Scheme, More

Slides adapted from Berkeley cs61a

Pairs and Lists

Pairs

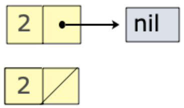
cons: construct
car and cdr: historical reason (Lisp on IBM 704)

- Pairs are created using the **cons** expression in Scheme
- **car** selects the first element in a pair
- **cdr** selects the second element in a pair
- The second element of a pair must be another pair, or **nil (empty)**

```
scm> (define x (cons 1 (cons 3 nil))
x
scm> x
(1 3)
scm> (car x)
1
scm> (cdr x)
(3)
```

Lists

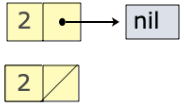
(cons 2 nil)



- The only type of sequence in Scheme is the linked list
  - We can create these with pairs using multiple **cons** expressions
- nil represents the empty list

Lists

(cons 2 nil)

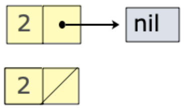


- The only type of sequence in Scheme is the linked list
  - We can create these with pairs using multiple **cons** expressions
- nil represents the empty list

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

>(define x (cons 1 (cons 2 nil))
>x

>(car x)

>(cdr x)

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

Lists

(cons 2 nil)



- The only type of sequence in Scheme is the linked list
  - We can create these with pairs using multiple **cons** expressions
- nil represents the empty list

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

Demo_1

Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

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

Symbolic Programming

Symbols normally refer to 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 normally refer to 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 is used to refer to symbols directly in Lisp.

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

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

Symbolic Programming

Symbols normally refer to 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 is used to refer to symbols directly in Lisp.

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

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

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

Symbolic Programming

Symbols normally refer to 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 is used to refer to symbols directly in Lisp.

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

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

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

Demo_2

Tail Recursion

Recursion Versus Iteration in Python

```
def rfactorial(n):
    if n == 0:
        return 1
    else:
        return n * rfactorial(n - 1)
def ifactorial(n):
    total = 1
    while n > 0:
        total *= n
        n -= 1
    return total
```

Multiplication Operations?	Frames?
n	n
n	1

Demo_3

Tail Recursion

- We say an expression is in a **tail context** if it is evaluated as the last step in the function call
    - That means nothing is evaluated/applied after it is evaluated
- Function calls in a tail context are called **tail calls**
- If the tail call calls the function itself, we say that function is **tail recursive**
    - If a language supports tail call optimization, a tail recursive function will only ever open a constant number of frames

Identifying Tail Contexts

An expression is in a tail context only if **it is the last thing evaluated** in every possible scenario (no other action is performed afterwards)

For each of the following expressions, which expressions (expr1, expr2, expr3) are in a tail context?

```
(and expr1 expr2 expr3)
```

```
(+ expr1 expr2)
```

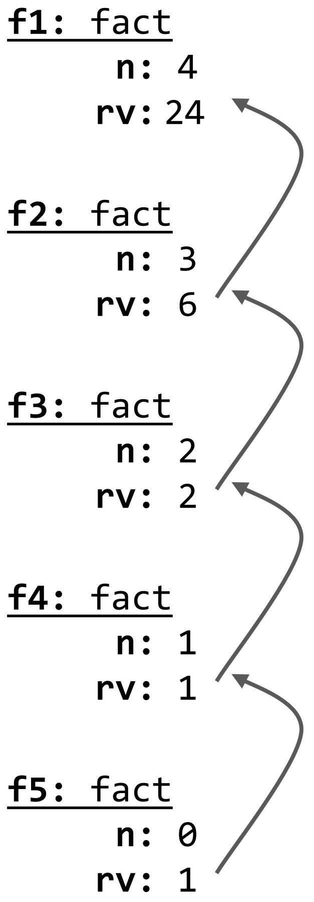
```
(if expr1 expr2 expr3)
```

```
((lambda (expr1) expr1) expr2)
```

Recursive frames

```
(define (fact n)
  (if (= n 0)
   1
   (* n (fact (- n 1)))))
```

Consider a call to (fact 4)



We need to keep these frames open because the last step in the function is to multiply n with the result of the recursive call.

Demo_4

Tail calls

```
(define (fact n)
  (define (fact-tail n result)
    (if (<= n 1)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))

(fact 4)
```

Number of frames the same regardless of input size!

ail

er (fact-tail (- 4 1)
(* 4 1)) returns

ail

er (fact-tail 2 12)

ail

er (fact-tail 1 24)

f4: fact-tail
n: 1
result: 24
rv: 24

Writing Tail Recursive Functions

1) Identify recursive calls that are not in a tail context. Tail contexts are:
   - The last body subexpression in a *lambda* (a function)
   - The consequent and alternative in a tail context *if*
   - The last sub-expression in a tail context *and, or, begin,* or *let*
2) Create a helper function with arguments to accumulate the computation that prevents it from being tail recursive

Example: Length of Linked List

Goal: Write a function that takes in a list and returns the length of the list. Make sure it is tail recursive.

```
(define (length lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst)))))

scm> (length '())
0
scm> (length '(1 2 (3 4))
3
```

```
(define (length-tail lst)



)
```

Demo_5