.0625
1.5625 4.6875 7.8125 10.9375 14.0625
1.5625 4.6875 7.8125 10.9375 14.0625
1.5625 4.6875 7.8125 10.9375 14.0625
1.5625 4.6875 7.8125 10.9375 14.0625
> (ppp centres-of-boxes-y :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

1.5625 1.5625 1.5625 1.5625 1.5625
4.6875 4.6875 4.6875 4.6875 4.6875
7.8125 7.8125 7.8125 7.8125 7.8125
10.9375 10.9375 10.9375 10.9375 10.9375
14.0625 14.0625 14.0625 14.0625 14.0625
```

3) In order to be able to carry out further computations we must transfer these to `(particle-x0!! particles)` and `(particle-y0!! particles)`. Note that we could easily compute both for each particle by operating on the `particles` pvar itself. But in this case we additionally learn how to pass data from grid to particles (we've done it so far only in the opposite direction). Let us first transfer `centres-of-boxes-x` to `x0`:

```
> (*with-vp-set box-geometry
  (*when (>!! *number-of-particles-in-box* (!! 0))
    (*pset :no-collisions centres-of-boxes-x
        (alias!! (particle-x0!! particles))
        *first-particle-in-box*
        :vp-set *default-vp-set*)
    (*pset :no-collisions centres-of-boxes-y
```

```
        (alias!! (particle-y0!! particles))
        *first-particle-in-box*
        :vp-set *default-vp-set*)))
NIL
> (ppp *number-of-particles-in-box* :mode :grid
       :end '(10 10))

     DIMENSION 0 (X)  ----->

0 3 0 1 1 1 1 1 1 0
1 0 1 0 1 3 2 1 2 0
2 1 1 1 2 1 3 0 1 0
1 0 3 0 0 0 2 0 1 1
2 0 0 0 0 1 1 1 3 1
1 2 0 0 1 0 0 0 0 0
2 0 1 0 1 3 0 3 2 0
0 1 3 1 0 2 0 0 2 0
2 2 3 1 0 2 0 0 0 1
1 0 1 1 0 1 0 0 1 1
> (print-particles particles 5 1)
#S(PARTICLE X 15.642749786376954 Y 3.9786696434020996 X0 17.1875
Y0 4.6875 WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK T ID 31
PROCESSOR 171)
#S(PARTICLE X 16.999805450439453 Y 4.175364971160889 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 473 PROCESSOR
172)
#S(PARTICLE X 15.692747116088868 Y 5.684483051300049 X0 0.0 Y0 0.0
WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 676 PROCESSOR
173)
NIL
```

4) We see that we have transferred the data from `centres-of-boxes-x` and from `centres-of-boxes-y` into the locations of the box marks in `particles` pvar. But the second, third, and so on particles in each box were not affected. We now must spread the data from the box mark locations over the segments.

```
> (*setf (particle-x0!! particles)
      (scan!! (particle-x0!! particles) 'copy!!
           :segment-pvar (particle-box-mark!!
particles)))
NIL
> (*setf (particle-y0!! particles)
      (scan!! (particle-y0!! particles) 'copy!!
           :segment-pvar (particle-box-mark!!
particles)))
NIL
```

Let us see the result of these two operations:

```
> (print-particles particles 5 1)
#S(PARTICLE X 15.642749786376954 Y 3.9786696434020996 X0 17.1875
Y0 4.6875 WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK T ID 31
PROCESSOR 171)
#S(PARTICLE X 16.999805450439453 Y 4.175364971160889 X0 17.1875 Y0
4.6875 WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 473
PROCESSOR 172)
#S(PARTICLE X 15.692747116088868 Y 5.684483051300049 X0 17.1875 Y0
4.6875 WORK1 0.0 WORK2 0.0 X-BIN 5 Y-BIN 1 BOX-MARK NIL ID 676
PROCESSOR 173)
NIL
> (print-particles particles 6 2)
#S(PARTICLE X 21.03689956665039 Y 6.321084499359131 X0 20.3125 Y0
7.8125 WORK1 0.0 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK T ID 912
PROCESSOR 209)
#S(PARTICLE X 21.653057098388672 Y 8.090567588806153 X0 20.3125 Y0
7.8125 WORK1 0.0 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK NIL ID 981
PROCESSOR 210)
#S(PARTICLE X 18.765186309814453 Y 8.745384216308594 X0 20.3125 Y0
7.8125 WORK1 0.0 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK NIL ID 551
PROCESSOR 211)
NIL
```

5) It is now quite trivial to compute the value of Joe's kernel for each particle in the system. Let us define the value of the smoothing length first. It can be made different for every particle within this formalism, but for simplicity we assume that

```
> (defpvar smoothing-length (!! x-bin-width) single-
float)
SMOOTHING-LENGTH
```

First we calculate the distance between any particle and its corresponding centre of the box

```
> (*setf (particle-work1!! particles)
       (*let ((dx (-!! (particle-x!! particles)
                       (particle-x0!! particles)))
              (dy (-!! (particle-y!! particles)
                       (particle-y0!!
                         particles))))
         (sqrt!! (+!! (*!! dx dx) (*!! dy dy)))))
NIL
> (print-particles particles 6 2)
#S(PARTICLE X 21.03689956665039 Y 6.321084499359131 X0 20.3125 Y0
7.8125 WORK1 1.6580334901809693 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK
T ID 912 PROCESSOR 209)
```

```
#S(PARTICLE X 21.653057098388672 Y 8.090567588806153 X0 20.3125 Y0
7.8125 WORK1 1.3690927028656006 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK
NIL ID 981 PROCESSOR 210)
#S(PARTICLE X 18.765186309814453 Y 8.745384216308594 X0 20.3125 Y0
7.8125 WORK1 1.8067796230316162 WORK2 0.0 X-BIN 6 Y-BIN 2 BOX-MARK
NIL ID 551 PROCESSOR 211)
NIL
>
```

6) The next step is to compute the value of Joe's kernel for every particle

```
> (*setf (particle-work2!! particles)
      (joes-kernel!! (particle-work1!! particles)
                     smoothing-length))
NIL
> (print-particles particles 6 2)
#S(PARTICLE X 21.03689956665039 Y 6.321084499359131 X0 20.3125 Y0
7.8125 WORK1 1.6580334901809693 WORK2 0.0071944668889045715 X-BIN
6 Y-BIN 2 BOX-MARK T ID 912 PROCESSOR 209)
#S(PARTICLE X 21.653057098388672 Y 8.090567588806153 X0 20.3125 Y0
7.8125 WORK1 1.3690927028656006 WORK2 0.008085190318524838 X-BIN 6
Y-BIN 2 BOX-MARK NIL ID 981 PROCESSOR 210)
#S(PARTICLE X 18.765186309814453 Y 8.745384216308594 X0 20.3125 Y0
7.8125 WORK1 1.8067796230316162 WORK2 0.006712291855365038 X-BIN 6
Y-BIN 2 BOX-MARK NIL ID 551 PROCESSOR 211)
NIL
>
```

7) In order to calculate, for example, the density field all we need to do is to sum up the contributions Monaghan-Lattanzio kernels make to the grid points located at the centre of each box from all particles within that box and beyond. We again invoke the segmented scan operation in order to do that and then transfer the data onto the density field variable which is of the `box-geometry`

```
> (*proclaim
    '(type (pvar single-float box-geometry)
           density))
NIL
> (*defvar density 0.0)
DENSITY
> (*let ((segment-sums
           (scan!!
             (particle-work2!! particles) '+!
             :segment-pvar
             (particle-box-mark!! particles)
             :direction :backward
```

```
                :segment-mode :segment
                :include-self t)))
      (*when (particle-box-mark!! particles)
        (*let ((address-pvar
                  (cube-from-vp-grid-address!!
                       box-geometry
                       (particle-x-bin!! particles)
                       (particle-y-bin!! particles))))
         (*pset :no-collisions segment-sums
                density address-pvar))))
NIL
> (ppp density :mode :grid :end '(5 5))

     DIMENSION 0 (X)  ----->

0.0 0.024203049018979073 0.0 0.007555817253887653
0.008214011788368225
0.006345123052597046 0.0 0.008530116640031338 0.0
0.005641032941639423
0.01436174102127552 0.010306714102625847 0.009300578385591507
0.009039076045155526 0.015887316316366196
0.008602922782301903 0.0 0.028247637674212456 0.0 0.0
0.01587417908012867 0.0 0.0 0.0 0.0
```

Summary

1) A new non-default geometry is defined with the function `def-vp-set`.
2) Variables of a non-standard geometry have to be specially declared using `*proclaim` before they are created with `*defvar` (note, not `defpvar`).
3) Data can be transferred from one `vp-set` to another one using the function `*pset`. This function operates the router.
4) The function `*with-vp-set` will alter the default geometry within its body to a specified one.