Introduction to Cognitive Robotics

Module 8: An Introduction to Functional Programming with Lisp
Lecture 2: Common Lisp - functions, I/O, recursion, iteration, lambda and mapping functions, CLOS, inference

www.cognitiverobotics.net

REPL

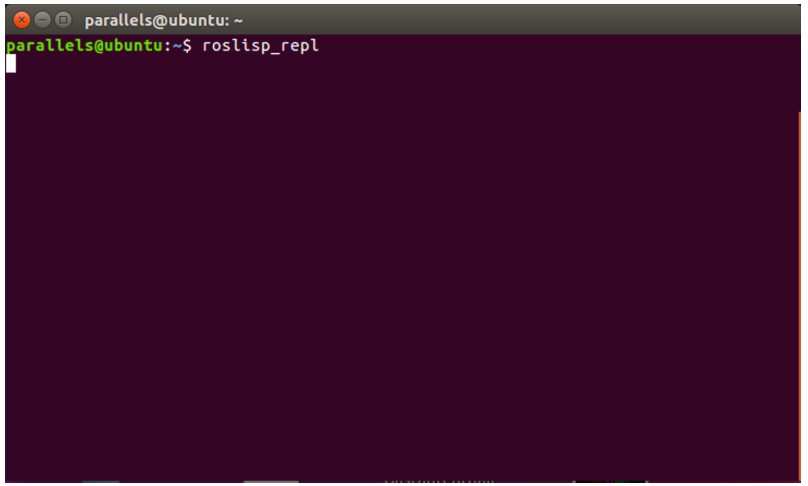
(Read-Eval-Print Loop)

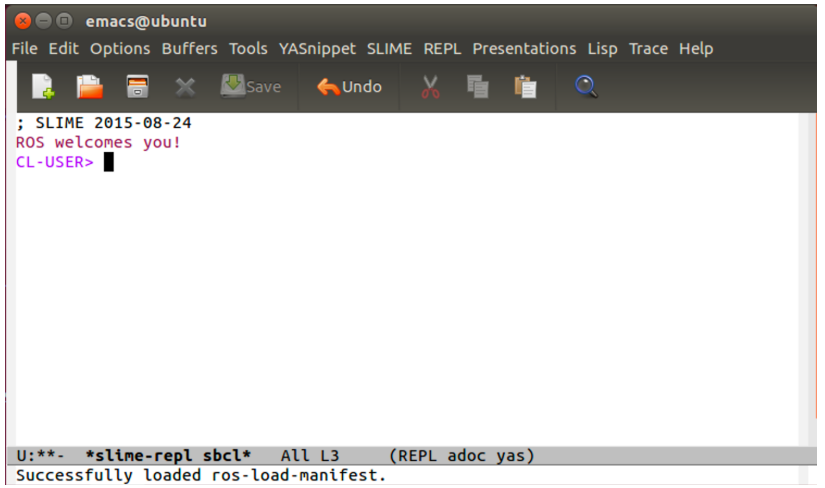
As you go through this set of slides, you might like to try out the examples.

Use `roslisp_repl` to launch the Lisp compiler's interactive front-end: REPL

```
$ roslisp_repl
```

Command entered in terminal



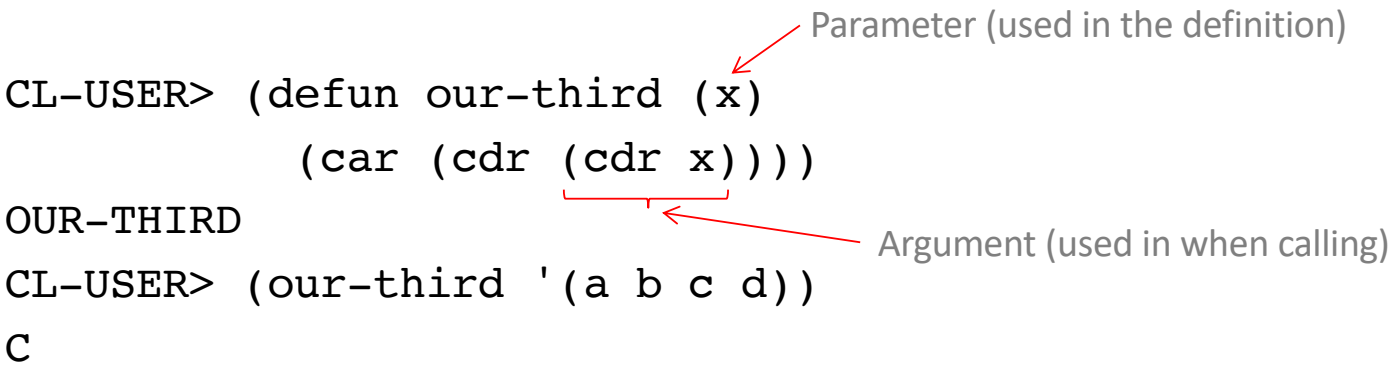


Functions

- You can define new functions with `defun`

- Functions usually takes three or more arguments:

  1. a name
  2. a list of parameters
  3. one or more expressions that will make up the body of the function



Parameter lists

Optional parameters

- Can be omitted and a default value used instead
- Place `&optional` after the the last required parameter in the parameter list

```
CL-USER> (defun philosoph (thing &optional property)
           (list thing 'is property)
CL-USER> (philosoph 'death)
(DEATH IS NIL)
```

No value provided
so it defaults to NIL

Parameter lists

Optional parameters

- Can be omitted and a default value used instead
- Place `&optional` after the the last required parameter in the parameter list

```
CL-USER> (defun philosoph (thing &optional (property 'fun))
           (list thing 'is property)
CL-USER> (philosoph 'death)
(DEATH IS FUN)
```

Default value provided after the parameter, enclosing both in brackets. It doesn't have to be a constant, it can be an expression

Parameter lists

Keyword parameters

(NB … these are used a lot in the CRAM plan language)

- Also an optional parameter but more flexible
- Place `&key` after the the last required parameter in the parameter list
- All parameters after it are optional
- When the function is called, these parameters will be identified not by their position (as is usual) but by symbolic tags that precede them

Parameter lists

Keyword parameters

(NB … these are used a lot in CRAM)

```
CL-USER> (defun keylist (a &key x y z)
           (list a x y z))
KEYLIST
CL-USER> (keylist 1 :y 2)
(1 NIL 2 NIL)
CL-USER> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

The parameter names are used to define the associated keywords by prepending a colon, e.g. `:x`, `:y`, and `:z`

Note the arbitrary ordering of the keyword arguments

Parameter lists

Variable number of arguments

- Place `&rest` before the last variable in the parameter list
- When the function is called, this variable will be set to be a list of all the remaining arguments

```
CL-USER>(defun our-funcall (fn &rest args)
          (apply fn args))
```

List of arguments

We cover the `apply` function later

The parameter is a function, i.e. we pass a function as an argument to our-funcall ... we cover this later too

Return Values

- All the functions we have seen so far return just one value
- Functions can return multiple values using the `values` function
  - It returns exactly the values you give it as arguments

```
CL-USER>(values 'a nil (+ 2 4))
A
NIL
6
```

  - For a function to return multiple values, make the `values` expression the last thing to be evaluated in the body of the function

Return Values

Functions can return multiple values using the `values` function

- The returned multiple values are accessed with the `multiple-value-bind` function

```
CL-USER> (multiple-value-bind (x y z) (values 1 2 3)
           (list x y z))
(1 2 3)
```

Instead of this expression, you would have the call to the function that returns multiple values

Return Values

Functions can return multiple values using the `values` function

- If there are more variables than values, the leftover ones will be `nil`.

- If there are more values than variables, the extra values will be discarded.

```
CL-USER> (multiple-value-bind (x y z) (values 1 2)
           (list x y z))
(1 2 NIL)
CL-USER> (multiple-value-bind (s m h) (get-decoded-time)
             (format nil "~A:~A:~A" h m s))
"4:32:13"
```

Pick up the first three values

The `format` function is covered later

If this had been `t` the time would have been printed to the terminal

This function returns the time in nine values: second, minute, hour, date, month, day, and two others

Recursion

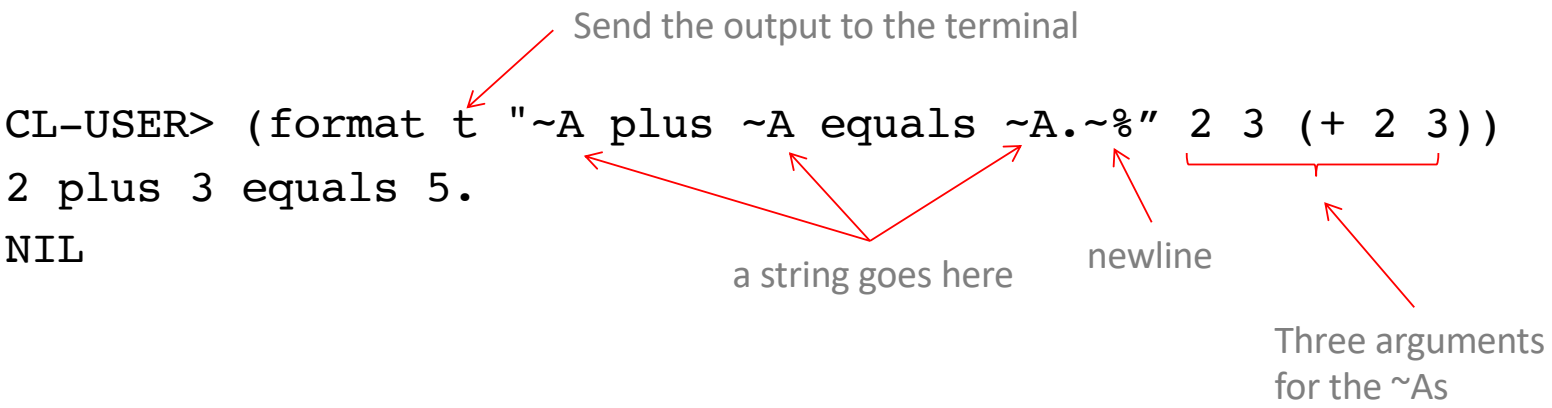
Function to test whether something is an element of a list

```
CL-USER> (defun our-member (obj lst)
           (if (null lst)
               nil
               (if (eql (car lst) obj)
                   lst
                   (our-member obj (cdr lst)))))
CL-USER> (our-member 'b '(a b c))
(BC) ;not NIL so therefore true
CL-USER> (our-member 'z '(a b c))
NIL  ;false
```

`null` is a function which has one argument, a list. It returns `t` if the list is empty; `nil` otherwise (Graham 1996), p. 13.

Input and Output

The `format` function



Input and Output

The `read` function

```
CL-USER> (defun askem (string)
           (format t "~A" string) ;prompt the user
           (read))
CL-USER> (askem "How old are you? ")
How old are you? 29
29
```

Two expressions in this function

← The function returns the value of the last expression evaluated

Variables

- The `let` operator introduces new local variables

```
CL-USER> (let ((x 1) (y 2))
           (+ x y))
3
```

- It has two parts
  - A list of instructions for creating variables
    - Each of the form (*variable expression*)
    - Each variable will be initially set to the value of the expression
    - The variables are valid in the body of the `let` (i.e. they are local variables)
  - The body of expressions
    - Each is evaluated in order
    - The value of the last expression is returned as the value of the `let`

Variables

- The `let` operator introduces new local variables

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
       val
       (ask-number))))
```

```
CL-USER> (ask-number)
Please enter a number. a
Please enter a number. (ho hum)
Please enter a number. 52
52
```

Variables

- The `let*` operator introduces new local variables, the values of which can depend on each other

```
CL-USER> (let* ((x 1)
                (y (+ x 1)))
           (+ x y))
3
```

- A `let*` is functionally equivalent to a series of nested `let`s

- In both `let` and `let*`, initial values default to `nil`

Variables

- The `defparameter` operator introduces new global variables

```
CL-USER> (defparameter *glob* 99)
*GLOB*
```

- Accessible from everywhere
  - Except in expressions with local variable with the same name
  - To distinguish global variables, use the convention of starting and ending a global variable name with an asterisk

Variables

- The `defconstant` operator introduces new global constants

```
CL-USER> (defconstant limit (+ *glob* 1))
```

Assignment

- The `setf` operator assigns values to local and global variables

```
CL-USER> (setf *glob* 98)
98
CL-USER> (let ((n 10))
           (setf n 2) ;overwrites the 10
           n)
2
```

- When the first argument to `setf` is not the name of a local variable, it is taken to be a global variable (and is created, if necessary)

```
CL-USER> (setf x (list 'a 'b 'c))
(A B C)
```

Assignment

- The first argument to `setf` can be an expression
- In this case, the second argument is inserted in the place referred to by the first

```
CL-USER> (setf (car x) 'n)
N
CL-USER> x
(N B C)
```

Assignment

- You can give any (even) number of arguments to `setf`

```
CL-USER> (setf a b
               c d
               e f)
```

Functional Programming

- Writing programs that work by returning values, instead of modifying things

- It is the dominant paradigm in Lisp

```
CL-USER> (setf lst '(c a r a t))
(C A R A T)
CL-USER> (remove 'a lst)
(C R T)
CL-USER> lst
(C A R A T)
```

Removes all the a elements and returns a list

But `lst` is unchanged

Functional Programming

- To remove the element from the list, rather than just returning a list with the value removed:

```
CL-USER> (setf x (remove 'a x))
```

- Functional programming avoids `setf` and similar constructs
- Focus on returning values rather than the side-effects
- Functional programming facilitates interactive testing

Iteration

- The `do` macro is the fundamental iteration operator in Lisp
- Like `let`, `do` can create variables
- The first argument is a list of variable specifications of the form (*variable initial update*)

- Initially, each variable is set to the value of the *initial* expression
- On each iteration, each variable is set to the value of the *update* expression

Iteration

- The second argument is a list containing one or more expressions
- The first of these is used to test whether iteration should stop
- The remaining expressions in this list will be evaluated in order when iteration stops
  - The value of the last expression is returned as the value of the `do`

Iteration

- The remaining arguments comprise the body of the loop
- They are evaluated, in order, on each iteration
- One each iteration the variables are updated,
  then the termination test is evaluated,
  and then (if the test failed) the body is evaluated

Iteration

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))

CL-USER> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Iteration

- The `dolist` macro iterates through the elements of a list
- The first argument is a list of the form (*variable expression*)
- This is followed by a body of expressions
- The body is evaluated with *variable* bound to successive elements of the list returned by *expression*

Iteration

Here is a function that returns the length of a list

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst) ;for each obj in lst
      (setf len (+ len 1)))
    len))
```

Functions as Objects

- In Lisp, functions are regular objects, like symbols, strings, or lists
- If we give the name of a function to `function`, it will return the associated object
- `function` is a special operator so we don't have to quote the argument

```
CL-USER> (function +)
#<Compiled-Function + 17BA4E>
```

Functions as Objects

- Just as we can use ' as an abbreviation for `quote`, we can use #' as an abbreviation for function:

```
CL-USER> #'+
#<Compiled-Function + 17BA4E>
```

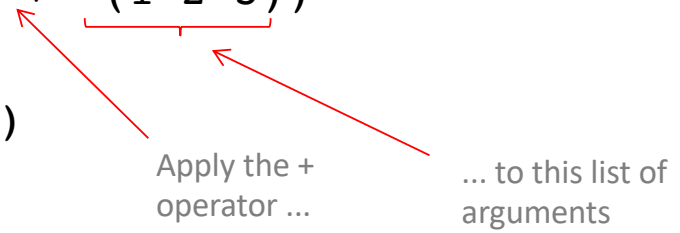
- This abbreviation is known as a sharp-quote

Functions as Objects

- We can pass functions as arguments

- The function `apply` takes a function as an argument, along with a list of arguments for that function, and returns the result of applying the function to the arguments

```
CL-USER> (apply #'+ '(1 2 3))
6
CL-USER> (+ 1 2 3)
6
```



Functions as Objects

- It can be given any number of arguments, so long as the last argument is a list

- The function `apply` takes a function as an argument, along with a list of arguments for that function, and returns the result of applying the function to the arguments

```
CL-USER> (apply #'+ 1 2 '(3 4 5))
15
```

Functions as Objects

- The function `funcall` does the same thing but does not need the arguments to be packaged in a list

```
CL-USER> (funcall #'+ 1 2 3)
6
```

Lambda

- The `defun` macro creates a function and gives it a name
- However, functions don't have to have names, and we don't need `defun` to define them
- We can refer to functions literally using a lambda expression
- A list containing the symbol lambda, followed by a list of parameters, followed by a body of zero or more expressions, e.g.

```
(lambda (x y)
  (+ x y))
```

Lambda

- A lambda expression can be considered as the name of a function and can be the first element of a function call

```
CL-USER> ((lambda (x) (+ x 100)) 1)
101
CL-USER> (funcall #'(lambda (x) (+ x 100)) 1)
101
```

- This notation allows us to use functions without naming them

- We'll see the need for this later

Mapping Functions

- Common Lisp provides several functions for calling functions on the elements of a list, e.g. `mapcar`, `maplist`, `mapc`, and `mapcan`

- The most frequently used is `mapcar`
  - Takes a function and one or more lists
  - Returns the result of applying the function to elements taken from each list (until some list runs out)

```
CL-USER> (mapcar #'(lambda (x) (+ x 10)) '(1 2 3))
(11 12 13)
CL-USER> (mapcar #'list
                 '(a b c)
                 '(1 2 3 4))
((A 1) (B 2) (C 3))
```

Macros

- Macros are operators that are implemented by translation (i.e. text-substitution) called a macro expansion
- Done automatically at compile time, not at runtime
- Macros are defined using `defmacro`
- We'll mainly be using macros rather than writing them

Macros

```
CL-USER> (defmacro nil! (x)  ;macro to set the argument
  (list 'setf x nil))        ;to nil
CL-USER> (nil! x)
NIL
CL-USER> x
NIL
CL-USER> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

This is translated into `(setf x nil)`

CLOS – Common Lisp Object System

- Class — has a name and a number of slots
- Slot — data associated with a class
- Superclass — a class from which other classes inherit slots
- Instance — an instantiation of a class using `make-instance`
- Method — a function associated with a class

CLOS – Common Lisp Object System

Examples: class definition

```
(defclass rectangle ()
  (height width))
```

Class with two slots

```
(defclass circle ()
  (radius center))
```

Class with two slots

```
(defclass colored ()
  (color))
```

Class with one slot

```
(defclass colored-circle (circle colored)
  ())
```

Class `colored-circle` inherits slots from two superclasses (slots `radius`, `center` and `color`)

CLOS – Common Lisp Object System

Examples: class methods

```
(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))
```

`r` is an instance of the class `rectangle`

```
CL-USER> (let ((r (make-instance 'rectangle)))
           (setf (slot-value r 'height) 2
                 (slot-value r 'width) 3)
           (area r))
6
```

`slot-value` provides access to the slots of a class, in this case the `width` slot of instance `r`

CLOS – Common Lisp Object System

Example: use of keyword arguments when defining a class

```
((defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))

CL-USER> (setf c (make-instance 'circle))
#<Circle #XC5C726>
CL-USER> (setf (circle-radius c) 1)
1
CL-USER>(circle-radius c)
1
```

Allows access to the slot value without having to use `slot-value`

CLOS – Common Lisp Object System

Example: use of keyword arguments when defining a class

```
((defclass circle ()
  ((radius :reader circle-radius)
   (center :writer circle-center)))
```

Allows read access to the slot value without having to use `slot-value`

Allows write access to the slot value without having to use `slot-value`

CLOS – Common Lisp Object System

Example: use of keyword arguments when defining a class

```
(defclass circle ()
  ((radius :accessor circle-radius
           :initarg :radius
           :initform 1)
   (center :accessor circle-center
           :initarg :center
           :initform (cons 0 0))))
```

Allows the specification of an initial value when creating an instance with make-instance using the `:radius` keyword

Specifies a default value for the slot

Note that `:initargs` take precedence over `:initforms`.

The default value for the center slot is a `cons` pair

CLOS – Common Lisp Object System

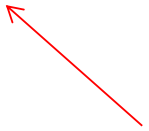
Example: use of keyword arguments when defining a class

```
CL-USER> (setf c (make-instance 'circle :radius 3))
#<Circle #XC2DE0E>
CL-USER> (circle-radius c)
3
CL-USER> (circle-center c)
(0 . 0)
```

CLOS – Common Lisp Object System

We can specify that a slot shares the same value in every instance of a class by declaring it to have `:allocation :class`

```
((defclass circle ()
  ((radius      :accessor circle-radius)
   (center      :accessor circle-center)
   (show-center :allocation :class)))
```



All instances of circle will either show the center point or not because they all share the same value of `show-center`

CLOS – Common Lisp Object System

Multiple inheritance (NB: this is a very shallow treatment)

– We have seen that a class can inherit from several superclasses

```
(defclass colored-circle (circle colored)
  ())
```

Class `colored-circle` inherits slots from two superclasses `radius` and `color`

– A class inherits the union of the slots of its superclasses

  • A superclass may itself have superclasses

CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)

- We represent the class hierarchy of inherited superclasses with a graph
- The precedence list of superclasses is an ordering of the class and its superclasses from most specific to least specific
  - Superclasses nearer to the class being defined are more specific
  - Superclasses farther from the class being defined are less specific
- The precedence list is determined by a traversal of the class hierarchy

CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)

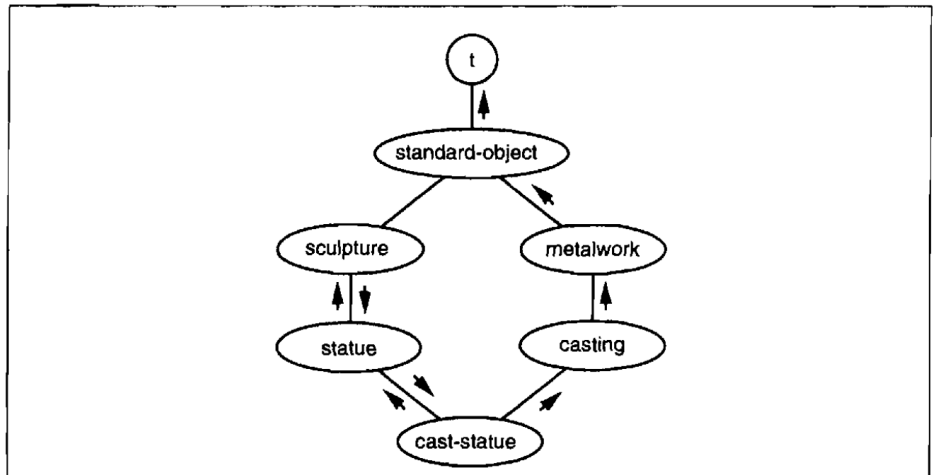


Figure 11.3: Class hierarchy.

Credit: P. Graham, ANSI Common Lisp, Prentice-Hall,1996

CLOS – Common Lisp Object System

Multiple inheritance (very shallow treatment)

1. When there are methods with the same name in the class hierarchy, how do we decided which one to call?

   Use the most specific method for which the classes of the arguments match the specializations of the parameters

2. When there are slots with the same name in the class hierarchy, how do we combine the properties of the slots?

   Use a single slot that combines the properties of the slots in the superclasses according to rules based on the precedence list

Inference

- Facts can be represented by a list comprising
  - Predicate
  - Zero or more arguments

donald is the parent of nancy

```
(parent donald nancy)
```

- Rules tell what can be inferred from the facts we already have

then-part

if-part

```
(<- head body)
```

"If y is the parent of x then x is the child of y"
[Alternatively, we can prove any fact of the form (child x y)
by proving (parent y x)]

```
(<- (child ?x ?y) (parent ?y ?x))
```

Variables are represented as symbols beginning with a question mark

Inference

- The body (if-part) of a rule can be a complex expression

- For example, a rule that if x is the parent of y, and x is male, then x is the father of y, would be written

```
(<- (father ?x ?y) (and (parent ?x ?y) (male ?x)))
```

Inference

- Rules may depend on facts implied by other rules
- The proof of an expression can continue back through any number of rules (so long as it eventually ends up using known facts)
- This is known as backward chaining from what we want to prove to what we already know
- Inference in CRAM is handled by a built-in Prolog interpreter (written in Lisp) in the cram_reasoning package

Packages

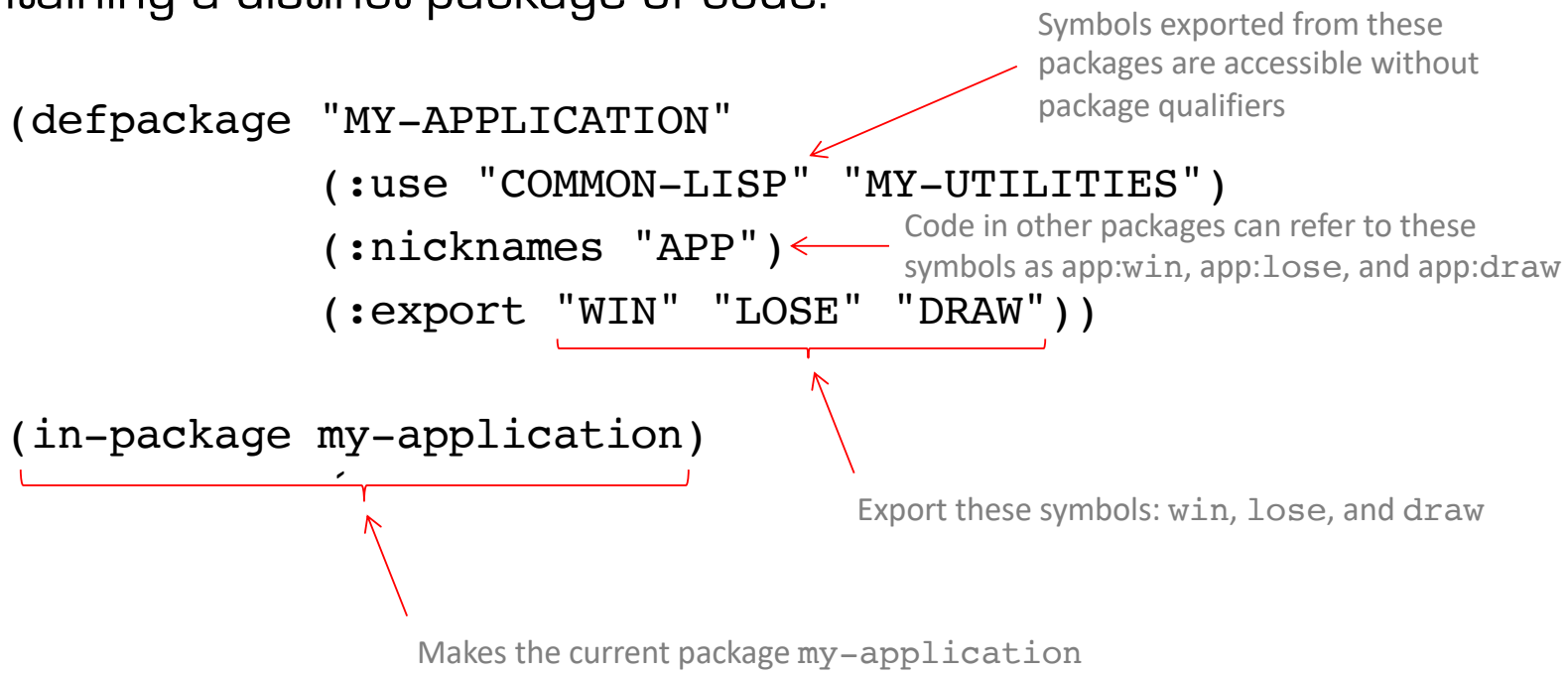
- Large programs are often divided up into multiple packages
- Packages provide the equivalent of a namespace in other languages
  - A symbol defined in one package is local to that package
  - A symbol has to be explicitly exported to be visible in another package
  - An exported symbol usually has to be qualified by in the package using it by preceding it with the name of the package that owns it

Packages

- For example, suppose a program is divided into two packages `math` and `disp`
- If the symbol `fft` is exported by the `math` package
  - Code in the `disp` package will be able to refer to it as `math:fft`
  - In the math package, it will be possible to refer to it as simply `fft`

Packages

The following is an example of what you put at the top of a file containing a distinct package of code:



Recommended Reading

P. Graham. *ANSI Common Lisp*, Prentice-Hall, 1996, Chapter 2.

`http://ep.yimg.com/ty/cdn/paulgraham/acl2.txt`

The Lisp pages on Paul Graham's website:

`http://paulgraham.com/lisp.html`

especially the following:

What Made Lisp Different:

`http://paulgraham.com/diff.html`

Revenge of the Nerds (Essentially, the story of Lisp)

`http://paulgraham.com/icad.html`