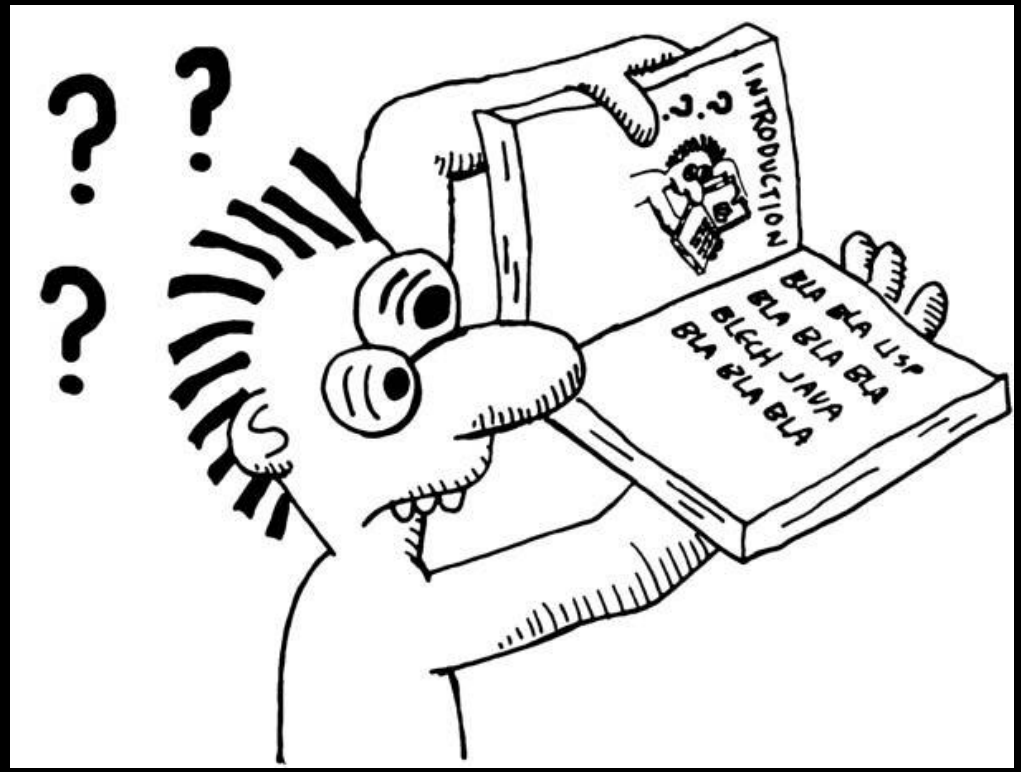
CMPUT 325

Introduction to Lisp



CMPUT 325 Labs

- 3 TAs
  - Ifaz (me): ikabir@ualberta.ca
  - Solimul: mdsolimu@ualberta.ca
  - Spencer: sjkillen@ualberta.ca
- Usually: help sessions for assignments
- Presentations
  - Same TA for all Labs that week

Resources

- Course website
  - Reference Materials
  - Guidelines for assignments
- These slides use some examples and pictures from Land of Lisp, by Conrad Barski

Lisp!

- Family of Languages
  - Common Lisp
- One of the first high level languages
- First language with Garbage Collection
- Dynamic typing

What is there to learn about Lisp?

- Lisp Syntax and Constructs
- Functional Programming

Focus of this course

- Programming with a Garbage Collector
  - Functional Data Structures
- Implementation awareness
  - Tail Calls (recursion as fast as loops)
- Macro System

Not discussed at all in this course

Functional Programming

- Expressions (no statements)
  - Equals with equals
  - Recursion instead of loops
  - Function calls for control flow
- Functions
  - Higher order functions
    - Pass functions as arguments
  - Functions can be created at runtime

Related terms/phrases:
Referential Transparency
Equational Reasoning

Lambda Expressions

Why Functions as Arguments?

- Functions contain control flow
- Functions are composable
- Composable Control Flow!

Other Functional Languages

- Other Lisps
  - Racket/Scheme, Clojure
- ML family of Languages
  - Standard ML, OCaml
  - Type inference!
- Scala
- Haskell

Case-sensitive functions
More natural function calls (no funcall)

Installing SBCL

- Tutorial Video posted last week
- TL;DR
  - Remote: `ssh` *`user`*`@ohaton.cs.ualbert.ca`
  - Ubuntu: `apt install sbcl`
  - Windows: download installer from sbcl.org
  - MacOS: `brew install sbcl`

REPL

- Run `sbcl`
- Brings up an interactive Lisp session
  - Also called a REPL
- Pro tip for Unix:
  - Install `rlwrap`
  - Run `rlwrap sbcl`

Unfamiliar Syntax

- Lisp does not use C-like syntax
  - You'll get used to it
  - You had to learn C-like syntax at one point

Lisp Syntax

- Only one way to organize bits of code
  - Into lists, using parenthesis
- All code written as lists
  - Lisp = LISt Processing
- These lists contain
  - Other lists
  - Numbers, Strings, Symbols

```
(defun square (n)
   (* n n))
```

Primitive Data (1)

- Numbers
  - Integers: `123`
  - Floating point: `456.789`
- Strings
  - “`Hello`”
  - Not used much in this course

Primitive Data (2): Symbols

- A stand alone word
- Fast equality checks
- Case inSENsitive in Common Lisp
  - Historical Reasons
  - Converted to uppercase
- Enter into REPL with a quote
  - E.g. `'hello` → `HELLO`
- Internally used for function names

More on quoting later

Function names also case insensitive

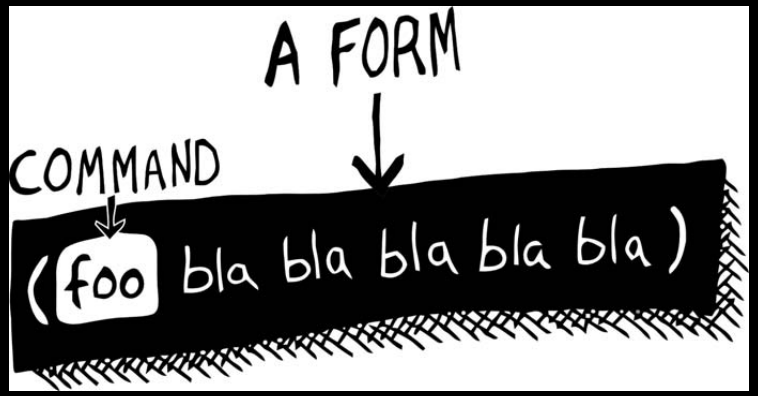
Function Calls

- Function/operator calls
  - Prefix syntax using ( )
  - `(+ 1 2)`
    - Naturally variadic: `(+ 1 2 3 4 5)`
  - `(expt 3 2)`
    - Whitespace separated
    - No commas between argument names

```
> (equal 1 1.0)
NIL
> (+ 1 1.0)
2.0
> (/ 4 6)

> (/ 4.0 6)
```

Code mode

- Default mode in Lisp
- Code is expected to be entered as a form
  - List that starts with a command
- Remaining items sent to command as arguments
- Arguments also in code mode
- Every command in Lisp returns a value




```
> (expt 2 3)
8
> (expt 2 (+ 3 4))
128
```

Data mode - Quoting

- In data mode, what you type in is treated as data
- Use a single quote before an expression to treat it as data
- **Everything** is treated as data
  - functions and variables are ignore (they are treated as symbols)

```
> (expt 2 3)
8

> '(expt 2 3)
(EXPT 2 3)
```

Defining functions

```
(defun function-name (arguments)
   …)
```

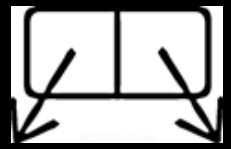
```
> (defun return-five ()
     (+ 2 3))
RETURN-FIVE
```

- Command: `defun`
- Return name of created function
- No return keyword
  - Returns final expression in the function body

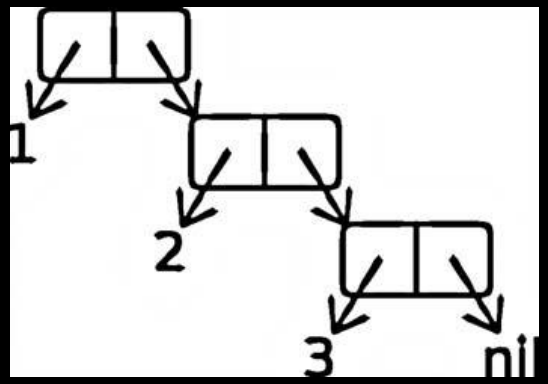
Lists

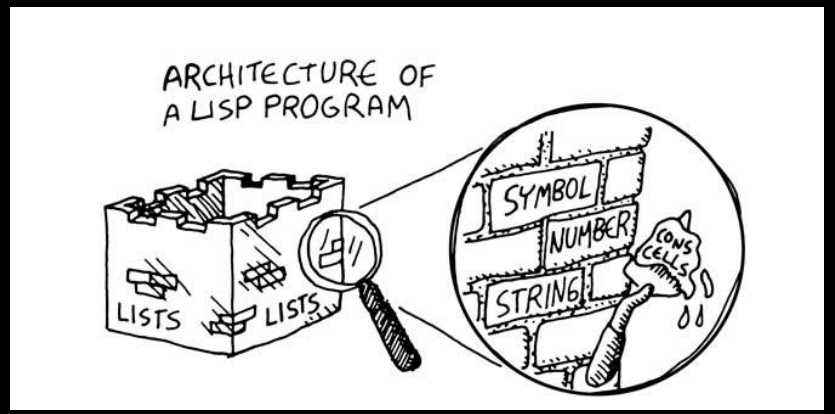
- List hold all Lisp code (and data together)
  - `(expt 2 3)` list with a symbol and two numbers
- Lists are stored in CONS CELLS

Cons Cells



- Primitive Data Structure in Lisp
- Cons cell = 2 connected boxes which can point at other things
- Other things = another cons cell or any type of Lisp data
- List = a series of linked cons cells (linked list)
  - ‘(1 2 3)





List functions

- Manipulating lists/cons cells is very important in Lisp
- Three basic functions
  - `CONS`
  - `CAR`
  - `CDR`
- Empty list represented by symbol `‘nil`

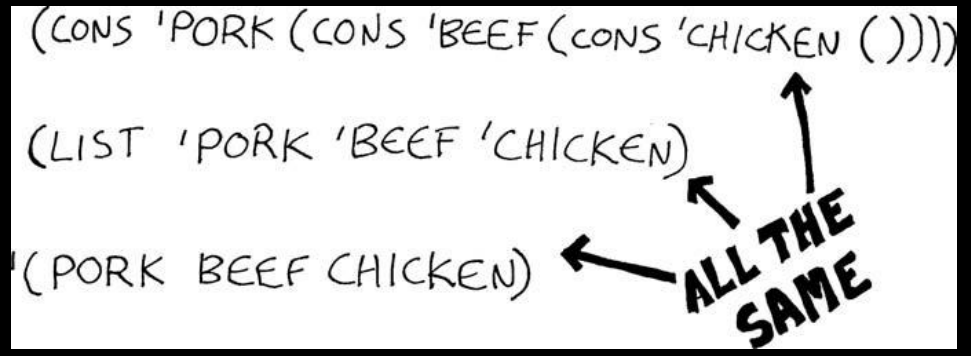
Cons (1)

`(cons left right)`

- Command: `cons`
- **CONS**truct a list
  - A cons cell is allocated which holds the reference to the two linked objects
- `> (cons ‘chicken ‘nil)`
  `(CHICKEN)`
  - When possible, Lisp will show results using lists
- `> (cons ‘chicken ())`
  - What does this do?

Cons (2)

- Can be used to add a new item to the front of the list
  - > (cons ‘pork ‘(beef chicken))
    (PORK BEEF CHICKEN)
  - > (cons ‘beef (cons ‘chicken ())
    (BEEF CHICKEN)
  - > (cons ‘pork (cons ‘beef (cons ‘chicken ())))
    (PORK BEEF CHICKEN)
- Convenience, use List function
  - > (list ‘pork ‘beef ‘chicken)
    (PORK BEEF CHICKEN)

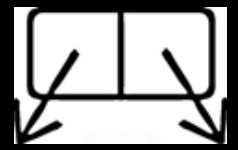


CAR = FIRST and CDR = REST

- CAR is used to get the first item of the list
  - `> (car ‘(pork beef chicken))`
    `PORK`
- CDR is used to get the rest of list
  - Equivalently take the first item away
  - `> (cdr ‘(pork beef chicken))`
    `(BEEF CHICKEN)`
- CAR + CDR = CADR, etc
  - Upto level 4
  - `> (car (cdr ‘(pork beef chicken)))`
    `BEEF`
  - `> (cadr ‘(pork beef chicken))`
    `BEEF`

CAR and CDR for Cons Cells

- Lists are Cons cells
- So CAR and CDR act on cons cells
  - CAR gets first item of a cons box
  - CDR gets second item of a cons box
- Really old machine instructions
  - CAR: Contents of Address Register
  - CDR: Contents of Decrement Register



CAR CAR

Nested Lists

- Lists can contain other lists, eg:
  - '(cat (duck bat) ant)
  - '((peas carrots tomatoes) (pork beef chicken)))
- exercise: create them using cons!
- > (car '((peas carrots tomatoes) (pork beef chicken)))
  (PEAS CARROTS TOMATOES)
- > (cdr '(peas carrots tomatoes))
  (CARROTS TOMATOES)
- > (cdr (car '((peas carrots tomatoes) (pork beef chicken))))
  (CARROTS TOMATOES)
- > (cdar '((peas carrots tomatoes) (pork beef chicken)))
  (CARROTS TOMATOES)

More examples

```
> (cddr '((peas carrots tomatoes) (pork beef chicken) duck))
?

> (caddr '((peas carrots tomatoes) (pork beef chicken) duck))
?

> (cddar '((peas carrots tomatoes) (pork beef chicken) duck))
?

> (cadadr '((peas carrots tomatoes) (pork beef chicken) duck))
?
```

More examples

```
> (cddr '((peas carrots tomatoes) (pork beef chicken) duck))
(DUCK)

> (caddr '((peas carrots tomatoes) (pork beef chicken) duck))
DUCK

> (cddar '((peas carrots tomatoes) (pork beef chicken) duck))
(TOMATOES)

> (cadadr '((peas carrots tomatoes) (pork beef chicken) duck))
BEEF
```

Conditionals (1)

- `IF` command
  - Not a function
    - Does not evaluate all arguments immediately
  - Evaluates first argument
    - If first argument is nil then evaluates third argument
    - Otherwise evaluates second argument

```
> (if (= (+ 1 2) 3)
          ‘yup
          'nope)

YUP

> (if (= (+ 1 2) 4)
          ‘yup
          'nope)

NOPE

> (if ‘(1)
          'the-list-has-stuff-in-it
          'the-list-is-empty)

THE-LIST-HAS-STUFF-IN-IT

> (if ‘()
          'the-list-has-stuff-in-it
          'the-list-is-empty)

THE-LIST-IS-EMPTY
```

Conditionals (2)

- If you want to test more cases -> use COND
- `COND` command:
  - Can handle more than one branch
  - Each branch may contain more than one command

Cond

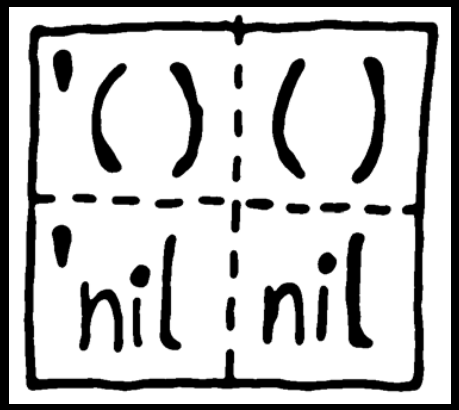
```
> (defun pudding-eater (person)
    (cond
      ((eq person 'henry) '(curse you lisp alien – you ate my pudding))
      ((eq person 'johnny) '(i hope you choked on my pudding johnny))
      (t '(why you eat my pudding stranger ?))))


> (pudding-eater 'johnny)
(I HOPE YOU CHOKED ON MY PUDDING JOHNNY)


> (pudding-eater 'george-clooney)
(WHY YOU EAT MY PUDDING STRANGER ?)
```

Nil and ()

- Empty list = false value = NIL

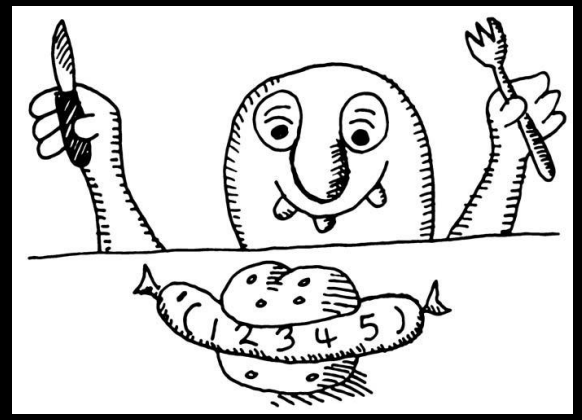
```
– > (if ‘()
        'i-am-true
        'i-am-false)
  I-AM-FALSE

– > (if ‘(1)
        'i-am-true
        'i-am-false)
  I-AM-TRUE
```

- Only false values in Lisp are:



Detecting Nil

- Detect empty list:
  - > (null List)
- Easy to use recursion
  - Until list is empty take first element of the list




```
(defun foo (L)
 (if (null L)
    nil
    (cons
      (first L)
      (foo
        (rest L)))))
```

What is truthy?

- Anything not ‘nil
- Nothing naturally truthy to return
  - Return ‘t or t

Comparison

- Arithmetic comparison: <, >, <=, >=, etc
- Equality operators: `equal`, `eq`, `=`
- Rule of using equals:
  - Use EQ to compare ATOMS
  - Use = to compare numbers
  - Use EQUAL for everything structured

EQ

- EQ: simplest, fast, but only true for equal atoms
  - > (eq 5 5)
    T
  - > (eq 'apple 'apple)
    T
  - > (eq ‘apple ‘banana)
    NIL

=

- = : Numbers, even if they are different types
  - > (= 4 4.0)
    T
  - > (equal 4 4.0)
    NIL

EQUAL

- EQUAL: structural equality
  - ;;comparing symbols
    > (equal 'apple 'apple)
    T
  - ;;comparing integers
    > (equal 5 5)
    T
  - ;;comparing floating point numbers
    > (equal 2.5 2.5)
    T
  - ;;comparing characters
    > (equal #\a #\a)
    T
  - ;;comparing lists
    > (equal (list 1 2 3) (list 1 2 3))
    >T
  - ;;Identical lists created in different ways
    ;;still compare as the same
    > (equal '(1 2 3) (cons 1 (cons 2 (cons 3 ()))))
    T

Equal is useful for doing tests

```
(if (equal (function inputs)
           expected-output))
    ‘passed
    ‘failed)
```

Loading and running files

- Loading a file
  - sbcl --load filename
  - Start sbcl and evaluate (load “filename”)
- Running a file (no repl)
  - sbcl --script filename
  - Turns off certain kinds of debugging

Create and Run

hi.lisp

```
(defun hi ()
   “Hi”)
```

bye.lisp

```
(load “hi.lisp”)
(defun bye ()
   (format t “~a~%” (hi)))
(bye)
```

- Start a session, load hi.lisp, call hi
- Run bye.lisp

Debugging

- Trace command allows to see the stack trace for a given function. Eg:
  - `(trace func1)` → enables tracing for that function
  - `(untrace func1)` → disables tracing

- When you have time (also in Reference Materials page)
  - http://malisper.me/category/debugging-common-lisp/

Notes

- Course website
  - Guidelines for assignments
- Lisp has loops, etc.
  - Avoid procedural Lisp in your assignments!
  - Misses the point of the course

End