61A Extra Lecture 8

Thursday, April 2

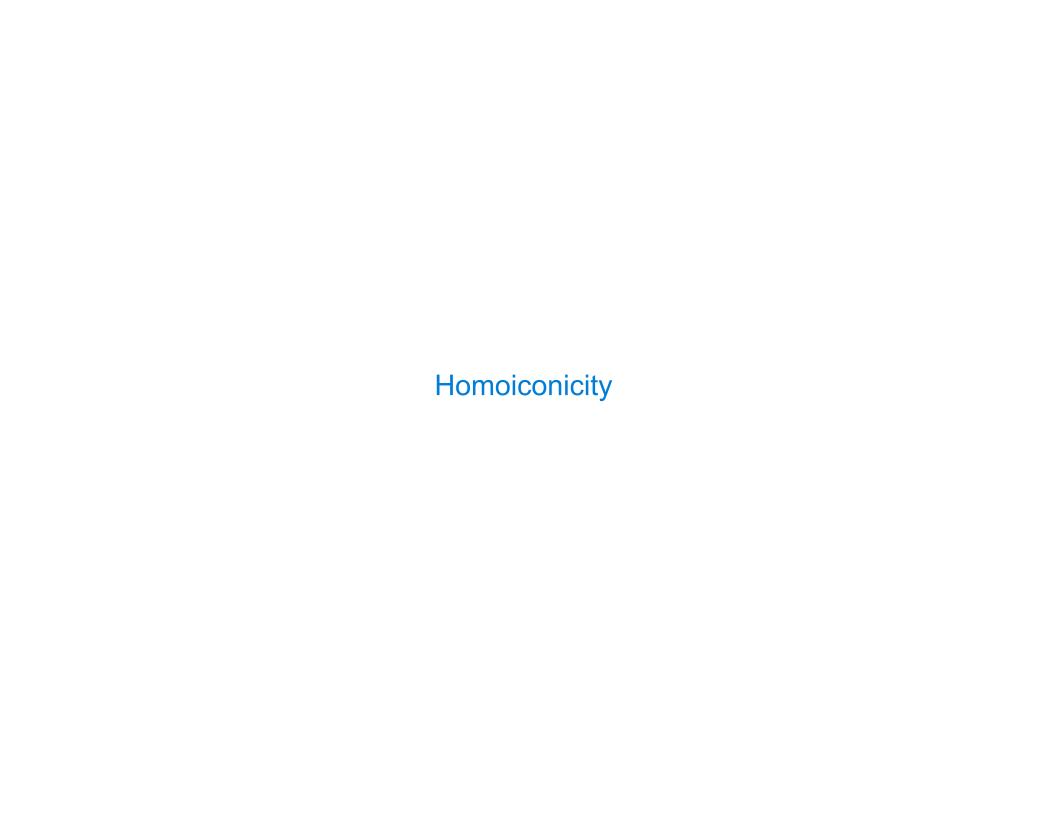
Announcements	

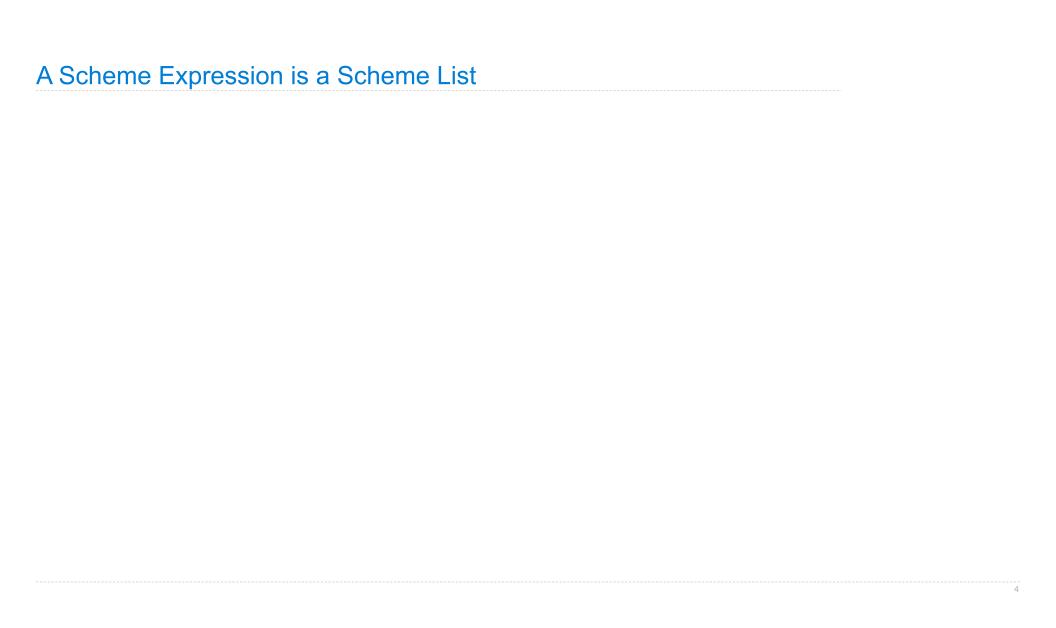
•Extra Homework 3 due Thursday 4/16 @ 11:59pm

- •Extra Homework 3 due Thursday 4/16 @ 11:59pm
 - Extending the object system created in Extra Lecture 6

- •Extra Homework 3 due Thursday 4/16 @ 11:59pm
 - Extending the object system created in Extra Lecture 6
- •Extra Homework 4 due Thursday 4/23 @ 11:59pm (Warning: same day as Project 4 is due!)

- •Extra Homework 3 due Thursday 4/16 @ 11:59pm
 - Extending the object system created in Extra Lecture 6
- •Extra Homework 4 due Thursday 4/23 @ 11:59pm (Warning: same day as Project 4 is due!)
 - •Complete the three extensions to Project 4 described today





Scheme programs consist of expressions, which can be:

Scheme programs consist of expressions, which can be:

• Primitive expressions: 2 3.3 true + quotient

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations

```
scm> (list 'quotient 10 2)
(quotient 10 2)
scm> (eval (list 'quotient 10 2))
5
```

In such a language, it is straightforward to write a program that writes a program

(Demo)

Homoiconic Languages	
	5

Н	lomo	iconic	Langi	uades

Languages have both a concrete syntax and an abstract syntax

Languages have both a concrete syntax and an abstract syntax (Python Demo)

Languages have both a concrete syntax and an abstract syntax

(Python Demo)

A language is *homoiconic* if the abstract syntax can be read from the concrete syntax

Languages have both a concrete syntax and an abstract syntax

(Python Demo)

A language is *homoiconic* if the abstract syntax can be read from the concrete syntax

(Scheme Demo)

Languages have both a concrete syntax and an abstract syntax

(Python Demo)

A language is *homoiconic* if the abstract syntax can be read from the concrete syntax

(Scheme Demo)

Quotation is actually a combination in disguise

Languages have both a concrete syntax and an abstract syntax

(Python Demo)

A language is *homoiconic* if the abstract syntax can be read from the concrete syntax

(Scheme Demo)

Quotation is actually a combination in disguise

(Quote Demo)

Languages have both a concrete syntax and an abstract syntax

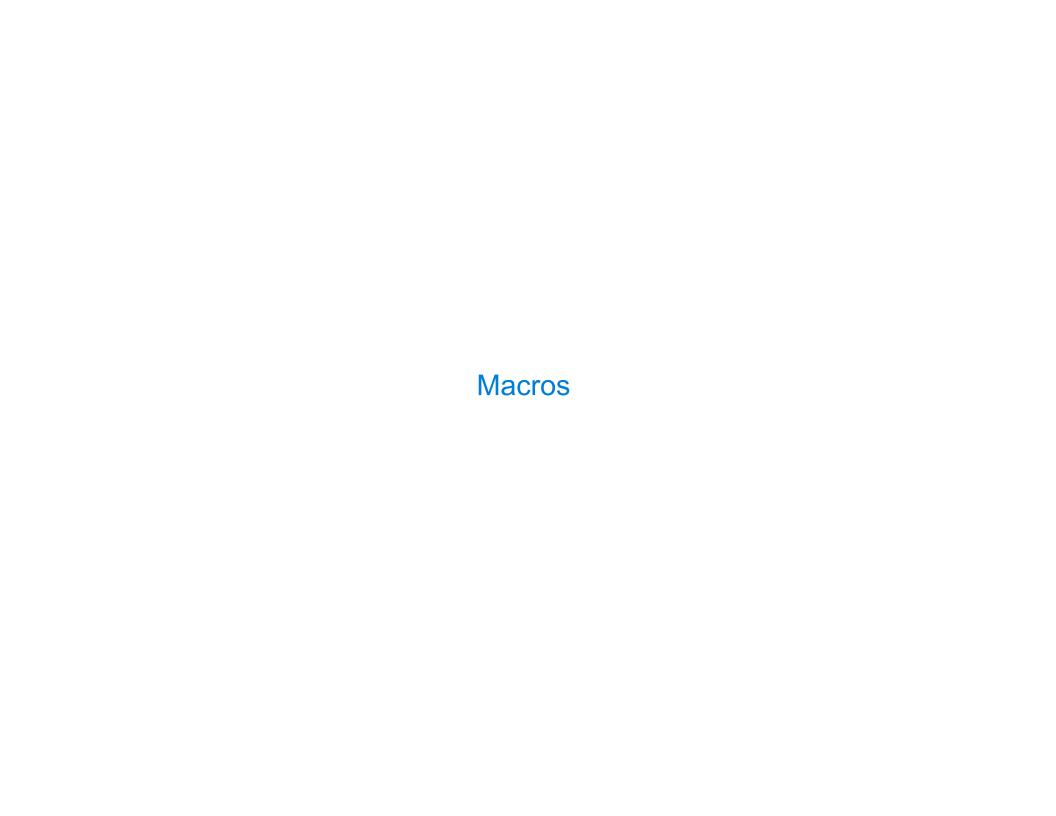
(Python Demo)

A language is *homoiconic* if the abstract syntax can be read from the concrete syntax

(Scheme Demo)

Quotation is actually a combination in disguise

(Quote Demo)



N /			OI	T	
IVIE	acros	Pertorm	Code	ı ransı	formations

A macro is an operation performed on the source code of a program before evaluation

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

• Evaluate the operator sub-expression, which evaluates to a macro

- /

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in homoiconic languages

Scheme has a **define-macro** special form that defines a source code transformation

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

(Demo)

Define a macro that evaluates an expression for each value in a sequence

(define (map fn vals)

8

```
(define (map fn vals)
  (if (null? vals)
```

Define a macro that evaluates an expression for each value in a sequence

8

Define a macro that evaluates an expression for each value in a sequence

8

scm> (for x (* x x) '(2 3 4 5))

(4 9 16 25)

(4 9 16 25)

Define a macro that evaluates an expression for each value in a sequence

8

Quasi-Quoting

(Demo)

Variable-Length Parameter Lists

(Demo)

Define a function nest that builds a nested list containing its arguments

Define a function nest that builds a nested list containing its arguments

(define (nest first . rest)

(if (null? rest)

(3 (4 (5 (6))))

```
Define a function nest that builds a nested list containing its arguments

(define (nest first . rest)

(if (null? rest)

scm> (nest 3)
(3)

scm> (nest 3 4 5 6)
```

```
Define a function nest that builds a nested list containing its arguments
  (define (nest first . rest)
    (if (null? rest)
                                   (list first)
  scm> (nest 3)
  (3)
  scm> (nest 3 4 5 6)
  (3 (4 (5 (6))))
```

```
Define a function nest that builds a nested list containing its arguments
  (define (nest first . rest)
    (if (null? rest)
                                   (list first)
                           (list first (apply nest rest))
  scm> (nest 3)
  (3)
  scm> (nest 3 4 5 6)
  (3 (4 (5 (6))))
```

Temporary Symbols

(Demo)