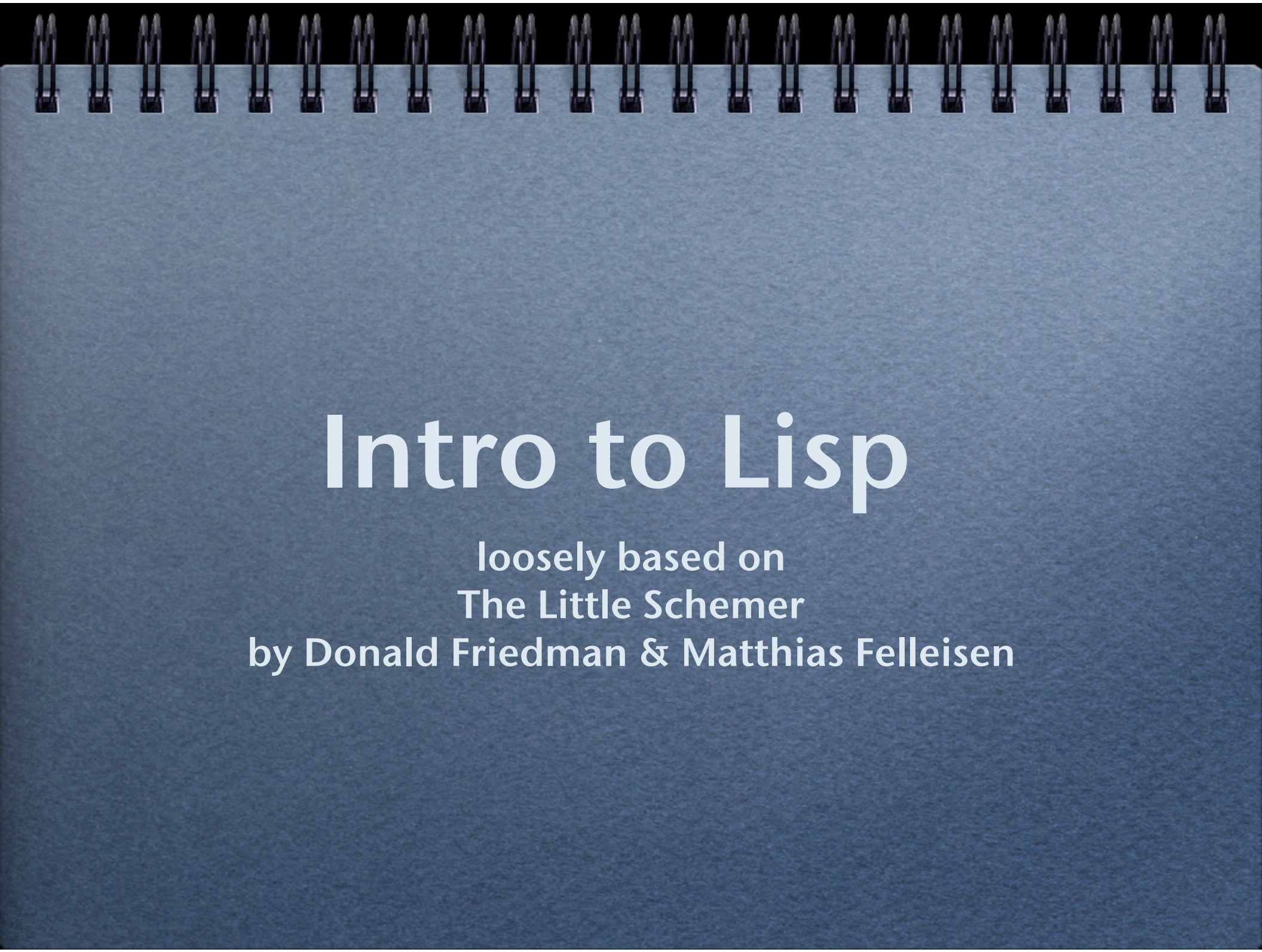

Intro to Lisp
loosely based on
The Little Schemer
by Donald Friedman & Matthias Felleisen

Is this an atom?

- atom
- 1492
- *abc$

Is this an atom?

- atom yes - atom is a string of charcters
- 1492
- *abc$

Is this an atom?

- atom — yes - atom is a string of charcters
- 1492 — yes - 1492 is a string of digits
- *abc$

Is this an atom?

- atom — yes - atom is a string of charcters
- 1492 — yes - 1492 is a string of digits
- *abc$ — yes - *abc$ is a string of characters starting with a special character that is not a left or right paren

Is this an list?

- (atom)
- (atom turkey or)
- (atom turkey) or

Is this an list?

- (atom) yes - (atom) is an atom enclosed by parens
- (atom turkey or)
- (atom turkey) or

Is this an list?

- (atom) yes - (atom) is an atom enclosed by parens
- (atom turkey or) yes - it is a collection of atoms enclosed by parens
- (atom turkey) or

Is this an list?

- (atom) yes - (atom) is an atom enclosed by parens
- (atom turkey or) yes - it is a collection of atoms enclosed by parens
- (atom turkey) or NO - these are two S-expressions - the first is a list and the second is an atom

Is this an list?

- ((atom turkey) or)

Is this an list?

- ((atom turkey) or) yes - the two S-expressions are now enclosed by parens

Is this an S-expression?

- xyz
- (x y z)
- ((x y) z)

Is this an S-expression?

- xyz — yes - all atoms are S-expressions
- (x y z)
- ((x y) z)

Is this an S-expression?

- xyz — yes - all atoms are S-expressions
- (x y z) — yes - because it is a list
- ((x y) z)

Is this an S-expression?

- xyz — yes - all atoms are S-expressions
- (x y z) — yes - because it is a list
- ((x y) z) — yes - all lists are S-expressions

How many S-expression in the list?

- (how are you doing so far?)
- ((how are) ((you doing) so) far?)

How many S-expression in the list?

- (how are you doing so far?)

  6 - how, are, you, doing, so, far?

- ((how are) ((you doing) so) far?)

How many S-expression in the list?

- (how are you doing so far?)

  6 - how, are, you, doing, so, far?

- ((how are) ((you doing) so) far?)

  3 - (how are), ((you doing) so), far?

Is this an list?

- ()
- (() () ())

Is this an list?

- ()

  yes - because it contains zero S-expressions enclosed by parens

- (() () ())

Is this an list?

- ()

  yes - because it contains zero S-expressions enclosed by parens - nil

- (() () ())

  yes - it is a collection of S-expresssions enclosed by parens

Is this an atom?

- ()
- (() () ())

Is this an atom?

- ()

  Well, you might have said "NO - because it is just a list." BUT, an empty list is an exception - it's both an atom & a list.

- (() () ())

Is this an atom?

- ()

  Well, you might have said "NO - because it is just a list." BUT, an empty list is an exception - it's both an atom & a list.

- (() () ())

  NO - because it is just a list

What is the car of *l*

- when *l* is (a b c)
- when *l* is ((a b c) x y z)
- when *l* is hotdog

What is the car of *l*

- when *l* is (a b c)

  a - is the first atom in *l*

- when *l* is ((a b c) x y z)

- when *l* is hotdog

What is the car of *l*

- when *l* is (a b c)

  a - is the first atom in *l*

- when *l* is ((a b c) x y z)

  (a b c) - is the first S-expression in *l*

- when *l* is hotdog

What is the car of *l*

- when *l* is (a b c)

  a - is the first atom in *l*

- when *l* is ((a b c) x y z)

  (a b c) - is the first S-expression in *l*

- when *l* is hotdog

  no answer - there is no car of an atom

The Law of Car

The primitive **CAR** is defined for lists and returns the first *S*-expression of its argument.

What is the car of *l* ?

- when *l* is (((hotdogs)) and relish)

What is the car of *l* ?

- when *l* is (((hotdogs)) and relish)

((hotdogs)) - is the first S-expression and is read as "the list of the list of hotdogs"

What is (car *l*)?

- when *l* is (((hotdogs)) and relish)

((hotdogs)) - is the first S-expression

(car *l*) is an S-expression asking for the car of *l*

What is the car of *l*

- when *l* is ()

  nil - nil is an empty list

  nil - is weird - it is a primitive that is both a list and an atom!

What is the cdr of *l*

- when *l* is (a b c)
- when *l* is ((a b c) x y z)
- when *l* is hotdog

What is the cdr of *l*

- when *l* is (a b c)

  (b c) - is the list *l* without (car *l*)

- when *l* is ((a b c) x y z)

- when *l* is hotdog

What is the cdr of *l*

- when *l* is (a b c)

  (b c) - is the list *l* without (car *l*)

- when *l* is ((a b c) x y z)

  (x y z) - is the list *l* without (car *l*)

- when *l* is hotdog

What is the cdr of *l*

- when *l* is (a b c)

  (b c) - is the list *l* without (car *l*)

- when *l* is ((a b c) x y z)

  (x y z) - is the list *l* without (car *l*)

- when *l* is hotdog

  no answer - there is no cdr of an atom

The Law of Cdr

The primitive **CDR** is defined for lists. The CDR of any list is a list.

What is the cdr of *l*

- when *l* is ()

What is the cdr of *l*

- when *l* is ()

  nil - nil is an empty list

  nil and () are equivalent!!!

  nil - is weird - it is a primitive that is both a list and an atom!

(car (cdr *l*))?

- when *l* is ((b) (x y) ((c)))

(car (cdr *l*))?

- when *l* is ((b) (x y) ((c)))

(x y) - (cdr *l*) returns ((x y) ((c))) and the car of that is (x y)

(cdr (cdr *l*))?

- when *l* is ((b) (x y) ((c)))

(cdr (cdr *l*))?

- when *l* is ((b) (x y) ((c)))

(((c))) - the (cdr *l*) returns ((x y) ((c)))
and the cdr of that is (((c)))

(cdr (car *l*))?

- when *l* is ((b) (x y) ((c)))

- when *l* is (a (b (c)) (d))

(cdr (car *l*))?

- when *l* is ((b) (x y) ((c)))

  nil - the (car *l*) is (b) and the cdr of (b) is nil

- when *l* is (a (b (c)) (d))

(cdr (car *l*))?

- when *l* is ((b) (x y) ((c)))

  nil - the (car *l*) is (b) and the cdr of (b) is nil

- when *l* is (a (b (c)) (d))

  error - the (car *l*) is a which is an atom and car only works with lists

(cons *a* *l*)?

- when *a* is the atom peanut and *l* is the list (butter and jelly)

- when *a* (banana and) and *l* (peanut butter and jelly)

(cons *a* *l*)?

- when *a* is the atom peanut and *l* is the list (butter and jelly)

  (peanut butter and jelly)

- when *a* (banana and) and *l* (peanut butter and jelly)

(cons *a* *l*)?

- when *a* is the atom peanut and *l* is the list (butter and jelly)

(peanut butter and jelly)

- when *a* (banana and) and *l* (peanut butter and jelly)

((banana and) peanut butter and jelly)

(cons *a l*)?

- when *a* is ((help) this) and *l* is the list (is very (hard to learn))

- when *a* (a (b c)) and *l* nil

(cons *a l*)?

- when *a* is ((help) this) and *l* is the list (is very (hard to learn))

  (((help) this) is very (hard to learn))

- when *a* (a (b c)) and *l* nil

(cons *a l*)?

- when *a* is ((help) this) and *l* is the list (is very (hard to learn))

  (((help) this) is very (hard to learn))

- when *a* (a (b c)) and *l* nil

  ((a (b c)))

(cons *a l*)?

- when *a* is g and *l* is nil

(cons *a l*)?

- when *a* is g and *l* is nil

  (g)

The Law of Cons

The primitive **CONS** takes two arguments, the first becomes the first S-expression (element) in the second argument. The first argument can be an atom or a list, the second argument must be a list. The result is a list.

(null *l*)?

- when *l* is the list ()

(null *l*)?

- when *l* is the list ()

  T - stands for true

(null '())?

- when '() is the argument to null

(null '())?

- when '() is the argument to null

T - the ' - quote - suppresses the evaluation of what follows it. Can be written as an S-expression

(null (quote ()) and (null nil)

(null '(a b c))?

- when '(a b c) is the literal argument to null

(null '(a b c))?

- when '(a b c) is the literal argument to null

nil

(null ‘atom)?

- when ‘atom is the literal argument to null

(null ‘atom)?

- when ‘atom is the literal argument to null

nil

The Law of NULL

The primitive NULL takes one argument and returns T when the argument is an empty list; NIL, otherwise.

(atom *a*)?

- when *a* is atom
- when *a* is 1492
- when *a* is *abc$
- when *a* is a non-empty list
- when *a* is nil

(atom *a*)?

- when *a* is atom T
- when *a* is 1492 T
- when *a* is *abc$ T
- when *a* is a non-empty list nil
- when *a* is nil T - weird - remember?

(atom (car *l*))?

- when *l* is (Harry likes apples)
- when *l* is ((Harry) likes apples)

(atom (car *l*))?

- when *l* is (Harry likes apples) T
- when *l* is ((Harry) likes apples) nil

(atom (cdr *l*))?

- when *l* is (Harry likes apples)
- when *l* is ((Harry) likes apples)

(atom (cdr *l*))?

- when *l* is (Harry likes apples) *nil*
- when *l* is ((Harry) likes apples) *nil*

(atom (car (cdr *l*)))?

- when *l* is (Harry likes apples)
- when *l* is (Harry (likes) apples)

(atom (car (cdr *l*)))?

- when *l* is (Harry likes apples) T
- when *l* is (Harry (likes) apples) nil

The Law of ATOM

The primitive ATOM takes one argument and returns T when the argument is an atom; NIL, otherwise.

What do you think LISTP does?

What do you think LISTP does?

The primitive **LISTP** takes one argument and returns **T** when the argument is an list; **NIL**, otherwise.

(equalp *a* *b*)?

- when *a* is apple and *b* is apple
- when *a* is apple and *b* is banana
- when *a* is apple and *b* is (apple)

(equalp *a* *b*)?

- when ***a*** is apple and ***b*** is apple

  T

- when ***a*** is apple and ***b*** is banana

  nil

- when ***a*** is apple and ***b*** is (apple)

  nil - ***a*** is an atom and ***b*** is a list

(equalp (car ***a***) (car (cdr ***b***))?

- when ***a*** is (apple pie) and ***b*** is (apple pie is tasty)
- when ***a*** (apple pie) and ***b*** is ((love those) apple pies)

(equalp (car ***a***) (car (cdr ***b***))?

- when ***a*** is (apple pie) and ***b*** is (apple pie is tasty) nil
- when ***a*** (apple pie) and ***b*** is ((love those) apple pies) T

(equalp (atom nil) (listp nil))?

- (atom nil)
- (listp nil)
- (equalp (atom nil) (listp nil))

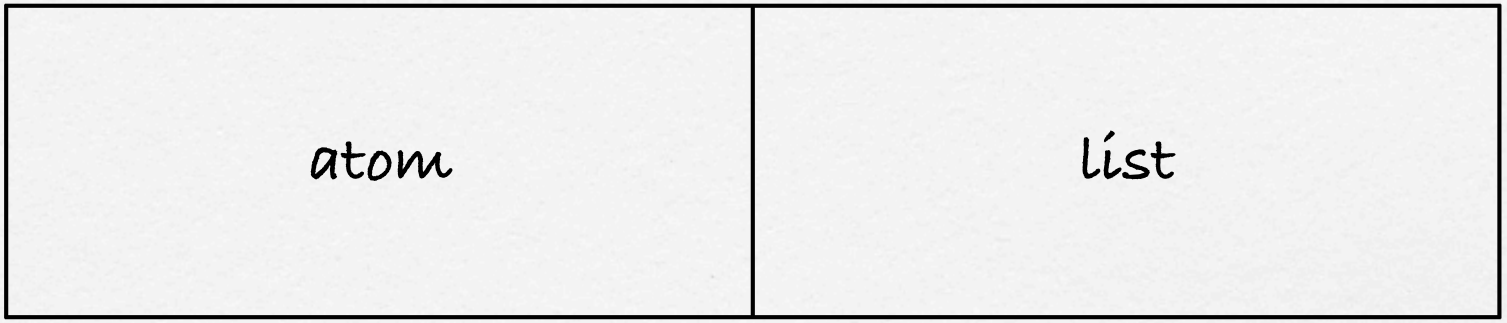
(equalp (atom nil) (listp nil))?

- (atom nil) ⊤
- (listp nil) ⊤
- (equalp (atom nil) (listp nil)) ⊤

New Terms?

atom	list	S-expression
element	argument	empty list
non-empty list	literal	returns
primitive		

Lisp Data Structures



New Lisp Primitives?

car	cdr	cons
nil	T	quote and '
atom	equalp	null
listp		

Lisp Basics

prefix	(function arg1 arg2... argn)	arithmetic operators + - * /
separate by space	no space just after or just before parens	everything gets evaluated

Next Step?

Clozure Cl - download your own personal copy for home	experiment with primitives	keep track of questions and confusions
look over "Common Lisp" an intro text to Lisp (link on Edline)	read the "LISt Processing" section of LispPrimer (link on Edline)	try out the Exercises at the end of the "LISt Processing" section