Scheme Lists
Class outline:

- Lists
- Quotation
- List procedures
- Exercises
Scheme lists
Constructing a list

Scheme lists are linked lists.

```
1 -> 2 -> None
```

Python (with our `Link` class:)

```python
class Link:
    pass
```
Constructing a list

Scheme lists are linked lists.

1 → 2

Python (with our Link class):

```
Link(1, Link(2))
```
Constructing a list

Scheme lists are linked lists.

```
1 \rightarrow 2 \rightarrow nil
```

Python (with our Link class):

```
Link(1, Link(2))
```

Scheme (with the cons form):

```
(cons 1 (cons 2 nil))
```

nil is the empty list.

Lists are written in parentheses with space-separated elements:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil)))) ; (1 2 3 4)
```
Accessing list elements

Python access:
Accessing list elements

Python access:

```python
lst = Link(1, Link(2))
lst.first  # 1
lst.rest   # Link(2)
```
Accessing list elements

![List diagram](image)

Python access:

```python
lst = Link(1, Link(2))
lst.first  # 1
lst.rest  # Link(2)
```

Scheme access:

```scheme
(define lst (cons 1 (cons 2 nil)))
(car lst) ; 1
(cdr lst) ; (2)
```

- **car**: Procedure that returns the first element of a list
- **cdr**: Procedure that returns the rest of the list

Remember: "cdr" = "Cee Da Rest"
The list procedure

The built-in list procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

(lis\text{t} 1 2 3) ; (1 2 3)
(lis\text{t} 1 (lis\text{t} 2 3) 4)
(lis\text{t} (cons 1 (cons 2 nil)) 3 4)

Procedure reference: list
The list procedure

The built-in list procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
(list 1 2 3)  ; (1 2 3)
(list 1 (list 2 3) 4)  ; (1 (2 3) 4)
(list (cons 1 (cons 2 nil)) 3 4)
```

Procedure reference: list
The list procedure

The built-in list procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

\[(\text{list} \ 1 \ 2 \ 3) \quad ; \quad (1 \ 2 \ 3)\]
\[(\text{list} \ 1 \ (\text{list} \ 2 \ 3) \ 4) \quad ; \quad (1 \ (2 \ 3) \ 4)\]
\[(\text{list} \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{nil})) \ 3 \ 4) \quad ; \quad ((1 \ 2) \ 3 \ 4)\]

Procedure reference: list
Quotation
Quoting symbols

Symbols typically refer to values:

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

**Quotation** is used to refer to symbols directly:

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

The `' is shorthand for the *quote* form:

```lisp
(list (quote a) (quote b))
```
Quoting symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b)
(list 'a b)
```

The `' is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Quoting symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b)  ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b)  ; (a b)
(list 'a b)
```

The `quote` is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Quoting symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b) ; (a b)
(list 'a b) ; (a 2)
```

The `` is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Quoting symbols

Symbols typically refer to values:

```scheme
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```scheme
(list 'a 'b) ; (a b)
(list 'a b) ; (a 2)
```

The `' is shorthand for the **quote** form:

```scheme
(list (quote a) (quote b)) ; (a b)
```
Quoting lists

Combinations can be quoted to form lists.

'(a b c) ; (a b c)
(car '(a b c))
(cdr '(a b c))

Remember: quoted symbols are not evaluated.
Quoting lists

Combinations can be quoted to form lists.

'(a b c) ; (a b c)
(car '(a b c)) ; a
(cdr '(a b c))

Remember: quoted symbols are not evaluated.
Quoting lists

Combinations can be quoted to form lists.

\[
\begin{align*}
' (a \ b \ c) &; \ (a \ b \ c) \\
(car \ ' (a \ b \ c)) &; \ a \\
(cdr \ ' (a \ b \ c)) &; \ (b \ c)
\end{align*}
\]

Remember: quoted symbols are not evaluated.
List procedures
**length**

length returns the length of a list.

```
(length '(1 2))
(length '())
(length nil)
(length 123)
```

Scheme built-in procedures: List manipulation
length

length returns the length of a list.

(length '(1 2)) ; 2
(length '())
(length nil)
(length 123)

Scheme built-in procedures: List manipulation
length

length returns the length of a list.

(length '(1 2)) ; 2
(length '()) ; 0
(length nil)
(length 123)

Scheme built-in procedures: List manipulation
length

length returns the length of a list.

(length '(1 2)) ; 2
(length '()) ; 0
(length nil) ; 0
(length 123)

Scheme built-in procedures: List manipulation
**length**

`length` returns the length of a list.

```
(length '(1 2)) ; 2
(length '())   ; 0
(length nil)   ; 0
(length 123)   ; Error!
```

**Scheme built-in procedures: List manipulation**
null?

null? returns whether a list is empty or not.

(null? '())
(null? nil)
(null? '(1 2))
(null? 123)

Scheme built-in procedures: Type checking
null?

null? returns whether a list is empty or not.

```
(null? '()) ; #t
(null? nil)
(null? '(1 2))
(null? 123)
```

Scheme built-in procedures: Type checking
null?

null? returns whether a list is empty or not.

```
(null? '()) ; #t
(null? nil) ; #t
(null? '(1 2))
(null? 123)
```

Scheme built-in procedures: Type checking
null?

null? returns whether a list is empty or not.

```
(null? '())       ; #t
(null? nil)       ; #t
(null? '(1 2))    ; #f
(null? 123)       
```

Scheme built-in procedures: Type checking
null?

returns whether a list is empty or not.

```scheme
(null? '()) ; #t
(null? nil)  ; #t
(null? '(1 2)) ; #f
(null? 123)  ; #f
```

Scheme built-in procedures: Type checking
**append**

returns the result of appending the items of all provided lists into a single list in the order provided.

\[
\text{(append (1 2) (3 4))}
\]

\[
\text{(append (1 2) (3 4) (5 6))}
\]

**Scheme built-in procedures: List manipulation**
append

append returns the result of appending the items of all provided lists into a single list in the order provided.

\[
\text{(append '}(1 \ 2) \ '}(3 \ 4)) \quad ; \quad (1 \ 2 \ 3 \ 4)
\]

\[
\text{(append '}(1 \ 2) \ '}(3 \ 4) \ '}(5 \ 6))
\]

Scheme built-in procedures: List manipulation
append returns the result of appending the items of all provided lists into a single list in the order provided.

(append '(1 2) '(3 4)) ; (1 2 3 4)
(append '(1 2) '(3 4) '(5 6)) ; (1 2 3 4 5 6)

Scheme built-in procedures: List manipulation
map

(map <proc> <lst>) returns a new list created by applying proc to each item in lst

(map abs '(-1 -2 3 4))
(map - '(1 2))

Scheme built-in procedures: List manipulation
map

(map <proc> <lst>) returns a new list created by applying proc to each item in lst

(map abs '(-1 -2 3 4)) ; (1 2 3 4)
(map - '(1 2))

Scheme built-in procedures: List manipulation
(map <proc> <lst>) returns a new list created by applying <proc> to each item in <lst>

(map abs '(-1 -2 3 4)) ; (1 2 3 4)  
(map - '(1 2)) ; (-1 -2)

Scheme built-in procedures: List manipulation
filter

(filter <pred> <lst>) returns a new list consisting only of elements of lst for which pred is true.

(filter even? '(0 1 2 3 4 5))
(filter odd? '(0 1 2 3 4 5))

Scheme built-in procedures: List manipulation
filter

(filter <pred> <lst>) returns a new list consisting only of elements of lst for which pred is true.

(filter even? '(0 1 2 3 4 5)) ; (0 2 4)
(filter odd? '(0 1 2 3 4 5))

Scheme built-in procedures: List manipulation
The function `(filter <pred> <lst>)` returns a new list consisting only of elements of `<lst>` for which `<pred>` is true.

```scheme
(filter even? '(0 1 2 3 4 5)) ; (0 2 4)
(filter odd? '(0 1 2 3 4 5)) ; (1 3 5)
```

**Scheme built-in procedures: List manipulation**
**reduce**

\[
(reduce \ <\text{combiner}>\ <\text{lst}>)\]

returns the result of sequentially combining each element in \(\text{lst}\) using \text{combiner} (a two-arg procedure).

\[
(reduce \ +\ '\((1\ 2\ 3\ 4\ 5))\\n(reduce \ \text{expt}\ '\((1\ 2\ 3\ 4\ 5))\\n(reduce \ \text{expt}\ '\((2\ 3\ 4\ 5))
\]

**Scheme built-in procedures: List manipulation**
reduce

(reduce <combiner> <lst>) returns the result of sequentially combining each element in lst using combiner (a two-arg procedure).

(reduce + '(1 2 3 4 5)) ; (15)
(reduce expt '(1 2 3 4 5))
(reduce expt '(2 3 4 5))

Scheme built-in procedures: List manipulation
reduce

\[(\text{reduce} \ <\text{combiner}> \ <\text{lst}>)\] returns the result of sequentially combining each element in \(\text{lst}\) using \text{combiner} (a two-arg procedure).

\[
\begin{align*}
(\text{reduce} + '(1 2 3 4 5)) & ; (15) \\
(\text{reduce} \text{expt} '(1 2 3 4 5)) & ; (1) \\
(\text{reduce} \text{expt} '(2 3 4 5)) &
\end{align*}
\]

Scheme built-in procedures: List manipulation
reduce

(reduce <combiner> <lst>) returns the result of sequentially combining each element in lst using combiner (a two-arg procedure).

(reduce + '(1 2 3 4 5)) ; (15)
(reduce expt '(1 2 3 4 5)) ; (1)
(reduce expt '(2 3 4 5)) ; (1152921504606846976)

Scheme built-in procedures: List manipulation
List equality

(define list1 '(a b c))
(define list2 '(a b c))

For lists, (eq? a b) returns whether a and b are the same list in memory.

(eq? list1 list2)

Scheme built-in procedures: Boolean operations
List equality

(define list1 '(a b c))
(define list2 '(a b c))

For lists, (eq? a b) returns whether a and b are the same list in memory.

(eq? list1 list2) #f

Scheme built-in procedures: Boolean operations
List equality

(define list1 '(a b c))
(define list2 '(a b c))

For lists, \texttt{(eq? a b)} returns whether \texttt{a} and \texttt{b} are the same list in memory.

\texttt{(eq? list1 list2)} #f

While \texttt{(equal? a b)} returns whether \texttt{a} and \texttt{b} are equivalent. Two lists are considered equivalent if \texttt{(car a)} is equivalent to \texttt{(car b)} and \texttt{(cdr a)} is equivalent to \texttt{(cdr b)}.

\texttt{(equal? list1 list2)}

\textbf{Scheme built-in procedures: Boolean operations}
List equality

(define list1 '(a b c))
(define list2 '(a b c))

For lists, (eq? a b) returns whether a and b are the same list in memory.

(eq? list1 list2) #f

While (equal? a b) returns whether a and b are equivalent. Two lists are considered equivalent if (car a) is equivalent to (car b) and (cdr a) is equivalent to (cdr b).

(equal? list1 list2) #t

Scheme built-in procedures: Boolean operations
Exercises
North of equator?

Implement `(north_of_eq point)`, a procedure that takes `point`, a two-element list with a latitude and longitude, and returns whether `point` is north of the Equator.

```
(define (north_of_eq point)
  )

(expect (north_of_eq '(67 10)) #t)
(expect (north_of_eq '(67 -10)) #t)
(expect (north_of_eq '(-67 10)) #f)
(expect (north_of_eq '(-67 -10)) #f)
```
North of equator? (Solution)

Implement `(north_of_eq point)`, a procedure that takes `point`, a two-element list with a latitude and longitude, and returns whether `point` is north of the Equator.

```
(define (north_of_eq point)
    (> (car point) 0))
(expect (north_of_eq '(67 10)) #t)
(expect (north_of_eq '(67 -10)) #t)
(expect (north_of_eq '(-67 10)) #f)
(expect (north_of_eq '(-67 -10)) #f)
```
All north?

Implement `(all_north_of_eq points)`, a procedure that takes `points`, a list of two-element lists, and returns whether all the `points` are north of the equator.

```scheme
(define (all_north_of_eq points)
)
```

- `(expect (all_north_of_eq '( (67 10) (14 43) (37 -122))) #t)`
- `(expect (all_north_of_eq '( (-67 10) (14 43) (37 -122))) #f)`
- `(expect (all_north_of_eq '( (67 10) (14 43) (-37 -122))) #f)`
- `(expect (all_north_of_eq '()) #t)`
All north? (Solution 1)

Implement `(all_north_of_eq points)`, a procedure that takes `points`, a list of two-element lists, and returns whether all the `points` are north of the equator.

```scheme
(define (all_north_of_eq points)
  (= (length (filter north_of_eq points)) (length points))
)

(expect (all_north_of_eq '( (67 10) (14 43) (37 -122))) #t)
(expect (all_north_of_eq '( (-67 10) (14 43) (37 -122))) #f)
(expect (all_north_of_eq '( (67 10) (14 43) (-37 -122))) #f)
(expect (all_north_of_eq '()) #t)
```
All north? (Solution 2)

Implement `(all_north_of_eq points)`, a procedure that takes `points`, a list of two-element lists, and returns whether all the `points` are north of the equator.

```scheme
(define (all_north_of_eq points)
  (cond
    ( (null? points) #t)
    ( (north_of_eq (car points)) (all_north_of_eq (cdr points)))
    (else #f))
)

(expect (all_north_of_eq '((67 10) (14 43) (37 -122))) #t)
(expect (all_north_of_eq '((-67 10) (14 43) (37 -122))) #f)
(expect (all_north_of_eq '((67 10) (14 43) (-37 -122))) #f)
(expect (all_north_of_eq '()) #t)
```
Countdown list

Implement countdown_list, a procedure which takes a number n and returns a list with all the numbers from n down to 1.

```
(define (countdown_list n)

)

(expect (countdown_list 3) (3 2 1))
(expect (countdown_list 1) (1))
```
Countdown list (Solution)

Implement `countdown_list`, a procedure which takes a number \( n \) and returns a list with all the numbers from \( n \) down to 1.

```
(define (countdown_list n)
  (if (= n 0) nil
      (cons n (countdown_list (- n 1))))
)

(expect (countdown_list 3) (3 2 1))
(expect (countdown_list 1) (1))
```
Countup list

Implement \texttt{countup\_list}, a procedure which takes a number \texttt{n} and returns a list with all the numbers from 1 up to (and including) \texttt{n}.

\begin{verbatim}
(define (countup_list n)

)

(expect (countup_list 3) (1 2 3))
(expect (countup_list 1) (1))
\end{verbatim}
Countup list (Solution)

Implement \texttt{countup\_list}, a procedure which takes a number \texttt{n} and returns a list with all the numbers from 1 up to (and including) \texttt{n}.

\begin{verbatim}
(define (countup_list n)
  (if (= n 0) nil
      (append (countup_list (- n 1)) (cons n nil))
  )
)

(expect (countup_list 3) (1 2 3))
(expect (countup_list 1) (1))
\end{verbatim}