Lists

- Aggregate data structures are built from **lists**
- *Examples:*

`(1 2 3)`	A list of 3 numbers
`()`	An empty list
`((1 2) (3 4))`	A list of 2 lists: `(1 2)`, `(3 4)`
`(1 (2 3) 4)`	A mixed list containing 3 members: `1`, `(2 3)` and `4`
`(())`	A singleton list containing `()`

Quoting

- How do we express list constants?

```
> (1 2 3)
```

Quoting

- How do we express list constants?

```
> (1 2 3)
```

Error!

- Scheme thought you were trying to apply the procedure **1** to arguments **2** and **3**.

Quoting

- How do we express list constants?

```
> (1 2 3)
Error!
```

- Scheme thought you were trying to apply the procedure **1** to arguments **2** and **3**.
- Thus, quoting . . .

```
> (quote (1 2 3))
(1 2 3)
```

Quoting

- How do we express list constants?

```
> (1 2 3)
Error!
```

- Scheme thought you were trying to apply the procedure **1** to arguments **2** and **3**.
- Thus, quoting . . .

```
> (quote (1 2 3))
(1 2 3)
```

- A more convenient form . . .

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

List Constructors

- **'()** evaluates to an empty list.
- **`(cons`** x L**`)`**

  Argument(s):

  x**:** any data object
  L**:** a list

  Return: A list with x as the first member, followed by the members of L.

List Constructors

- **’()** evaluates to an empty list.
- `(cons` x L`)`

  Argument(s):

  x: any data object
  L: a list

  Return: A list with x as the first member, followed by the members of L.

  Examples:

```
> (cons 1 '(2 3))
(1 2 3)
```

List Constructors

- **'()** evaluates to an empty list.
- **`(cons`** x L**`)`**

  Argument(s):

  x**:** any data object
  L**:** a list

  Return: A list with x as the first member, followed by the members of L.

  Examples:

```
> (cons 1 '(2 3))
(1 2 3)
> (cons 1 '())
(1)
```

List Constructors

cons constructs memory objects which hold two values or pointers to two values.

These objects are referred to as (cons) cells, conses, non-atomic s-expressions ("NATSes"), or (cons) pairs.

In Lisp jargon, the expression "to cons x onto y" means to construct a new object with (cons x y).

- **'()** evaluates to an empty list.
- **(cons** x L**)**

  Argument(s):

  x: any data object
  L: a list

  Return: A list with x as the first member, followed by the members of L.

  Examples:

```
> (cons 1 '(2 3))
(1 2 3)
> (cons 1 '())
(1)
> (cons '(1 2) '(3 4))
((1 2) 3 4)
```

Selector: `car`

- **`(car` L `)`**

  Argument(s):

  L: a non-empty list

  Return: the first element of L

- *Examples:*

```
> (car '(1 2 3))
1
> (car '((1 2) 3 4))
(1 2)
> (car (car '((1 2) 3 4)))
1
> (car (cons 1 '(2 3)))
1
```

cons constructs memory objects which hold two values or pointers to two values.

These objects are referred to as (cons) cells, conses, non-atomic s-expressions ("NATSes"), or (cons) pairs.

In Lisp jargon, the expression "to cons x onto y" means to construct a new object with (cons x y).

The resulting pair has a left half, referred to as the car (the first element, or contents of the address part of register), and a right half, referred to as the cdr (the second element, or contents of the decrement part of register).

Selector: `cdr`

- `(cdr` L`)`

  Argument(s):

  L**:** a non-empty list

  Return: a list containing all but the first element of L

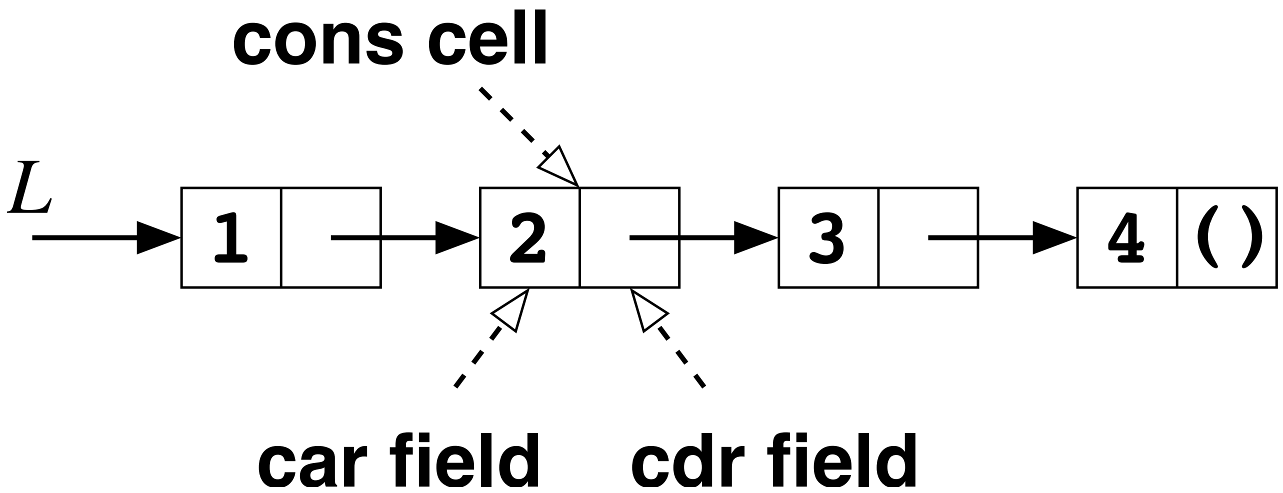
- *Examples:*

```
> (cdr '(1 2 3))
(2 3)
> (cdr '((1 2) 3 4))
(3 4)
> (cdr (cdr '((1 2) 3 4)))
(4)
> (cdr (cons 1 '(2 3)))
(2 3)
```

How are Lists Represented Internally?

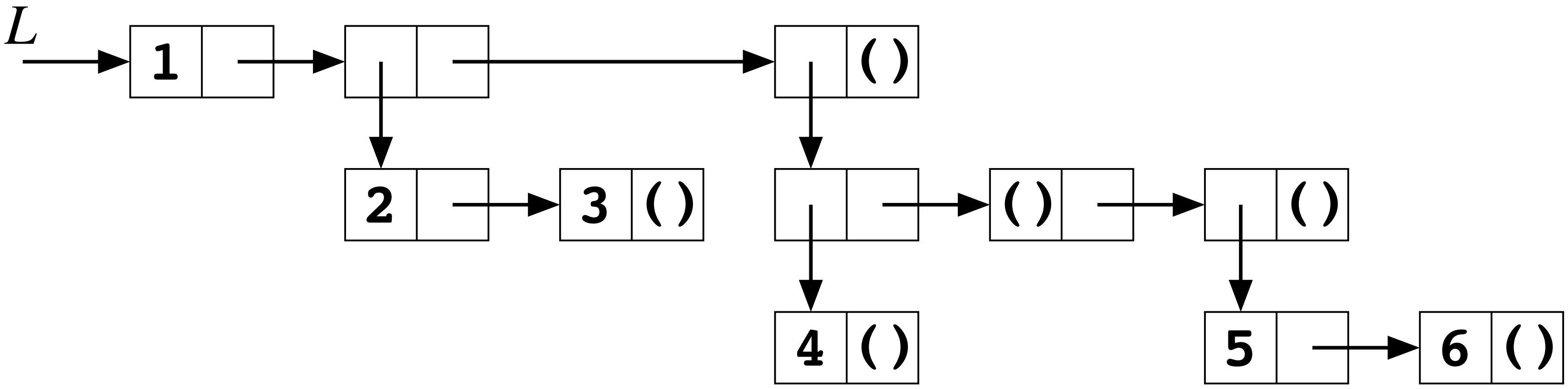
```
(define L '(1 2 3 4))
```

- $L =$ **(1 2 3 4)**



A Complex Example

- $L =$ `(1 (2 3) ((4) () (5 6)))`



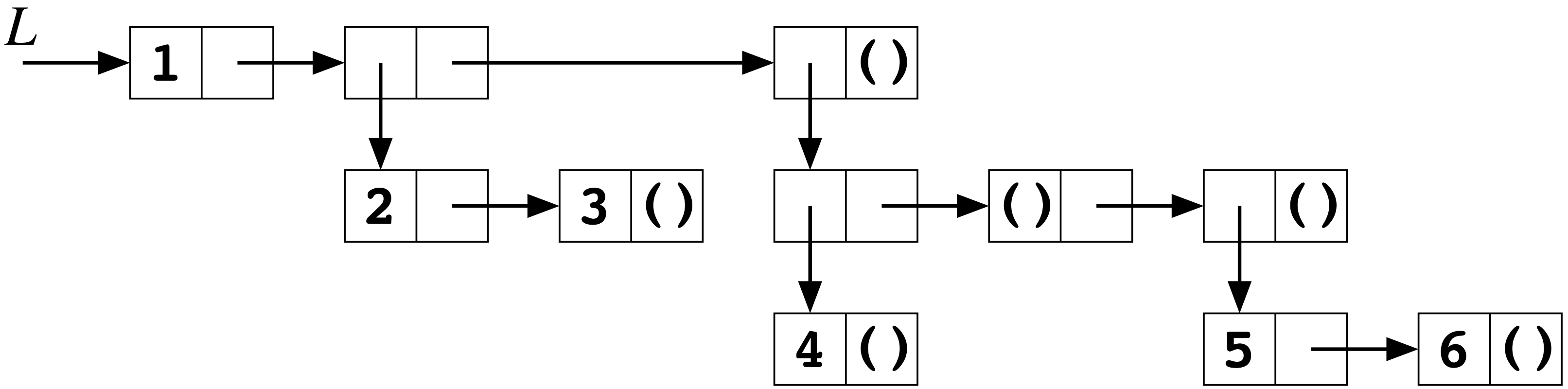
An Alternative Notation for List Constants

- The previously discussed notation for list constants (i.e., `(1 2 3)`) is very user-friendly, but it does not make explicit the internal representation of lists.
- An equivalent, more explicit, but less user-friendly notation:
  Dotted Pairs: `(car . cdr)`
- *Examples:*

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

Exercise

- $L =$ **(1 (2 3) ((4) () (5 6)))**

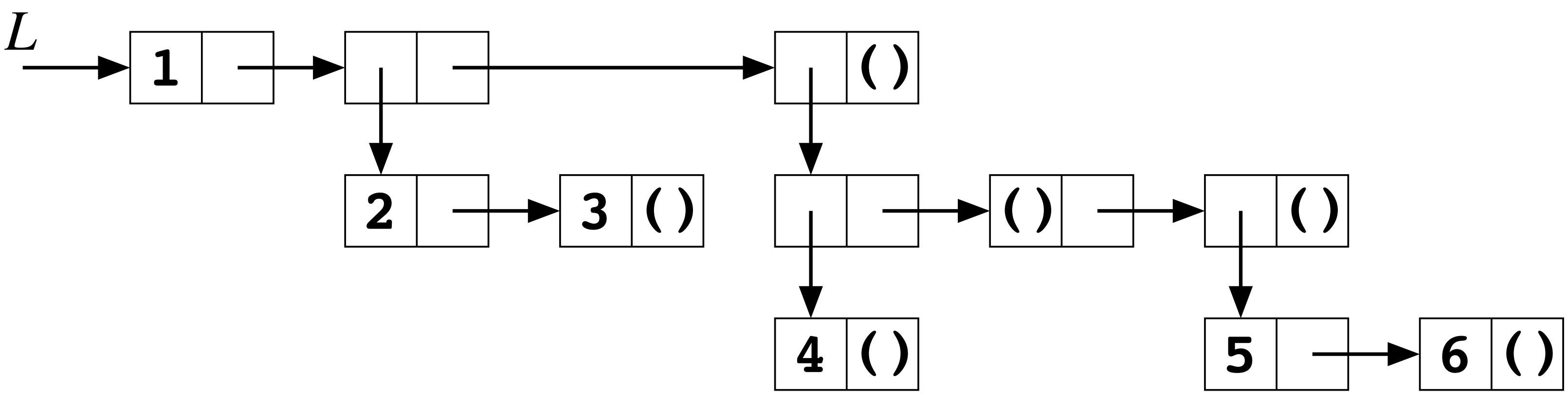



```
> (car (cdr L))
(2 3)
> (car (car (cdr L)))
2
```

- *Exercise:* How do you fetch **3**, **4**, **5**, and **6**?

Shorthand

- $L =$ `(1 (2 3) ((4) () (5 6)))`



- Shorthand for `(car (cdr` L`))`

```
> (cadr L)
(2 3)
```

- Shorthand for `(car (car (cdr` L`)))`

```
> (caadr L)
2
```

Recursion with Lists

- **`(list-sum`** L**`)`**

  Argument(s):

  L: a list of numbers

  Return: the sum of all numbers in L

- *Example:*

```
> (list-sum '(2 -1 4))
5
> (list-sum '())
0
```

Recursion with Lists

```
(define list-sum
  (lambda (L)
    (if (null? L)
      0
      (+ (car L) (list-sum (cdr L))))))
```

Recursion with Lists

```
  (list-sum '(2 -1 4))
= (if (null? '(2 -1 4))
      0
      (+ (car '(2 -1 4)) (list-sum (cdr '(2 -1 4)))))
= (+ (car '(2 -1 4)) (list-sum (cdr '(2 -1 4))))
= (+ 2 (list-sum (cdr '(2 -1 4))))
= (+ 2 (list-sum '(-1 4)))
= (+ 2 (if (null? '(-1 4))
           0
           (+ (car '(-1 4)) (list-sum (cdr '(-1 4))))))
= (+ 2 (+ (car '(-1 4)) (list-sum (cdr '(-1 4)))))
= (+ 2 (+ -1 (list-sum '(4))))
```

Recursion with Lists

```
= (+ 2 (+ -1 (list-sum '(4))))
= (+ 2 (+ -1 (if (null? '(4))
                 0
                 (+ (car '(4))
                    (list-sum (cdr '(4)))))))
= (+ 2 (+ -1 (+ 4 (list-sum '()))))
= (+ 2 (+ -1 (+ 4 (if (null? '())
                      0
                      (+ (car '())
                         (list-sum (cdr '()))))))))
= (+ 2 (+ -1 (+ 4 0)))
= 5
```

More Built-in Procedures

Procedure	Meaning
(list x_1 x_2 **...** x_n**)**	create a list containing the arguments
(list? x**)**	test if x is a list
(null? x**)**	test if x is the empty list
(pair? x**)**	test if x is a **cons** cell
(member x L**)**	test if x is a member of list L
(length L**)**	the number of members of list L

Even More Built-in Procedures

- **`(reverse`** L**`)`**
  - Returns a list containing exactly the same objects as the members of L, but in reversed order.

```
(reverse '(1 2 3)) ⇒ (3 2 1)
(reverse '()) ⇒ ()
(reverse '((1 2) 3)) ⇒ (3 (1 2))
```

- **`(append`** L_1 L_2**`)`**
  - Returns a list containing both the elements of L_1 and L_2, with those from L_2 following those from L_1

```
(append '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(append '(1 2) '()) ⇒ (1 2)
(append '() '(1 2)) ⇒ (1 2)
(append '((1 2) 3) '((4))) ⇒ ((1 2) 3 (4))
```

Symbols

Symbols

A symbol is just a special name for a value.

The value could be anything, but the symbol is used to refer to the same value every time, and is used for fast comparisons.

They are like numerical constants in C.

- *Examples:*
  - **`hello`**
  - **`if`**
  - **`a3`**
  - **`+`**
  - **`zero?`**
- Symbols are **case insensitive**
  - **`cos`** and **`COS`** are the same symbol

Quoting Revisited . . .

- Expressing symbol constant . . .

  ```
  > hello
  Error!
  ```

- Scheme thought you want to retrieve the value of the global variable **hello**.
- Again, quoting . . .

  ```
  > 'hello
  hello
  > '(hello world)
  (hello world)
  > '(a (b c) ((d)))
  (a (b c) ((d)))
  ```

Symbol-Related Procedures

Procedure	Meaning
`(symbol?` x**`)`**	test if x is a symbol
`(eq?` x y**`)`**	test if x & y denote the same symbol
`(eqv?` x y**`)`**	test if x & y denote the same symbol or the same number