61A Extra Lecture 8
Announcements
Homoiconicity
A Scheme Expression is a Scheme List
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- **Primitive expressions**: 2  3.3   true   +   quotient
- **Combinations**:  (quotient 10 2)   (not true)
A Scheme Expression is a Scheme List

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
A Scheme Expression is a Scheme List

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
(scm> (list 'quotient 10 2))
```
A Scheme Expression is a Scheme List

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)
```
A Scheme Expression is a Scheme List

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)
```
A Scheme Expression is a Scheme List

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))
```
A Scheme Expression is a Scheme List

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
```
A Scheme Expression is a Scheme List

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
```
A Scheme Expression is a Scheme List

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
A Scheme Expression is a Scheme List

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
cm> (list 'quotient 10 2)
(quotient 10 2)

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

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

(Demo)
Homoiconic Languages
Homoiconic Languages

Languages have both a concrete syntax and an abstract syntax.
Homoiconic Languages

Languages have both a concrete syntax and an abstract syntax

(Python Demo)
Homoiconic Languages

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.
Homoiconic Languages

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)
Homoiconic Languages

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
Homoiconic Languages

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)
Homoiconic Languages

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)
Macros
Macros Perform Code Transformations
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.
Macros Perform Code Transformations

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.
Macros Perform Code Transformations

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.
Macros Perform Code Transformations

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))
```
Macros Perform Code Transformations

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))

> (twice (print 2))
2
2
```
Macros Perform Code Transformations

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))
```

```
> (twice (print 2))
2
2
```

Evaluation procedure of a macro call expression:
Macros Perform Code Transformations

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:

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

Evaluation procedure of a macro call expression:

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

```
> (twice (print 2))
2
2
```
Macros Perform Code Transformations

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))
```

```
> (twice (print 2))
 2
 2
```

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`.
Macros Perform Code Transformations

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))
```

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.

```
> (twice (print 2))
2
2
```
Macros Perform Code Transformations

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))
```

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)
Problem 1

Define a macro that evaluates an expression for each value in a sequence
Problem 1

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

```
(define (map fn vals)
```

Problem 1

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

```
(define (map fn vals)
  (if (null? vals)
    '
    (fn vals)))
```
Problem 1

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

\[
\text{(define (map \textit{fn} \textit{vals})}
\text{(if (null? \textit{vals})}
\text{()})
\]
Problem 1

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

\[
\text{(define (map \textit{fn} \textit{vals})} \\
\text{(if (null? \textit{vals})} \\
\text{() } \\
\text{(cons (fn (car \textit{vals})))}
\]
Problem 1

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

```scheme
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))
)
Problem 1

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

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))
```
Problem 1

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

```lisp
(define (map fn vals)
  (if (null? vals)
    ()
    (cons (fn (car vals))
          (map fn (cdr vals))))
)

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
```
Problem 1

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

```scheme
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))
)

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)
```
Problem 1

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

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)

scm> (for x (* x x) '(2 3 4 5))
(4 9 16 25)
```
Problem 1

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

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)

(define-macro (for sym expr vals)
  (list 'map
        __________________________________________________________________________

scm> (for x (* x x) '(2 3 4 5))
(4 9 16 25)
```
Problem 1

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

```
(define (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals))
            (map fn (cdr vals))))
)

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)

(define-macro (for sym expr vals)
  (list 'map _________________
        (list 'lambda (list sym) expr) vals)_______________)

scm> (for x (* x x) '(2 3 4 5))
(4 9 16 25)
```
Quasi-Quoting

(Demo)
Variable-Length Parameter Lists

(Demo)
Problem 2

Define a function `nest` that builds a nested list containing its arguments
Problem 2

Define a function \texttt{nest} that builds a nested list containing its arguments

\begin{verbatim}
(define (nest first . rest)
  (if (null? rest)
      ________________________________
      ________________________________
      ________________________________
    ))
\end{verbatim}
Problem 2

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)
(3 (4 (5 (6))))
```
Problem 2

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)))))
```
Problem 2

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)