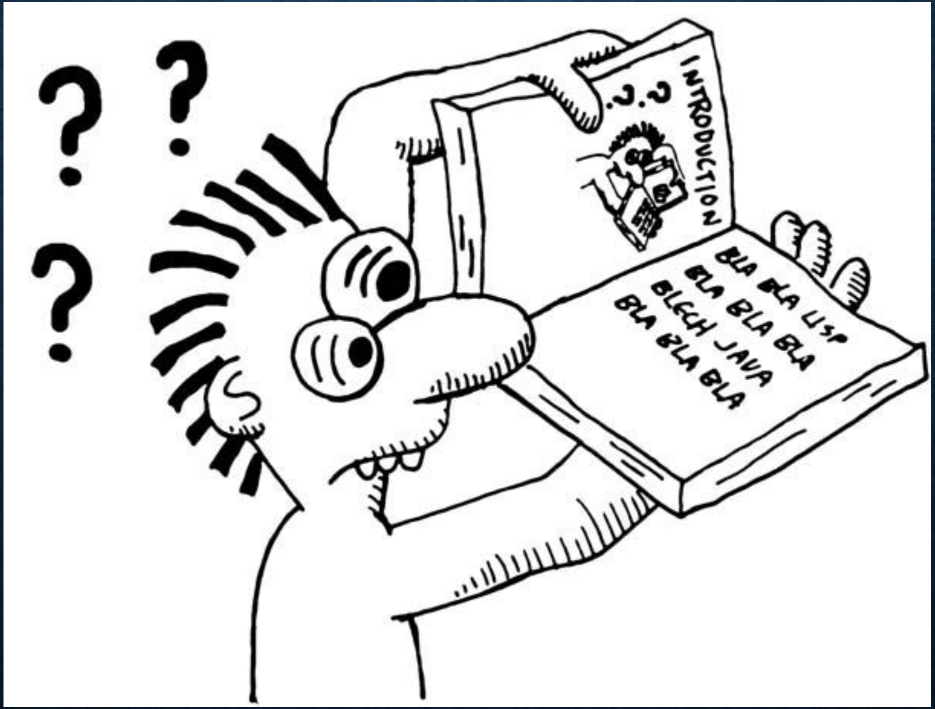
LISP INTRODUCTION

CMPUT 325



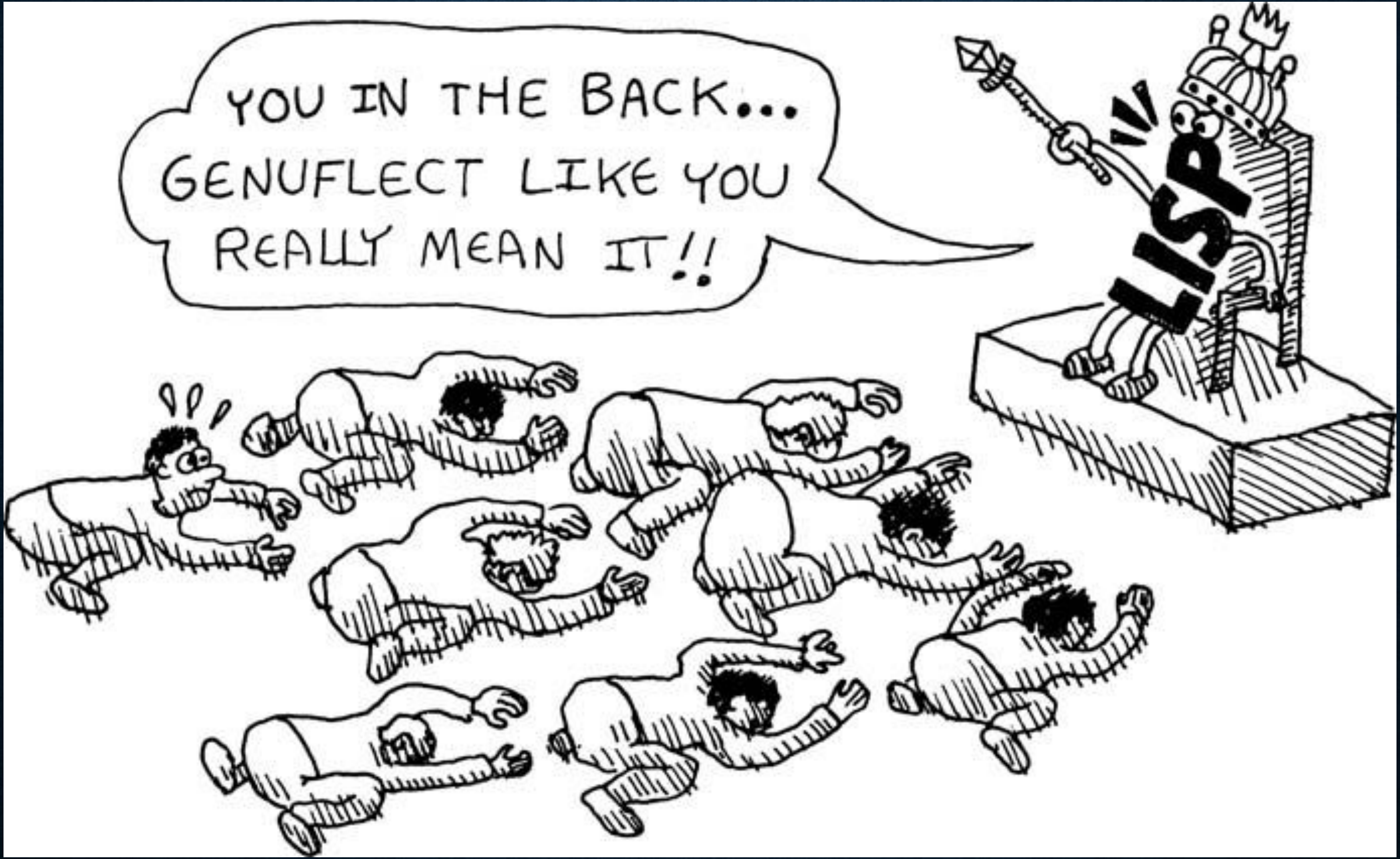
[redacted]

GENERAL INFORMATION

- Two parts:
  - **Lisp**
    - TAs: Arta Seify <seify@ualberta.ca>, Md Solimul Chowdhury <mdsolimu@ualberta.ca>
  - **Prolog**
    - TAs: Arash Karimi <akarimi@ualberta.ca>, Ifaz Kabir <ikabir@ualberta.ca>
- Lisp: two assignments (first one due January 28th)
- Labs:
  - Every week
  - One introductory lab (this one and another one for Prolog)
  - The remaining labs are help sessions for the assignments and general questions
  - Lab presence is optional
- You also can post questions (and answer them) on the forum

WHY LISP?

- Very expressive language, can encode complicated programming ideas in a clear and appropriate way (and using small amounts of code)
- Even if you don’t write Lisp code ever again, learning lisp helps you as a programmer
- Not difficult !
- Altering the compiler/interpreter is very easy, and you can easily
    - Mess around with the language within Lisp
    - Devise new custom commands
    - Write your own object oriented programming support, for example
- How? Writing a Lisp directly in Lisp is very simple!


YOU IN THE BACK... GENUFLECT LIKE YOU REALLY MEAN IT!!
LISP

RESOURCES

- Resources
  - **Reference Materials** (page on main course page)
  - **Guidelines for Assignments** (page on eclass)

- If you want to find out even more
  - David B. Lamkins. *"Successful Lisp: how to understand and use Common Lisp"*.
  - Ulf Nilsson and Jan Maluszynski. "*Logic, Programming and Prolog*" 2nd Ed.

  (Both are free, find links in **Course Outline**)

  - Land of Lisp, by Conrad Barski (I used some examples and pictures in these slides)

STEEL BANK COMMON LISP
SBCL

- Already installed in lab. Type **sbcl** at the command line on the undergrad machines
- For home, download from http://sbcl.org/platform-table.html or ssh to the lab machines
- Like all Common Lisp environments, SBCL takes place into a **read-eval-print loop** (REPL) after you start it up:

```
This is SBCL 1.3.1, an implementation of ANSI Common Lisp. More information about SBCL is available at
<http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty. It is mostly in the public domain; some portions are
provided under BSD-style licenses.  See the CREDITS and COPYING files in the distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded code.  Unfortunately, the development team
currently lacks the time and resources this platform demands.

* (+ 3 (* 2 4))
11
```

- Expressions are immediately evaluated and the resulting value is returned.

LISP SYNTAX

- Only one way of organizing bits of code: into lists, using parentheses

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

- All Lisp code is written as lists

  Lisp = **LIS**t **P**rocessing

- What can we put into these lists?
  - Other lists
  - Symbols
  - Numbers
  - Strings

LISP SYNTAX (2)

- SYMBOLS: fundamental type of data. A stand alone word, typically made of letters, numbers and + - / < _ ? ! etc
  - Case INSENSitive

Eg: foo, bar5, i-like_lisp

```
> (eq 'fooo 'FoOo)
T
```

- NUMBERS: integers and floating point (uses a decimal point)

```
> (equal 1 1.0)
NIL

> (+ 1 1.0)
2.0

> (/ 4 6)   vs.   > (/ 4.0 6)
   2/3      And      0.6666667
```

```
> (expt 99 99)
3697296376497267726571879056288054405
9566876428174110243025997242355257045
5277523421410650010128232727940978889
5483265401194299967694943594516215701
9364401441807106066765930138499977999
9159200499899
```

FUNCTION DEFINITION

```
(defun function_name (arguments)
    ...)
```

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

- Parentheses indentation doesn’t matter but make sure you place all of them in the correct spot
- Every command in Lisp returns a value.
  Defun returns the name of the created function
- There is no return keyword;
  the function created returns the final value calculated in the body of the function

BASIC LISP ETIQUETTE

- You need to surround the command (and its arguments) with parentheses. Otherwise it won't be called!
  - **(+ 1 2)**
  - 3
- All spaces (and line breaks) are ignored when Lisp reads your code. Eg:

```
 > (                    +
          1          2         )
3
```

- There is a file on the course website with more details –
  Guidelines for Assignments

STRINGS (NOT USED MUCH IN 325)

- Surround characters with double quotes. Backslash for escaped characters.
- Can use princ to display a string

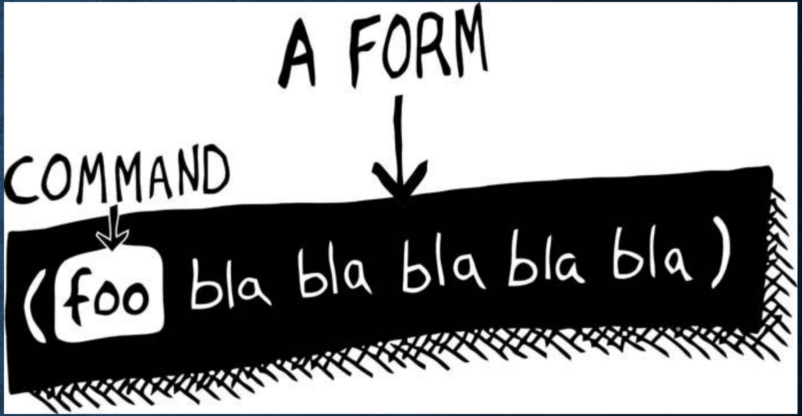
```
> (princ "I like Lisp")
I like Lisp
"I like Lisp“
```

- Text appears twice?
  - First, because of the princ command
  - Second, because REPL always shows the result of the entered expression (and princ function returns the source string as a value)

```
> (princ "He yelled \"Stop that thief!\" from the busy street.")
He yelled "Stop that thief!" from the busy street.
```

CODE MODE

- Default mode in Lisp
- Code is expected to be entered as a **FORM**
    - list that starts with a **COMMAND/FUNCTION**
- All remaining items are sent to the command (function) as PARAMETERS
- These parameters are also in code mode



```
> (expt 2 3)
8

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

DATA MODE - QUOTING

- Treated as data = NOT executed

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

- Use a **single quote** before an expression to prevent evaluation

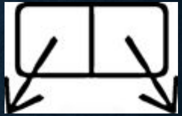
- Everything is treated as data and FUNCTIONS or VARIABLES inside are ignored

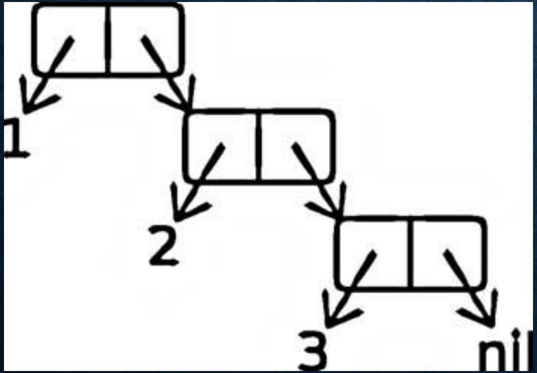
LISTS

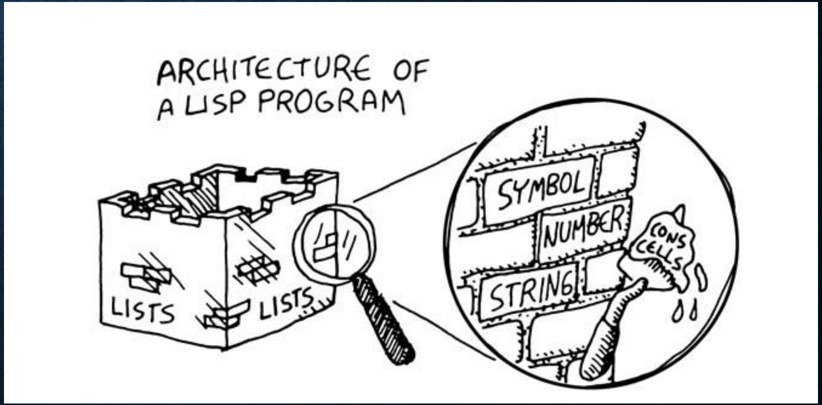
- What holds all Lisp code (and data) together, eg:

  (expt 2 3) a list that contains a symbol and two numbers

- Lists are stored in CONS CELLS
  - a cons cell = 2 connected boxes, which can point at other things
  - other things = another cons cell or any type of Lisp data
  - a list = a series of linked cons cells (linked list)
  - '(1 2 3)







LIST FUNCTIONS

- Manipulating lists is very important in Lisp
- Three basic functions for interacting with lists: **CONS**, **CAR** and **CDR**

- **CONS**
  - CONStruct a list
  - A cons cell is allocated, which will hold the references to the 2 linked objects

```
> (cons 'chicken 'nil)              - nil is used to terminate a list
(CHICKEN)                           - when it can, Lisp will show results using lists

> (cons 'chicken ())                - what does it do? → Same effect!
```

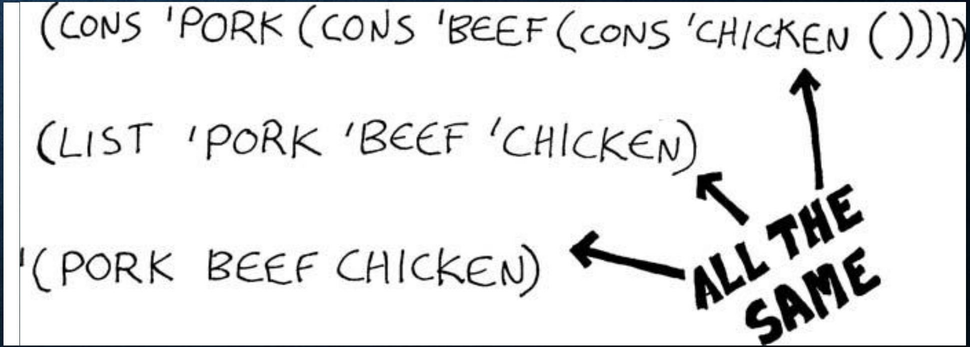
CONS

- Can be used to add a new item to the front of a list

```
> (cons 'pork '(beef chicken))
(PORK BEEF CHICKEN)

> (cons 'beef (cons 'chicken ()))
(BEEF CHICKEN)

> (cons 'pork (cons 'beef (cons 'chicken ())))
(PORK BEEF CHICKEN)
```

- For convenience, use LIST function:

```
> (list 'pork 'beef 'chicken)
(PORK BEEF CHICKEN)
```


(CONS 'PORK (CONS 'BEEF (CONS 'CHICKEN ())))
(LIST 'PORK 'BEEF 'CHICKEN)
'(PORK BEEF CHICKEN)
ALL THE SAME

CAR = FIRST AND CDR = REST

- CAR used to get FIRST list item

```
> (car '(pork beef chicken))
PORK
```

- CDR used to get the REST of list, or equivalently, to take away the FIRST item

```
> (cdr '(pork beef chicken))
(BEEF CHICKEN)
```

- CAR + CDR = CADR, etc … (up to lvl 4)

```
> (cdr '(pork beef chicken))
(BEEF CHICKEN)
> (car '(beef chicken))
BEEF

> (car (cdr '(pork beef chicken)))
BEEF
> (cadr '(pork beef chicken))
BEEF
```

NESTED LISTS

- Lists can contain other lists - “SUBLISTs”, eg:

  '(cat (duck bat) ant)

  '((peas carrots tomatoes) (pork beef chicken)))

```
> (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

- **IF command**

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


> (if '(1)
   'the-list-has-stuff-in-it
   'the-list-is-empty)
THE-LIST-HAS-STUFF-IN-IT
```

- Only one of the expressions after the if is actually evaluated.

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


> (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   AND   Each branch may contain more than one command

```
> (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 ()

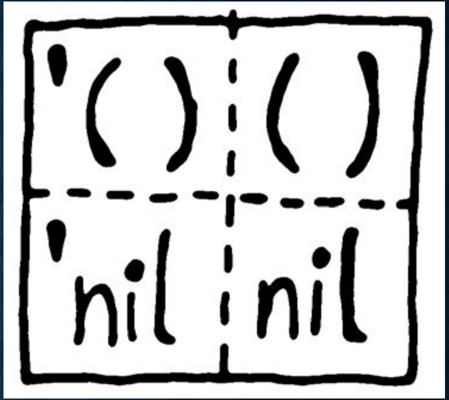
- Empty list = false value = NIL

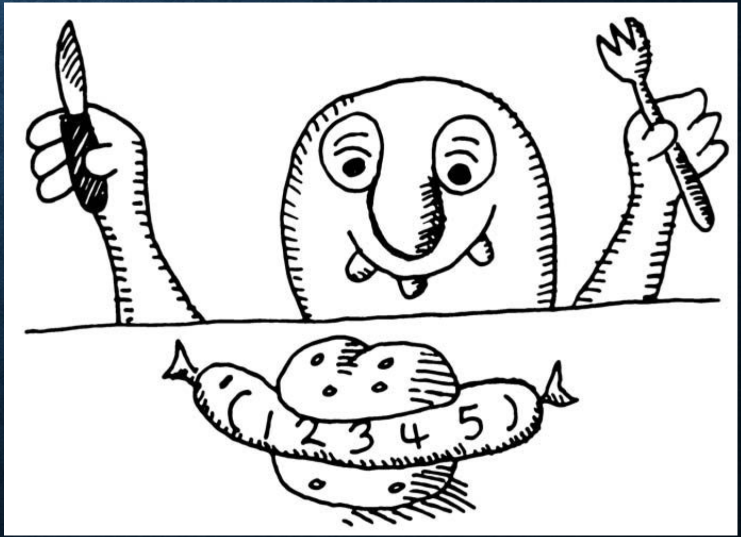
```
> (if '()
    'i-am-true
      'i-am-false)
I-AM-FALSE

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

- Detect empty list: (null List)
- -> easy to use recursion: take first element of the list and process the rest with the recursive function call until list is empty.

- Only false values

  in Lisp are:





COMPARING STUFF: EQ, EQUAL …

- To compare 2 values in Lisp: equal, eq, =

- EQ: simplest, fast, but only true for equal atoms

```
> (eq 5 5)
T
> (eq 'apple 'apple)
T

> (eq ‘apple ‘banana)
NIL
```

- Rule of using equals:
  - Use EQ to compare ATOMS
  - Use = to compare numbers
  - Use EQUAL for everything structured

- EQUAL:

```
;;comparing symbols
> (equal 'apple 'apple)
T

;;comparing lists
> (equal (list 1 2 3) (list 1 2 3))
T
```

EQUAL (CONT.)

;;Identical lists created in different ways ;;still compare as the same

```
> (equal '(1 2 3) (cons 1 (cons 2 (cons 3))))
T
```

;;comparing integers

```
> (equal 5 5)
T
```

;;comparing floating point numbers

```
> (equal 2.5 2.5)
T
```

;;comparing strings

```
> (equal "foo" "foo")
T
```

;;comparing characters

```
> (equal #\a #\a)
T
```

More on equality check: **http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node74.html**

I THINK LITTLE EQ NEEDS A DIAPER CHANGE!
THAT'S LITTLE EQL... CAN'T YOU EVEN RECOGNIZE YOUR OWN CHILDREN??
EQ
=
EQUAL

LOADING A FILE

In terminal, navigate to the file directory and enter

1. sbcl --load *filename*
   - Eg: sbcl --load builtin.lisp
2. sbcl, then write (load "*filename*") (need to have the quotation marks)
   1. Eg: sbcl. (load "builtin.lisp")

DEBUGGING IN SBCL

- **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/