Department of Computer and information Science
Norwegian University of Science and Technology

http://www.idi.ntnu.no/

# A Crash Course in LISP MNFIT272 – 2002

Anders Kofod-Petersen

anderpe@idi.ntnu.no

## Introduction

What are the topics for this crash course?

## Introduction

What are the topics for this crash course?

- Overview

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data
  - Functions

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data
  - Functions
  - Recursion

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data
  - Functions
  - Recursion
  - Assignment

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data
  - Functions
  - Recursion
  - Assignment
  - Functional programming

## Introduction

What are the topics for this crash course?

- Overview
- Symbol manipulation
- LISP
  - Data
  - Functions
  - Recursion
  - Assignment
  - Functional programming
  - Iteration

## Overview

## Overview

- LISP takes its name from <u>L</u>i<u>s</u>t <u>P</u>roccessing, some argue that it comes from <u>L</u>ots of <u>I</u>rritating <u>S</u>illy <u>P</u>arentheses

## Overview

- LISP takes its name from List Proccessing, some argue that it comes from Lots of Irritating Silly Parentheses
- LISP has the easiest syntax of all languages

## Overview

- LISP takes its name from List Proccessing, some argue that it comes from Lots of Irritating Silly Parentheses
- LISP has the easiest syntax of all languages
- LISP is oriented towards the manipulation of symbols

## Overview

- LISP takes its name from List Proccessing, some argue that it comes from Lots of Irritating Silly Parentheses
- LISP has the easiest syntax of all languages
- LISP is oriented towards the manipulation of symbols
- LISP is very flexible in that users can change the syntax of the language if they don't like it

## New tools

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

```
(defun sum (n)
       (let ((s 0))
            (dotimes (i n s)
       (incf s i))))
```

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

```
(defun sum (n)
      (let ((s 0))
           (dotimes (i n s)
      (incf s i))))
```

```
int sum(int n){
     int i,s=0;
     for(1=0;i<n;i++){
          s+=i;}
return(s);
}
```

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

```
(defun sum (n)
      (let ((s 0))
           (dotimes (i n s)
      (incf s i))))
```

```
int sum(int n){
     int i,s=0;
     for(1=0;i<n;i++){
          s+=i;}
return(s);
}
```

- To write a function that takes a number $n$ and returns a function that adds $n$ n to its arguments:

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

```
(defun sum (n)
      (let ((s 0))
           (dotimes (i n s)
      (incf s i))))
```

```
int sum(int n){
     int i,s=0;
     for(1=0;i<n;i++){
          s+=i;}
return(s);
}
```

- To write a function that takes a number $n$ and returns a function that adds $n$ n to its arguments:

```
(defun addn (n)
     #'(lambda (x)
          (+ x n)))
```

## New tools

- You can do things in LISP that can't be done in "ordinary" programming languages
- To write a function that returns the sum:

```
(defun sum (n)
      (let ((s 0))
           (dotimes (i n s)
      (incf s i))))
```

```
int sum(int n){
     int i,s=0;
     for(1=0;i<n;i++){
          s+=i;}
return(s);
}
```

- To write a function that takes a number $n$ and returns a function that adds $n$ n to its arguments:

```
(defun addn (n)
     #'(lambda (x)
          (+ x n)))
```

Sorry, no can do!

## Symbol Manipulation

## Symbol Manipulation

- Everything in a computer is a string of bits

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`
- Groups of atoms can form sentence like objects called *lists*

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`
- Groups of atoms can form sentence like objects called *lists*
  - Examples are: `(1 2 3)` `(cow horse (hog pig))`

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`
- Groups of atoms can form sentence like objects called *lists*
  - Examples are: `(1 2 3) (cow horse (hog pig))`
- Atoms and lists are collectively called *symbolic expressions*

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`
- Groups of atoms can form sentence like objects called *lists*
  - Examples are: `(1 2 3) (cow horse (hog pig))`
- Atoms and lists are collectively called *symbolic expressions*
- Programs and data are represented the same way

## Symbol Manipulation

- Everything in a computer is a string of bits
- Bits can be interpreted as a code for word like objects
- The fundamental things formed from bits in LISP are *atoms*
  - Examples are: `3.1416`, `This-is-an-atom`, `nil`
- Groups of atoms can form sentence like objects called *lists*
  - Examples are: `(1 2 3) (cow horse (hog pig))`
- Atoms and lists are collectively called *symbolic expressions*
- Programs and data are represented the same way
  - is `(+ 10 15) a program or data?`

## LISP

### Data

## LISP

### Data

- LISP offers all the data types that other languages do

# LISP

## Data

- LISP offers all the data types that other languages do
- “New” data types are:

## LISP

### Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`

## LISP

### Data

- LISP offers all the data types that other languages do
- “New” data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`

# LISP

## Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:

## LISP

### Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`

# LISP

## Data

- LISP offers all the data types that other languages do
- “New” data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`
- Lists are build by using `cons`:

# LISP

## Data

- LISP offers all the data types that other languages do
- “New” data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`
- Lists are build by using `cons`:
  - `(cons 'a '(b c d)) -- (a b c d)`

## LISP

### Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`
- Lists are build by using `cons`:
  - `(cons 'a '(b c d)) -- (a b c d)`
- Lists are manipulated by using `car` and `cdr`:

## LISP

### Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`
- Lists are build by using `cons`:
  - `(cons 'a '(b c d)) -- (a b c d)`
- Lists are manipulated by using `car` and `cdr`:
  - `(car '(a b c)) -- a`

## LISP

### Data

- LISP offers all the data types that other languages do
- "New" data types are:
  - Symbols – `'LISP`
  - Lists – `'(LISP has 2 new types)`
- You can build lists by `list`:
  - `(list 'LISP 'has (+ 1 1) 'new ``types'')`
- Lists are build by using `cons`:
  - `(cons 'a '(b c d)) -- (a b c d)`
- Lists are manipulated by using `car` and `cdr`:
  - `(car '(a b c)) -- a`
  - `(cdr '(a b c)) -- (c d)`

## Functions

## Functions

- You can define new functions with `defun`

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function
  - 
    ```
    (defun square (x)
           (* x x))
    ```

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function

  -
    ```
    (defun square (x)
           (* x x))
    ```
  - Notice that LISP uses *prefix* and not *infix* notation

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function

  -
    ```
    (defun square (x)
            (* x x))
    ```
  - Notice that LISP uses *prefix* and not *infix* notation
  - LISP makes no distinction between a *program*, a *procedure*, or a *function* – Functions do for everything

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function

  -
    ```
    (defun square (x)
           (* x x))
    ```
  - Notice that LISP uses *prefix* and not *infix* notation
  - LISP makes no distinction between a *program*, a *procedure*, or a *function* – Functions do for everything
- You can also read and write input and output

## Functions

- You can define new functions with `defun`
  - `Defun` takes three or more arguments:
    1. A name
    2. A list of parameters
    3. One or more expressions that makes the body of the function

  -
    ```
    (defun square (x)
           (* x x))
    ```
  - Notice that LISP uses *prefix* and not *infix* notation
  - LISP makes no distinction between a *program*, a *procedure*, or a *function* – Functions do for everything
- You can also read and write input and output

  -
    ```
    (defun ask (string)
          (format t `` A'' string)
          (read))
    ```

## Recursion

## Recursion

- Recursion is a big thing in LISP

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

```
(defun len (lst)
       (if (null lst) 0
            (+ (len (cdr lst)) 1)))
```

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

```
(defun len (lst)
       (if (null lst) 0
            (+ (len (cdr lst)) 1)))
 > (len '(a b c d))
4
```

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

```
(defun len (lst)
       (if (null lst) 0
            (+ (len (cdr lst)) 1)))
 > (len '(a b c d))
4
```

  - Recursive functions can be used for more complex problems:

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

```
(defun len (lst)
       (if (null lst) 0
            (+ (len (cdr lst)) 1)))
 > (len '(a b c d))
4
```

  - Recursive functions can be used for more complex problems: `(defun fibonacci (x)`

```
     (if (<= x 2) 1
          (+ (fibonacci (- x 2))(fibonacci (1- x)))))
```

## Recursion

- Recursion is a big thing in LISP
  - When writing recursive functions two issues are paramount:
    1. Handle the trivial case first
    2. Then handle the recursive case
  - Defining a function that calculates the length of a list:

```
(defun len (lst)
       (if (null lst) 0
            (+ (len (cdr lst)) 1)))
 > (len '(a b c d))
4
```

  - Recursive functions can be used for more complex problems: `(defun fibonacci (x)`

```
     (if (<= x 2) 1
          (+ (fibonacci (- x 2))(fibonacci (1- x))))) > (fibonacci 10)
55
```

## Variables

## Variables

- There are several ways of defining variables in LISP:

## Variables

- There are several ways of defining variables in LISP:
    1. Using the `setf` construction

## Variables

- There are several ways of defining variables in LISP:
  1. Using the `setf` construction
  2. Using the `let` construction

## Variables

- There are several ways of defining variables in LISP:
  1. Using the `setf` construction
  2. Using the `let` construction

```
(defun showvar (x y)
      (setf var x)
      (let ((var y))
            (print var))
      (print var)
      t)
```

## Variables

- There are several ways of defining variables in LISP:
  1. Using the `setf` construction
  2. Using the `let` construction

```
(defun showvar (x y)
      (setf var x)
      (let ((var y))
           (print var))
      (print var)
      t)
 > (showvar 10 5)
5
10
t
```

## Variables

- There are several ways of defining variables in LISP:
  1. Using the `setf` construction
  2. Using the `let` construction

```
(defun showvar (x y)
     (setf var x)
     (let ((var y))
          (print var))
     (print var)
     t)
 > (showvar 10 5)
5
10
t
```

- For global parameters use `defparameter` or `defconstant`

## Variables

- There are several ways of defining variables in LISP:
  1. Using the `setf` construction
  2. Using the `let` construction

```
(defun showvar (x y)
      (setf var x)
      (let ((var y))
            (print var))
      (print var)
      t)
 > (showvar 10 5)
5
10
t
```

- For global parameters use `defparameter` or `defconstant`
  - `(defparameter *global* 100)`

## Functional programming

## Functional programming

- Functional programming means writing functions that *returns* values

## Functional programming

- Functional programming means writing functions that *returns* values
- `> (setf lst '(H e l l o))`

## Functional programming

- Functional programming means writing functions that *returns* values
- `> (setf lst '(H e l l o))`
- `> (remove 'l lst)`

  `(H e o)`

## Functional programming

- Functional programming means writing functions that *returns* values

-
  ```
  > (setf lst '(H e l l o))
  ```

-
  ```
  > (remove 'l lst)
  (H e o)
  ```

-
  ```
  > lst
  (H e l l o)
  ```

## Functional programming

- Functional programming means writing functions that *returns* values
- `> (setf lst '(H e l l o))`

-
  ```
  > (remove 'l lst)
  (H e o)
  ```

-
  ```
  > lst
  (H e l l o)
  ```
- The most important advantage with this non-destructive way of writing, is the possibility if testing each function as you write them, and guarantee that they are correct

## Functional programming

- Functional programming means writing functions that *returns* values

-
  ```
  > (setf lst '(H e l l o))
  ```

-
  ```
  > (remove 'l lst)
  (H e o)
  ```

-
  ```
  > lst
  (H e l l o)
  ```
- The most important advantage with this non-destructive way of writing, is the possibility if testing each function as you write them, and guarantee that they are correct
- If you want to write destructive code you can do:

## Functional programming

- Functional programming means writing functions that *returns* values
- `> (setf lst '(H e l l o))`

-
  ```
  > (remove 'l lst)
  (H e o)
  ```

-
  ```
  > lst
  (H e l l o)
  ```
- The most important advantage with this non-destructive way of writing, is the possibility if testing each function as you write them, and guarantee that they are correct
- If you want to write destructive code you can do:
  ```
  > (setf lst (remove 'l lst))
  ```

## Iteration

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))
```

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))


> (lenit ’(a b c d))
4
```

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))


> (lenit '(a b c d))
4
```

- `dolist` is not the only interactive function, others are: `do`, `dotimes`, and `loop`

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))


> (lenit '(a b c d))
4
```

- `dolist` is not the only interactive function, others are: `do`, `dotimes`, and `loop`
- For handling batch like coding, LISP offers constructions like `progn`

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))


> (lenit '(a b c d))
4
```

- `dolist` is not the only interactive function, others are: `do`, `dotimes`, and `loop`
- For handling batch like coding, LISP offers constructions like `progn`

```
(defun write2 (lst)
     (progn (print lst)
          (print (reverse lst))))
```

## Iteration

- It can sometimes be more natural repeat a procedure in an *interactive* way, as compared to a recursive way
- Redefining the length function:

```
(defun lenit (lst)
     (let ((len 0))
          (dolist (obj lst) (setf len (+ len 1))) len))


> (lenit '(a b c d))
4
```

- `dolist` is not the only interactive function, others are: `do`, `dotimes`, and `loop`
- For handling batch like coding, LISP offers constructions like `progn`

```
(defun write2 (lst)
     (progn (print lst)
          (print (reverse lst))))
```

```
> (write2 '(a b c d e))
(a b c b a)
(e d c b a)
```

## LISP at IDI

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

```
(setq inferior-lisp-program "/local/acl/mlisp")
(load "/usr/local/acl/eli/fi-site-init.el")
```

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

  ```
  (setq inferior-lisp-program "/local/acl/mlisp")
  (load "/usr/local/acl/eli/fi-site-init.el")
  ```

- Notice that emacs is controlled with LISP!

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

```
(setq inferior-lisp-program "/local/acl/mlisp")
(load "/usr/local/acl/eli/fi-site-init.el")
```

- Notice that emacs is controlled with LISP!

1. Open an emacs

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

```
(setq inferior-lisp-program "/local/acl/mlisp")
(load "/usr/local/acl/eli/fi-site-init.el")
```

- Notice that emacs is controlled with LISP!

1. Open an emacs
2. Type `<esc>-x`

   `fi:commmon-lisp`

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

```
(setq inferior-lisp-program "/local/acl/mlisp")
(load "/usr/local/acl/eli/fi-site-init.el")
```

- Notice that emacs is controlled with LISP!

1. Open an emacs
2. `Type <esc>-x`
   `fi:commmon-lisp`
3. Make a new frame

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

  ```
  (setq inferior-lisp-program "/local/acl/mlisp")
  (load "/usr/local/acl/eli/fi-site-init.el")
  ```
- Notice that emacs is controlled with LISP!

1. Open an emacs
2. Type `<esc>-x`
   `fi:commmon-lisp`
3. Make a new frame
4. type `<esc>-x`
   `lisp-mode`

## LISP at IDI

- The LISP used at IDI is Allegro Common LISP; a de facto standard
- To get LISP to work in emacs, put the following into your .emacs file:

  ```
  (setq inferior-lisp-program "/local/acl/mlisp")
  (load "/usr/local/acl/eli/fi-site-init.el")
  ```

- Notice that emacs is controlled with LISP!

1. Open an emacs
2. `Type <esc>-x`

   `fi:commmon-lisp`
3. Make a new frame
4. `type <esc>-x`

   `lisp-mode`
5. You can now write your lisp code in one window, and compile it with `<ctrl>-x`

## Litterature

- LISP 3rd edition
  Patrick Henry Winston
  Berthold Klaus Paul Horn
  ISBN: 0-201-08379-1
- ANSI Common Lisp
  Paul Graham
  ISBN: 0-13-370875-6
- Common Lisp the Language, 2nd edition
  Guy L. Steel
  ISBN: 1-55558-041-6
  http://www.math.uio.no/cltl/cltl2.html