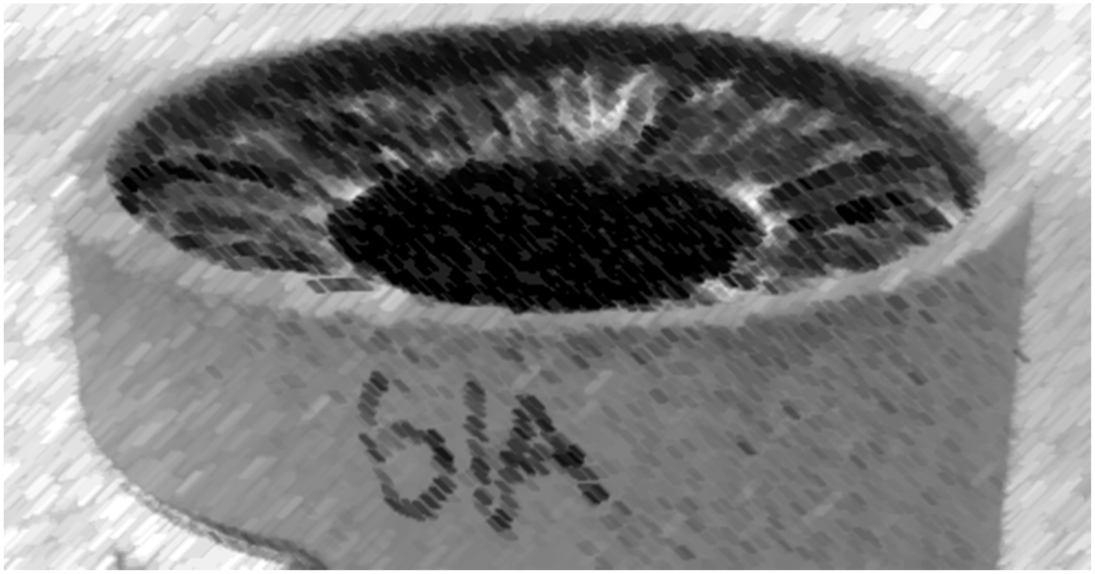


CS61A Lecture 30

Amir Kamil

UC Berkeley

April 1, 2013

Announcements

- HW9 due Wednesday
- Ants extra credit due Wednesday
  - See Piazza for submission instructions
- Hog revisions out, due next Monday

Scheme Is a Dialect of Lisp

Scheme Is a Dialect of Lisp

"The greatest single programming language ever designed."

-Alan Kay, co-inventor of Smalltalk and OOP

Scheme Is a Dialect of Lisp

"The greatest single programming language ever designed."
-Alan Kay, co-inventor of Smalltalk and OOP

"The only computer language that is beautiful."
-Neal Stephenson, sci-fi author

Scheme Is a Dialect of Lisp

"The greatest single programming language ever designed."
 -Alan Kay, co-inventor of Smalltalk and OOP

"The only computer language that is beautiful."
 -Neal Stephenson, sci-fi author

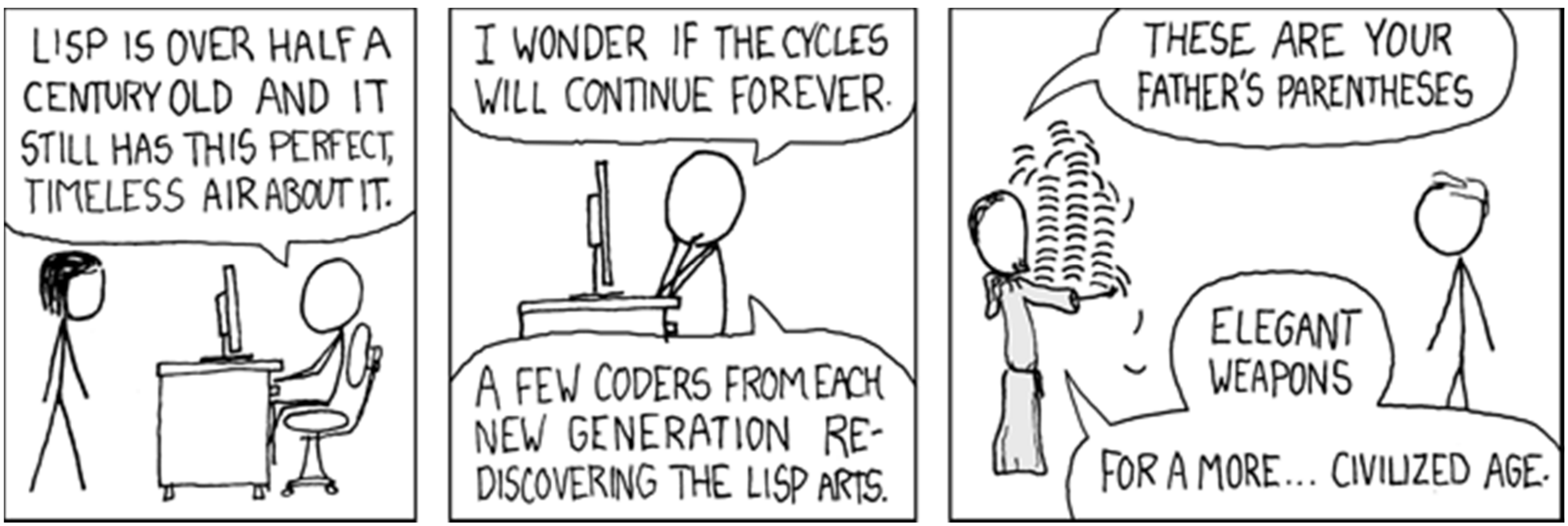
"The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages."
 -Richard Stallman, founder of the Free Software movement

Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
-Alan Kay, co-inventor of Smalltalk and OOP

“The only computer language that is beautiful.”
-Neal Stephenson, sci-fi author

“The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.”
-Richard Stallman, founder of the Free Software movement



http://imgs.xkcd.com/comics/lisp_cycles.png

Scheme Fundamentals

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

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

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines
(spacing doesn't matter)

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines
(spacing doesn't matter)

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines
(spacing doesn't matter)

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines
(spacing doesn't matter)

Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- Primitive expressions: **`2`**, **`3.3`**, **`true`**, **`+`**, **`quotient`**, ...
- Combinations: **`(quotient 10 2)`**, **`(not true)`**, ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)))
     (+ (- 10 7)
        6))
57
```

"quotient" names Scheme's built-in integer division procedure (i.e., function)

Combinations can span multiple lines
(spacing doesn't matter)

Special Forms

Special Forms

A combination that is not a call expression is a *special form*:

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
```

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
```

The name “pi” is bound to 3.14 in the global frame

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
> (* pi 2)
```

The name “pi” is bound to 3.14 in the global frame

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name "pi" is bound to 3.14 in the global frame

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Binding names: `(define <name> <expression>)`
- New procedures: `(define (<name> <formal parameters>) <body>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
```

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Binding names: `(define <name> <expression>)`
- New procedures: `(define (<name> <formal parameters>) <body>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
```

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name "pi" is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
```

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: (if <predicate> <consequent> <alternative>)
- **And** and **or**: (and <e_1> ... <e_n>), (or <e_1> ... <e_n>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name "pi" is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
```

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Binding names: `(define <name> <expression>)`
- New procedures: `(define (<name> <formal parameters>) <body>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
```

A procedure is created and bound to the name “abs”

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Binding names: `(define <name> <expression>)`
- New procedures: `(define (<name> <formal parameters>) <body>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
```

A procedure is created and bound to the name “abs”

Special Forms

A combination that is not a call expression is a *special form*:

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Binding names: `(define <name> <expression>)`
- New procedures: `(define (<name> <formal parameters>) <body>)`

```
> (define pi 3.14)
> (* pi 2)
6.28
```

The name “pi” is bound to 3.14 in the global frame

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

A procedure is created and bound to the name “abs”

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a combination too:

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a combination too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a combination too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

Evaluates to the *add-x-&-y-&-z*^{2} procedure

Pairs

Pairs

We can implement pairs functionally:

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also has built-in pairs that use weird names:

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also has built-in pairs that use weird names:

- **`cons`**: Two-argument procedure that **creates a pair**

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
(1 . 2)
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
```

Pairs

We can implement pairs functionally:

```
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also 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, all in parens

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

Recursive Lists

Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list

Recursive Lists

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:

Recursive Lists

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

Recursive Lists

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

Recursive Lists

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

Recursive Lists

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

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

Recursive Lists

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

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

Recursive Lists

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

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

Recursive Lists

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

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

Recursive Lists

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

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

Recursive Lists

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

```
> (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)))
```

Recursive Lists

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

```
> (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)
```

Recursive Lists

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

```
> (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

Symbolic Programming

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

Symbolic Programming

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

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

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)
```

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)
```

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)
```

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)
```

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

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

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?

```
> (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
```

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))
a
> (cdr '(a b c))
```

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))
a
> (cdr '(a b c))
(b c)
```

Scheme Lists and Quotation

Scheme Lists and Quotation

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

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)))
```

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
```

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
```

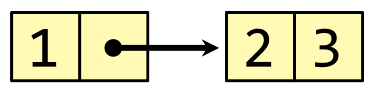
However, dots appear in the output only of ill-formed lists

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



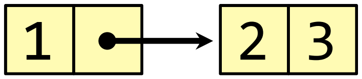
Scheme Lists and Quotation

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)
```



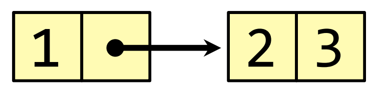
Scheme Lists and Quotation

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)
```



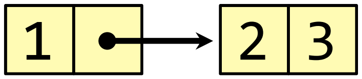
Scheme Lists and Quotation

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))
```



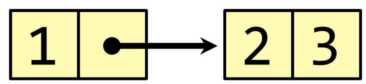
Scheme Lists and Quotation

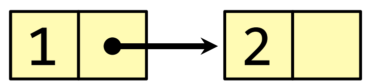
Dots can be used in a quoted list to specify the second element of the final pair

```
> (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))
```





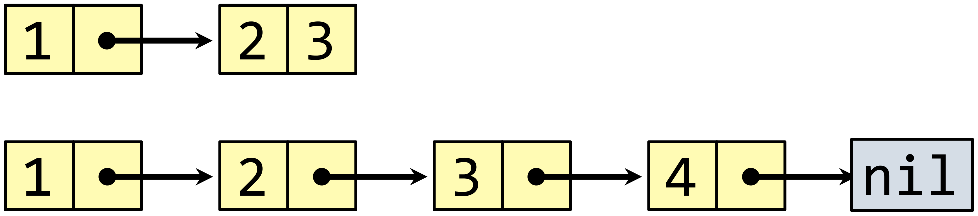
Scheme Lists and Quotation

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))
```



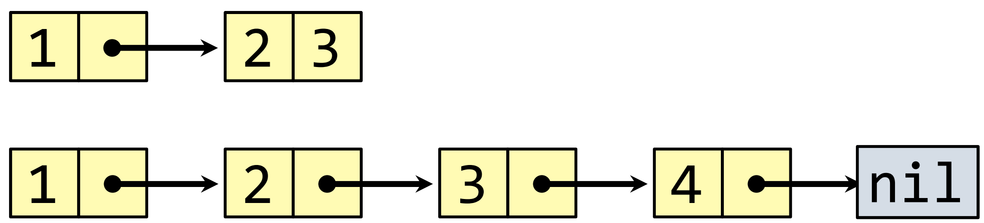
Scheme Lists and Quotation

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)
```



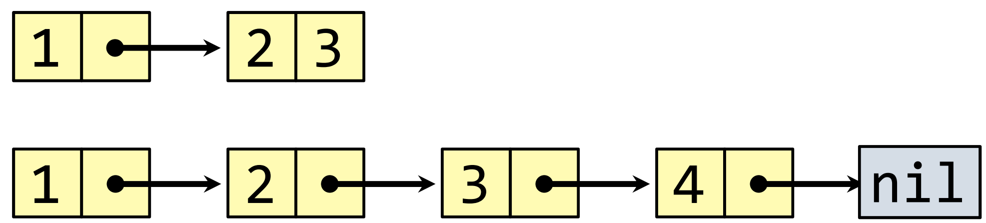
Scheme Lists and Quotation

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)
```



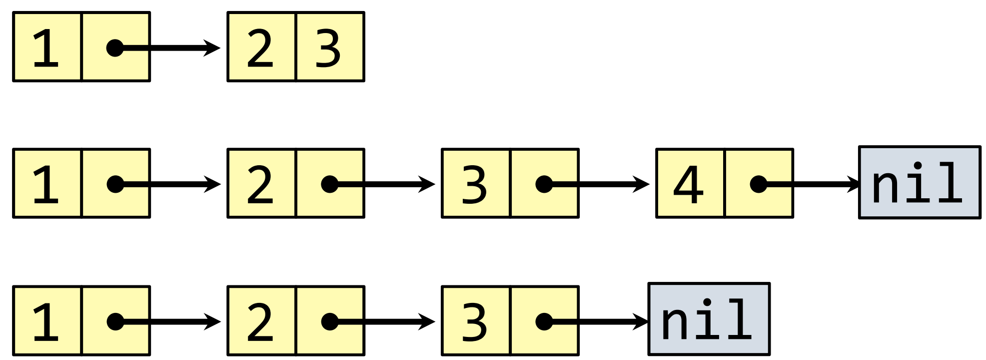
Scheme Lists and Quotation

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)
```



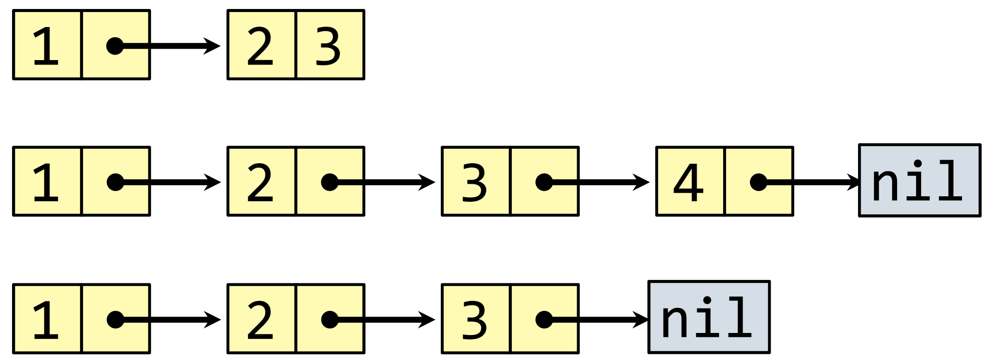
Scheme Lists and Quotation

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)
```



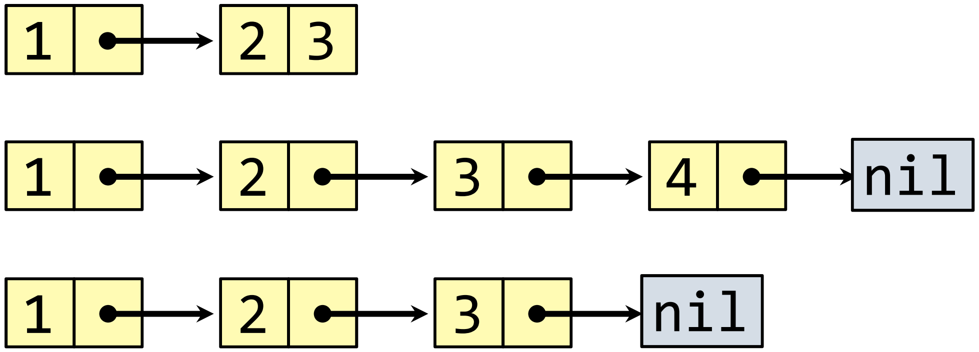
Scheme Lists and Quotation

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)
```



What is the printed result of evaluating this expression?

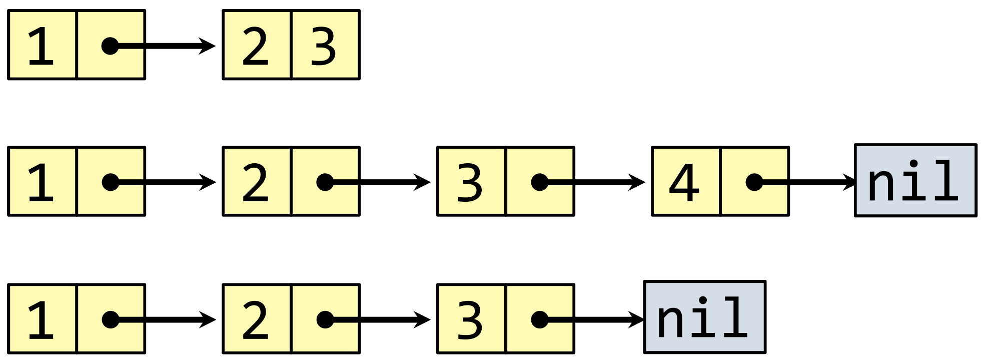
Scheme Lists and Quotation

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)
```



What is the printed result of evaluating this expression?

```
> (cdr '((1 2) . (3 4 . (5))))
```

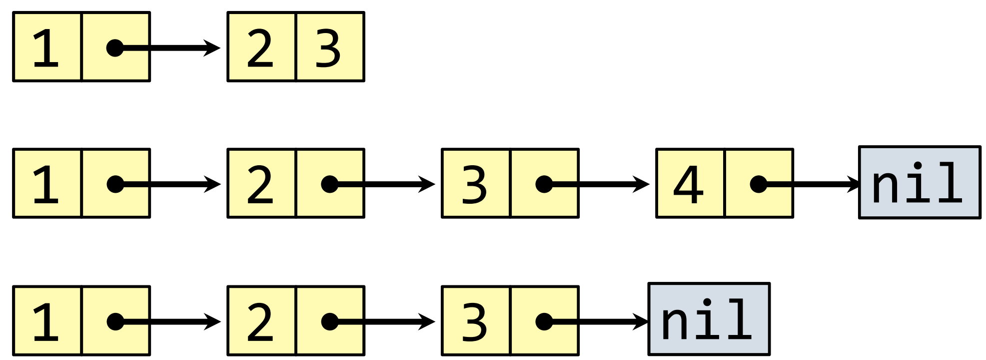
Scheme Lists and Quotation

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)
```



What is the printed result of evaluating this expression?

```
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```