Announcements

- HW9 due tonight

- Ants extra credit due tonight
  - See Piazza for submission instructions

- Hog revisions out, due Monday

- HW10 out tonight
Pairs

Scheme has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

A pair is represented by a dot between the elements, enclosed in parentheses

```plaintext
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```
A recursive list can be represented as a pair in which the second element is a recursive list or the empty list.

Scheme lists are recursive lists:

- **nil** is the empty list.
- A non-empty Scheme list is a pair in which the second element is **nil** or a Scheme list.

Scheme lists are written as space-separated combinations.

```scheme
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4))
(1 2 3 . 4)
```

Not a well-formed list!
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)

(1 2)

No sign of “a” and “b” in the resulting value
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

No sign of “a” and “b” in the resulting value
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
Symbols are normally evaluated to produce values; how do we refer to symbols?

```lisp
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

Quotation prevents something from being evaluated by Lisp

```lisp
> (list 'a 'b)
(a b)
```
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

No sign of “a” and “b” in the resulting value

Symbols are now values
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```lisp
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

Quotation prevents something from being evaluated by Lisp

```lisp
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Quotation can also be applied to combinations to form lists
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

Symbols are now values

Quotation can also be applied to combinations to form lists

> (car '(a b c))
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```lisp
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

Quotation prevents something from being evaluated by Lisp

```lisp
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Quotation can also be applied to combinations to form lists

```lisp
> (car '(a b c))
a
```
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

Symbols are now values

Quotation can also be applied to combinations to form lists

> (car '(a b c))
a
> (cdr '(a b c))
Symbols are normally evaluated to produce values; how do we refer to symbols?

\[
\begin{align*}
&> \text{(define a 1)} \\
&> \text{(define b 2)} \\
&> \text{(list a b)} \\
&\quad (1 \ 2)
\end{align*}
\]

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

\[
\begin{align*}
&> \text{(list 'a 'b)} \\
&\quad (a \ b) \\
&> \text{(list 'a b)} \\
&\quad (a \ 2)
\end{align*}
\]

Symbols are now values

Quotation can also be applied to combinations to form lists

\[
\begin{align*}
&> \text{(car '(a b c))} \\
&\quad a \\
&> \text{(cdr '(a b c))} \\
&\quad (b \ c)
\end{align*}
\]
Scheme Lists and Quotation
Dots can be used in a quoted list to specify the second element of the final pair.
Dots can be used in a quoted list to specify the second element of the final pair

```scheme
> (cdr (cdr (1 2 . 3)))
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```
Dots can be used in a quoted list to specify the second element of the final pair

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists.
Dots can be used in a quoted list to specify the second element of the final pair

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```scheme
> '(1 2 . 3)
```
Dots can be used in a quoted list to specify the second element of the final pair

\[
\text{\texttt{(cdr (cdr } \texttt{'(1 2 . 3))})}
\]

3

However, dots appear in the output only of ill-formed lists

\[
\text{\texttt{'}(1 2 . 3)\texttt{}}
\]

1 2 3
Dots can be used in a quoted list to specify the second element of the final pair

\> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists

\> '(1 2 . 3)
(1 2 . 3)
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

\[ \texttt{(cdr (cdr '(1 2 . 3)))} \]
\[ 3 \]

However, dots appear in the output only of ill-formed lists

\[ \texttt{'(1 2 . 3)} \]
\[ (1 2 . 3) \]
\[ \texttt{'(1 2 . (3 4))} \]
Dots can be used in a quoted list to specify the second element of the final pair

\[
\text{\texttt{\textgreater{} (\texttt{cdr (\texttt{cdr \ '(1 2 . 3)\texttt{))}})}}
\]

3

However, dots appear in the output only of ill-formed lists

\[
\text{\texttt{\textgreater{} \ '(1 2 . 3) \quad 1 \rightarrow 2 3}}
\]

\[
\text{\texttt{\textgreater{} (1 2 . 3) \quad 1 \rightarrow 2}}
\]

\[
\text{\texttt{\textgreater{} \ '(1 2 . (3 4)) \quad 1 \rightarrow 2}}
\]
Dots can be used in a quoted list to specify the second element of the final pair

```scheme
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```scheme
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
```

```
1 2 3
1 2 3 4
1 2 3 4 nil
```
Dots can be used in a quoted list to specify the second element of the final pair

\[
\text{> (cdr (cdr (quote (1 2 . 3))))}
\]

3

However, dots appear in the output only of ill-formed lists

\[
\text{> '(1 2 . 3)}
\]

(1 2 . 3)

\[
\text{> '}(1 2 . (3 4))}
\]

(1 2 3 4)
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
```
Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
```

```
1 2 3
1 2 3
1 2 3 4
1 2 3
```
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

> (cdr (cdr '(1 2 . 3)))
3

However, dots appear in the output only of ill-formed lists

> '(1 2 . 3)  
(1 2 . 3)  
> '(1 2 . (3 4))  
(1 2 3 4)  
> '(1 2 3 . nil)  
(1 2 3)
Dots can be used in a quoted list to specify the second element of the final pair

\[ \texttt{(cdr (cdr '(1 2 . 3)))} \]

3

However, dots appear in the output only of ill-formed lists

\[ \texttt{'(1 2 . 3)} \]

(1 2 . 3)

\[ \texttt{'(1 2 . (3 4))} \]

(1 2 3 4)

\[ \texttt{'(1 2 3 . nil)} \]

(1 2 3)

What is the printed result of evaluating this expression?
Dots can be used in a quoted list to specify the second element of the final pair

\[
\text{> (cdr (cdr '(1 2 . 3)))}
\]

3

However, dots appear in the output only of ill-formed lists

\[
\text{> '(1 2 . 3)}
\]

(1 2 . 3)

\[
\text{> '(1 2 . (3 4))}
\]

(1 2 3 4)

\[
\text{> '(1 2 3 . nil)}
\]

(1 2 3)

What is the printed result of evaluating this expression?

\[
\text{> (cdr '((1 2) . (3 4 . (5))))}
\]
Dots can be used in a quoted list to specify the second element of the final pair

\[
> (\text{cdr} \ (\text{cdr} \ ' (1 \ 2 \ . \ 3)))
\]

3

However, dots appear in the output only of ill-formed lists

\[
> ' (1 \ 2 \ . \ 3)
\]

(1 2 . 3)

\[
> ' (1 \ 2 \ . \ (3 \ 4))
\]

(1 2 3 4)

\[
> ' (1 \ 2 \ 3 \ . \ \text{nil})
\]

(1 2 3)

What is the printed result of evaluating this expression?

\[
> (\text{cdr} \ ' ((1 \ 2) \ . \ (3 \ 4 \ . \ (5))))
\]

(3 4 5)
The Let Special Form

Let expressions introduce a new frame, with the given bindings
The Let Special Form

Let expressions introduce a new frame, with the given bindings

(let ((<name> <exp>) ...) <body>)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let ((<name> <exp>) \ldots) <body>)}
\]
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let ((<name> <exp>) \ldots) <body>)}
\]
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{let } ((\text{name} \text{ exp}) \ldots) \text{ body}
\]

(let ((<name> <exp>) ... <body>)

(define (filter fn s)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let (((name> exp>)) \ldots) body>)}
\]

(define (filter fn s)
  (if (null? s)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let (\((<\text{name}> <\exp>>) \ldots) <\text{body}>))}
\]

\[
(\text{define (filter fn s)}
  \text{(if (null? s)}
  s
\]

\[
(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow \ldots)
\]
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let ((<name> <exp>) \ldots) <body>)}
\]

(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s)))
        \ldots))
  \ldots)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let ((<name> <exp>) \ldots) <body>)}
\]

(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
             (rest (filter fn (cdr s)))))
      (rest (filter fn (cdr s))))
)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
(\text{let \ ((name <exp>) \ ...) \ <body>)}
\]

\[
(\text{define \ (filter fn s)} \n\quad (\text{if \ (null? s)} \n\quad \quad s \n\quad (\text{let \ ((first (car s))} \n\quad \quad (\text{rest \ (filter fn \ (cdr s)}))) \n\quad (\text{if \ (fn \ first)})
\]

```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
             (rest (filter fn (cdr s))))
        (if (fn first)
            ...) ...
        
```
Let expressions introduce a new frame, with the given bindings

\[
\text{(let (((name> <exp>) ...}) <body>)}
\]

(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            ...)))
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let } ((\langle\text{name}\rangle \ \langle\text{exp}\rangle) \ \ldots) \ \langle\text{body}\rangle)\]

\[
\text{(define } (\text{filter }) \text{fn } \text{s})
\text{(if } (\text{null? } \text{s})
\text{s}
\text{(let } ((\text{first } (\text{car } \text{s}))
\text{(rest } (\text{filter } \text{fn } (\text{cdr } \text{s}))))
\text{(if } (\text{fn first})
\text{(cons first rest)
rest))))
\]
The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

```lisp
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest))))

> (filter even? '(1 2 3 4 5 6 7))
```
The Let Special Form

Let expressions introduce a new frame, with the given bindings

\[
\text{(let ((<name> <exp>) ...) <body>)}
\]

```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
           (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest))))
```

> (filter even? '(1 2 3 4 5 6 7))
(2 4 6)
Quick Sort
Quick Sort

Quick sort algorithm:
Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece
Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece
Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces: 
   \(<\ pivot, =\ pivot, >\ pivot\)
3. Recurse on first and last piece
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:  
   < pivot, = pivot, > pivot
3. Recurse on first and last piece
Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

\[
\text{(define (filter-comp comp pivot s)} \\
\quad \text{(filter (lambda (x) (comp x pivot)) s))}
\]
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece

\[(\text{define } (\text{filter-comp comp pivot s})\]
\[\text{(filter (lambda (x) (comp x pivot))) s})\]

\[(\text{define } (\text{quick-sort s})\]
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces: < pivot, = pivot, > pivot
3. Recurse on first and last piece

\[
\text{(define (filter-comp comp pivot s) (filter (lambda (x) (comp x pivot)) s))}
\]

\[
\text{(define (quick-sort s) (if (<= (length s) 1))}
\]
Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces: 
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      ...))
```
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   - < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s))))
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
                (quick-sort (filter-comp = pivot s))
                (quick-sort (filter-comp > pivot s))))))
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
                (filter-comp = pivot s)
                (quick-sort (filter-comp > pivot s))))))
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
                 (filter-comp = pivot s)
                 (quick-sort (filter-comp > pivot s)))))))
The Begin Special Form

Begin expressions allow sequencing
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin} \ <\text{exp}_1> \ <\text{exp}_2> \ \ldots \ <\text{exp}_n>)\]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin} \ <\text{exp}_1> \ <\text{exp}_2> \ \ldots \ <\text{exp}_n>)\]

\[(\text{define} \ (\text{repeat} \ k \ \text{fn})\)]
Begin expressions allow sequencing

\[(\text{begin} \ <\exp_1> \ <\exp_2> \ \ldots \ <\exp_n>)\]

\[(\text{define} \ (\text{repeat} \ k \ \text{fn})\]
\[\ (\text{if} \ (> \ k \ 0)\]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin } \langle\text{exp}_1\rangle \ \langle\text{exp}_2\rangle \ \ldots \ \langle\text{exp}_n\rangle)\]

\[(\text{define} \ (\text{repeat} \ k \ \text{fn}) \\\text{if} \ (> \ k \ 0) \\\text{begin} \ \text{fn} \ \text{repeat} \ (- \ k \ 1) \ \text{fn})\]
The Begin Special Form

Begin expressions allow sequencing

\[ \text{(begin } \langle \text{exp}_1 \rangle \ \langle \text{exp}_2 \rangle \ \ldots \ \langle \text{exp}_n \rangle \text{)} \]

\[ \text{(define } (\text{repeat } k \ \text{fn}) \]
\[ \quad \text{(if } (> \ k \ 0) \]
\[ \quad \quad \text{(begin } \text{fn} \ (\text{repeat } (- \ k \ 1) \ \text{fn})) \]
\[ \quad \quad \quad \text{'done}) \]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin } \langle \text{exp}_1 \rangle \ \langle \text{exp}_2 \rangle \ \ldots \ \langle \text{exp}_n \rangle)\]

\[
\begin{align*}
\text{(define (repeat k fn)} & \\
\quad & \text{(if (> k 0)} \\
\quad & \quad \text{(begin (fn) (repeat (- k 1) fn))} \\
\quad & \quad \text{\'}done\'})
\end{align*}
\]

\[
\begin{align*}
\text{(define (tri fn)} & \\
\end{align*}
\]
Begin expressions allow sequencing

\[
\text{(begin } \langle \text{exp}_1 \rangle \ \langle \text{exp}_2 \rangle \ \ldots \ \langle \text{exp}_n \rangle \text{)}
\]

\[
\text{(define } (\text{repeat } k \ \text{fn})
\quad \text{(if } (> k 0)
\quad \quad \text{(begin } (\text{fn}) \ (\text{repeat } (- k 1) \ \text{fn}))
\quad \quad \text{'done})\)
\]

\[
\text{(define } (\text{tri } \text{fn})
\quad (\text{repeat } 3 \ (\text{lambda } () \ (\text{fn}) \ (\text{lt } 120))))\)\]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin } \text{<exp}_1> \text{<exp}_2> \ldots \text{<exp}_n>)\]

\[(\text{define } (\text{repeat } k \text{ fn})\]
\[\hspace{1em} (\text{if } (> k 0)\]
\[\hspace{2em} (\text{begin } (\text{fn}) (\text{repeat } (- k 1) \text{ fn}))\]
\[\hspace{2em} \text{'done})\]
\[\text{(define } (\text{tri } \text{fn})\]
\[\hspace{1em} (\text{repeat } 3 (\text{lambda } () (\text{fn}) (\text{lt } 120))))\]
\[\text{(define } (\text{sier } d \text{ k})\]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin} \ <\exp_1> \ <\exp_2> \ ... \ <\exp_n>)\]

\[(\text{define} \ (\text{repeat} \ k \ \text{fn}) \n\quad \begin{cases} 
\quad \text{if} \ (> \ k \ 0) \\
\quad \quad \begin{cases} 
\quad \quad \begin{cases} 
\quad \quad \quad \text{(begin} \ \text{fn} \ (\text{repeat} \ (- \ k \ 1) \ \text{fn})) \\
\quad \quad \quad \text{'done})\end{cases} \\
\end{cases} \\
\end{cases}\]

\[(\text{define} \ (\text{tri} \ \text{fn}) \n\quad (\text{repeat} \ 3 \ (\lambda () \ (\text{fn} \ (\text{lt} \ 120))))))\]

\[(\text{define} \ (\text{sier} \ d \ k) \n\quad (\text{tri} \ (\lambda () \ (\text{if} \ (= \ k \ 1) \ (\text{fd} \ d) \ (\text{leg} \ d \ k))))))\]
The Begin Special Form

Begin expressions allow sequencing

\[
\text{(begin } \text{<exp}_1\text{ } \text{<exp}_2\text{ } \ldots \text{<exp}_n\text{)}\]

\[
\text{(define } \text{(repeat } k \text{ } \text{fn}) \\
\text{ (if } (> k 0) \\
\text{ (begin } \text{fn) } \text{(repeat } (- k 1) \text{ fn})) \\
\text{'done}))
\]

\[
\text{(define } \text{(tri } \text{fn}) \\
\text{ (repeat 3 } \text{(lambda } () \text{ } \text{(fn) } \text{(lt 120)))))
\]

\[
\text{(define } \text{(sier } d \text{ } k) \\
\text{ (tri } \text{(lambda } () \text{ } \text{(if } (= k 1) \text{ (fd } d\text{) } \text{(leg } d \text{ } k))))
\]

\[
\text{(define } \text{(leg } d \text{ } k)
\]
The Begin Special Form

Begin expressions allow sequencing

\[
\text{(begin } \langle \text{exp}_1 \rangle \ \langle \text{exp}_2 \rangle \ \ldots \ \langle \text{exp}_n \rangle \text{)}
\]

(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120)))))

(define (sier d k)
  (tri (lambda () (if (= k 1) (fd d) (leg d k))))))

(define (leg d k)
  (sier (/ d 2) (- k 1)) (penup) (fd d) (pendown))