Announcements

- CS Department Undergrads
  A decision was made to restrict departmental scholarship awards to students who have a current departmental scholarship application on file. Furthermore, applications dated earlier than January 1 of this year will be considered out of date. The next Scholarship Committee is scheduled for Feb 5th, so update your application before that date.

- Emails
  Please send me an email if you have not already.

- TA: Introducing your TA, Lydia Tapia.

Components

Symbolic expression = ATOM or LIST.

- Atom:
  
  ![image](https://via.placeholder.com/150)

- List:
  
  ![image](https://via.placeholder.com/150)

- NIL: it is an atom and at the same time a list.
  NIL is the same as ()

- T: true, as opposed to NIL.
  See conditionals and predicates.

LISP: A Quick Overview

- Components: Atoms, Lists, and Functions.
- Basics: list, math, etc.
- Arrays and SETQ vs. SETF
- Variable binding
- Lexical vs. dynamic scope
- Conditionals, predicates, iterations, etc.
- User-defined function
- Recursion
- Output

Basics

- quote: returns a literal (i.e. not evaluated) atom or a list.
  
  ![image](https://via.placeholder.com/150)

  Compare with:
  
  ![image](https://via.placeholder.com/150)

- first, second, third, ..., nth, rest:
  
  ![image](https://via.placeholder.com/150)
Basics: List

- **car**: returns first element (atom or list)
  
  (car '(a (b c))) \rightarrow A

  (car '((b c) a)) \rightarrow (B C)

- **cdr**: returns all except the first element of a list, as a list
  
  (cdr '(a (b c))) \rightarrow ((B C))

  (cdr '((b c) a)) \rightarrow (A)

- **Combinations are possible**: cXXXxr where X=(a|r)
  
  (cadr '(a (b c))) == (car (cdr '(a (b c))))) \rightarrow (B C)

- **list**: creates a list out of atoms and lists
  
  (list 'a '(1 2) '((3 5) (7 8))) \rightarrow (A (1 2) ((3 5) (7 8)))

- **length**: number of elements in a list
  
  (length '(a b c)) \rightarrow 3

Basics: Assignments/Arrays

- **setq**: assignment of value to a symbol
  
  (setq x 10) \rightarrow 10

  x \rightarrow 10

- **setf**: can set the value of a symbol (== setq) or location or structure (next slide).

Arrays and SETQ vs. SETF

- **make-array**: create an array

- **aref**: array reference

- **setf**: set value of array element

> (setq a (make-array '(3 3)))

#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))

> (aref a 2 2)

NIL

> (setf (aref a 2 2) 1000)

1000

>a

#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL 1000))

> (setq (aref a 2 2) 1000)

Error: (AREF A 2 ...) is not a symbol.

...
Basics: Math

- \((+ 1 2) (* 3 4) (+ (* 2 3) (/ 4 5))\) etc.
- \((\text{max} 1 2 3 4 5) (\text{min} 4 6 5)\)
- \((\text{sqrt} 16) (\text{expt} 2 3) (\text{round} 3.141592)\)

Basics: File Loading

- \((\text{load} \ "\text{filename}\")\)

Group of Commands

\text{progn} returns the result of the last element, but evaluates all s-expressions in the argument list.

- \((\text{progn} (\text{setq} a 123) (* 5 10)) \rightarrow 50\)
  \(a \rightarrow 123\)

Binding

You can bind variables anywhere in a program with the \text{let} or \text{let*} special forms to create a local context.

- \text{let and let*: lexical scope (local context)}
  \((\text{let} \ (\text{local var list}) \ \text{BODY})\)
  \((\text{let} \ ((x 10) y (z 20)) \ \text{BODY})\)
  \((\text{let*} \ ((x 10) (y (* 2 x)) z)) \ \text{BODY})\)

- Either just a variable or (variable default-value).

- With \text{let*}, values from previous vars can be used to define new value.
  \((\text{let*} \ ((x 10) (y (* 2 x)) z)) \ \text{BODY})\)

Binding: Example

\begin{verbatim}
> (let ((a 3)) (+ a 1))
4
> (let ((a 2)
  (b 3)
  (c 0))
  (setq c (+ a b))
  c)
5
> (setq c 4)
4
> (let ((c 5)) c)
5
> c
4
\end{verbatim}
**Lexical Scope**

Return value according to the lexical scope where it was defined.

```lisp
> (setq regular 5)
5
> (defun check-regular () regular)
CHECK-REGULAR
> (check-regular)
5
> (let ((regular 6)) (check-regular))
5
```

**Dynamic Scope**

Use the defvar to define a special variable that is dynamically scoped.

```lisp
> (defvar *special* 5)
*SPECIAL*
> (defun check-special () *special*)
CHECK-SPECIAL
> (check-special)
5
> (let ((*special* 6)) (check-special))
6
> *special*
5
> (let ((x 23)) (check-special))
```

**Conditionals: the Ps.**

p is for predicate:

- `numberp`, `listp`, `symbolp`, `zerop`, ...
- common comparisons: `<`, `>`,
- `equal`: if the values are the same.
- `eq`: if the memory locations are the same.
- `and`, `or`, `not`: logical operators.

Returns either NIL or T.

**Control Flow**

**IF STATEMENT**

```lisp
(if (> 2 3) ; condition
  (+ 4 5) ; when true
  (* 4 5) ; when false
)
```

**SWITCH STATEMENT**

```lisp
(cond ((testp1) (return-value1)) ; condition 1
  ((testp2) (return-value2)) ; condition 2
  ((testp3) (return-value3)) ; condition 3
  (t (default-value)) ; default
)
Iterations

DOTIMES
(dotimes (index-var upper-bound result-var) BODY)

> (dotimes (k 1 val) (setq val k))
0

> (dotimes (k 10 val) (setq val k))
9

Also find out more about dolist, do, and loop.

Recursion

• Fibonacci number:
  \[ F(N) = F(N-1) + F(N-2), F(1)=1, F(2)=2. \]
  (defun fibo (x)
    (cond
      ((equal x 1) 1)
      ((equal x 2) 2)
      ( (> x 2)
        (+ (fibo (- x 1)) (fibo (- x 2)))))
    )
  )

> (fibo 4)
5
> (fibo 5)
8

Function

• defun: user defined function
  > (defun mult (x y) (* x y))
  DEFUN
  > (mult 10 20)
  200

• Use the let and let* forms:
  (defun mult (x y)
    (let ((tx x) (ty y))
      (* tx ty)
    )
  )

Output

• print: print a string
  (print "hello")

• format: format a string; (format dest string args)
  -%: insert CR
  -S: S-expression
  -A: ascii
  -D: integer
  -widthD: blank space e.g. -5D
  -F: floating point
  -width,decimalF: width and decimal point
Format: examples

> (format t "Hello, world!")
Hello, world!
NIL

> (format nil "Hello, world!")
"Hello, world!"

Format: examples

> (format
  nil
  "The list is ~s and~%the text is ~a"
  (list 'a 'b 'c)
  "This is a string"
)
"The list is (A B C) and
the text is This is a string"

Format: examples

> (format
  nil
  "One:~d~%Two:~f~%Three:~5,2f"
  12 (/ 4 3) (/ 4 3)
)
"One: 12
Two:1.3333334
Three: 1.33"

Dealing with Errors

>aa     <-- error

Error: The variable AA is unbound.
Fast links are on: do (si::use-fast-links nil) for
Error signalled by EVAL.
Broken at EVAL. Type :H for Help.
>>:h     <-- get help
...
>>:q     <-- go back to top level

Top level.
>