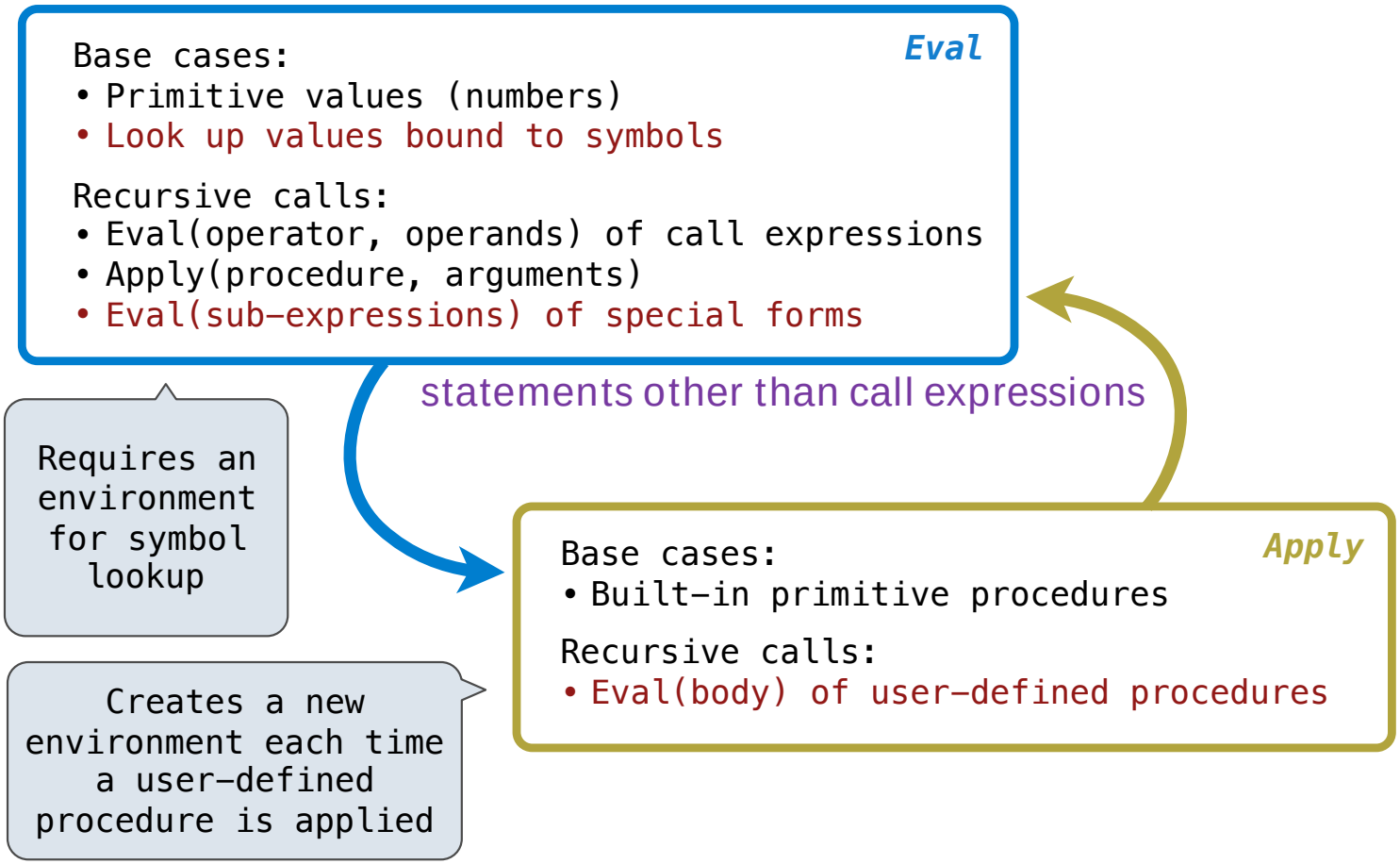
61A Lecture 27

Announcements

Interpreting Scheme

The Structure of an Interpreter

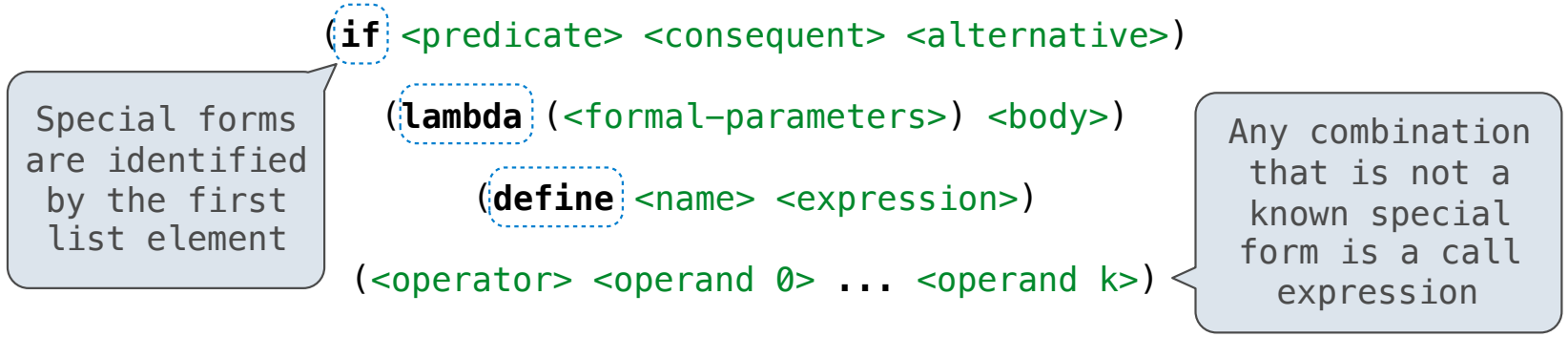


Special Forms

Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations




```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))

(demo (list 1 2))
```

```
scm> 2
scheme_eval(2, <Global Frame>):
scheme_eval(2, <Global Frame>) -> 2
2
scm> (- 2)
scheme_eval(<(- 2)>, <Global Frame>):
    scheme_eval('-', <Global Frame>):
    scheme_eval('-', <Global Frame>) -> #[-]
    scheme_eval(2, <Global Frame>):
    scheme_eval(2, <Global Frame>) -> 2
scheme_eval(<(- 2)>, <Global Frame>) -> -2
-2
```

```
scm> (define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
scheme_eval(<(define (demo s) (if (null? s) (quote (3)) (cons (car s) (demo (cdr s)))))>, <Global Frame>):
scheme_eval(<(define (demo s) (if (null? s) (quote (3)) (cons (car s) (demo (cdr s)))))>, <Global Frame>) -> demo
demo
```

this just defines a function

Python

```
scm> (demo (list 1 2))
scheme_eval(<(demo (list 1 2))>, <Global Frame>):
    scheme_eval('demo', <Global Frame>):
    scheme_eval('demo', <Global Frame>) -> (lambda (s) (if (null? s) (quote (3)) (cons (car s) (demo (cdr s)))))
    scheme_eval(<(list 1 2)>, <Global Frame>):
        scheme_eval('list', <Global Frame>):
        scheme_eval('list', <Global Frame>) -> #[list]
        scheme_eval(1, <Global Frame>):
        scheme_eval(1, <Global Frame>) -> 1
        scheme_eval(2, <Global Frame>):
        scheme_eval(2, <Global Frame>) -> 2
    scheme_eval(<(list 1 2)>, <Global Frame>) -> (1 2)
    scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: (1 2)} -> <Global Frame>>):
        scheme_eval(<(null? s)>, <{s: (1 2)} -> <Global Frame>>):
            scheme_eval('null?', <{s: (1 2)} -> <Global Frame>>):
            scheme_eval('null?', <{s: (1 2)} -> <Global Frame>>) -> #[null?]
            scheme_eval('s', <{s: (1 2)} -> <Global Frame>>):
            scheme_eval('s', <{s: (1 2)} -> <Global Frame>>) -> (1 2)
        scheme_eval(<(null? s)>, <{s: (1 2)} -> <Global Frame>>) -> False
        scheme_eval(<(cons (car s) (demo (cdr s)))>, <{s: (1 2)} -> <Global Frame>>):
            scheme_eval('cons', <{s: (1 2)} -> <Global Frame>>):
            scheme_eval('cons', <{s: (1 2)} -> <Global Frame>>) -> #[cons]
            scheme_eval(<(car s)>, <{s: (1 2)} -> <Global Frame>>):
                scheme_eval('car', <{s: (1 2)} -> <Global Frame>>):
                scheme_eval('car', <{s: (1 2)} -> <Global Frame>>) -> #[car]
                scheme_eval('s', <{s: (1 2)} -> <Global Frame>>):
                scheme_eval('s', <{s: (1 2)} -> <Global Frame>>) -> (1 2)
            scheme_eval(<(car s)>, <{s: (1 2)} -> <Global Frame>>) -> 1
            scheme_eval(<(demo (cdr s))>, <{s: (1 2)} -> <Global Frame>>):
```

find out it's list

now we have a func, an arg, and an environment

a frame where s=(1, 2), followed by global

```
scheme_eval(<(cdr s)>, <{s: (1 2)} -> <Global Frame>>):
    scheme_eval('cdr', <{s: (1 2)} -> <Global Frame>>):
    scheme_eval('cdr', <{s: (1 2)} -> <Global Frame>>) -> #[cdr]
    scheme_eval('s', <{s: (1 2)} -> <Global Frame>>):
    scheme_eval('s', <{s: (1 2)} -> <Global Frame>>) -> (1 2)
scheme_eval(<(cdr s)>, <{s: (1 2)} -> <Global Frame>>) -> (2)
scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: (2)} -> <Global Frame>>):
    scheme_eval(<(null? s)>, <{s: (2)} -> <Global Frame>>):
        scheme_eval('null?', <{s: (2)} -> <Global Frame>>):
        scheme_eval('null?', <{s: (2)} -> <Global Frame>>) -> #[null?]
```

the environment changes

```
                scheme_eval(<(cdr s)>, <{s: (2)} -> <Global Frame>>):
                    scheme_eval('cdr', <{s: (2)} -> <Global Frame>>):
                    scheme_eval('cdr', <{s: (2)} -> <Global Frame>>) -> #[cdr]
                    scheme_eval('s', <{s: (2)} -> <Global Frame>>):
                    scheme_eval('s', <{s: (2)} -> <Global Frame>>) -> (2)
                scheme_eval(<(cdr s)>, <{s: (2)} -> <Global Frame>>) -> ()
                scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: ()} -> <Global Frame>>):
                    scheme_eval(<(null? s)>, <{s: ()} -> <Global Frame>>):
                        scheme_eval('null?', <{s: ()} -> <Global Frame>>):
                        scheme_eval('null?', <{s: ()} -> <Global Frame>>) -> #[null?]
                        scheme_eval('s', <{s: ()} -> <Global Frame>>):
                        scheme_eval('s', <{s: ()} -> <Global Frame>>) -> ()
                    scheme_eval(<(null? s)>, <{s: ()} -> <Global Frame>>) -> True
                    scheme_eval(<(quote (3))>, <{s: ()} -> <Global Frame>>):
                    scheme_eval(<(quote (3))>, <{s: ()} -> <Global Frame>>) -> (3)
                scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: ()} -> <Global Frame>>) -
> (3)
            scheme_eval(<(demo (cdr s))>, <{s: (2)} -> <Global Frame>>) -> (3)
        scheme_eval(<(cons (car s) (demo (cdr s)))>, <{s: (2)} -> <Global Frame>>) -> (2 3)
    scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: (2)} -> <Global Frame>>) -> (2 3)
scheme_eval(<
scheme_eval(<cons
scheme_eval(<(if (null? s) (quote (3)) (cons (car s) (demo (cdr s))))>, <{s: (1 2)} -> <Global Frame>>) -> (1 2 3)
scheme_eval(<(demo (list 1 2))>, <Global Frame>) -> (1 2 3)
```

detect the base case

entend a 3

a recursive result

base case we detect the base case here

the final result

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e1> **...** <en>), (**or** <e1> **...** <en>)
- **Cond** expression: (**cond** (<p1> <e1>) **...** (<pn> <en>) (else <e>))

The value of an if expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

do_if_form

(Demo)

```
scm> (if #t 1 2)
scheme_eval(<(if True 1 2)>, <Global Frame>):
    scheme_eval(True, <Global Frame>):
    scheme_eval(True, <Global Frame>) -> True
    scheme_eval(1, <Global Frame>):
    scheme_eval(1, <Global Frame>) -> 1
scheme_eval(<(if True 1 2)>, <Global Frame>) -> 1
1
scm> (if #t 1 (/ 1 0))
scheme_eval(<(if True 1 (/ 1 0))>, <Global Frame>):
    scheme_eval(True, <Global Frame>):
    scheme_eval(True, <Global Frame>) -> True
    scheme_eval(1, <Global Frame>):
    scheme_eval(1, <Global Frame>) -> 1
scheme_eval(<(if True 1 (/ 1 0))>, <Global Frame>) -> 1
1
scm> (if #f 1 (/ 1 0))
scheme_eval(<(if False 1 (/ 1 0))>, <Global Frame>):
    scheme_eval(False, <Global Frame>):
    scheme_eval(False, <Global Frame>) -> False
    scheme_eval(<(/ 1 0)>, <Global Frame>):
        scheme_eval('/', <Global Frame>):
        scheme_eval('/', <Global Frame>) -> #[/]
        scheme_eval(1, <Global Frame>):
        scheme_eval(1, <Global Frame>) -> 1
        scheme_eval(0, <Global Frame>):
        scheme_eval(0, <Global Frame>) -> 0
        scheme_eval exited via exception
    scheme_eval exited via exception
Traceback (most recent call last):
  0     (/ 1 0)
Error: division by zero
```

2 is never evaluated

1/0 is never evaluated,
so there will be no error

1/0 is evaluated, and thus
there is zero division error

Quotation

Quotation

The quote special form evaluates to the quoted expression, which is not evaluated

(quote <expression>) (quote (+ 1 2)) evaluates to the three-element Scheme list (+ 1 2)

The <expression> itself is the value of the whole quote expression

'<expression> is shorthand for (quote <expression>)

(quote (1 2)) is equivalent to '(1 2)

The scheme_read parser converts shorthand ' to a combination that starts with quote

(Demo)

```
read> (quote (1 2))
(quote (1 2))
read> '(1 2)
(quote (1 2))
```

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

```
(lambda (x) (* x x))
```

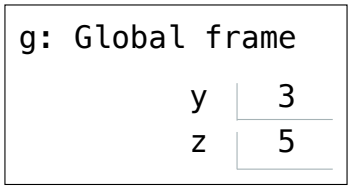
```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals ..... A scheme list of symbols
        self.body = body ..... A scheme list of expressions
        self.env = env ..... A Frame instance
```

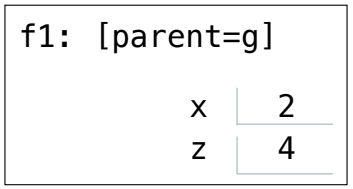
Frames and Environments

A frame represents an environment by having a parent frame

Frames are Python instances with methods **lookup** and **define**

In Project 4, Frames do not hold return values





(Demo)

```
>>> g = Frame(None)
>>> g
<Global Frame>
>>> f1 = Frame(g)
>>> f1
<{} -> <Global Frame>>
>>> g.define('y', 3)
>>> g.define('z', 5)
>>> g.lookup('y')
3
>>> g.lookup('z')
5
>>> f1.define('x', 2)
>>> f1.define('z', 4)
>>> f1
<{x: 2, z: 4} -> <Global Frame>>
>>> f1.lookup('x')
2
>>> f1.lookup('z')
4
>>> f1.lookup('y')
3
```

Define Expressions

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>
2. Bind <name> to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression

```
(define (<name> <formal parameters>) <body>)
```

```
(define <name> (lambda (<formal parameters>) <body>))
```

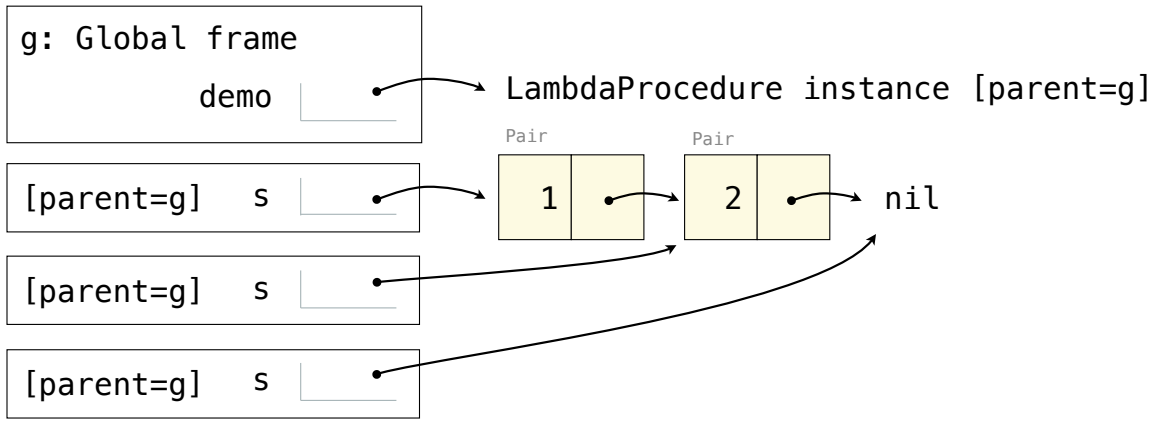
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** attribute of the procedure

Evaluate the body of the procedure in the environment that starts with this new frame

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

(demo (list 1 2))
```



Eval/Apply in Lisp 1.5

```
apply[fn;x;a] =
      [atom[fn] → [eq[fn;CAR] → caar[x];
                   eq[fn;CDR] → cdar[x];
                   eq[fn;CONS] → cons[car[x];cadr[x]];
                   eq[fn;ATOM] → atom[car[x]];
                   eq[fn;EQ] → eq[car[x];cadr[x]];
                   T → apply[eval[fn;a];x;a]];
       eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
       eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                     caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
      atom[car[e]] →
                  [eq[car[e],QUOTE] → cadr[e];
                  eq[car[e];COND] → evcon[cdr[e];a];
                  T → apply[car[e];evlis[cdr[e];a];a]];
      T → apply[car[e];evlis[cdr[e];a];a]]
```