Scheme

Slides adapted from Berkeley cs61a

Scheme is a Dialect of Lisp

What are people saying about Lisp?

- "If you don't know Lisp, you don't know what it means for a programming language to be powerful and elegant."

  - Richard Stallman, created Emacs & the first free variant of UNIX

- "The only computer language that is beautiful."

  -Neal Stephenson, DeNero's favorite sci-fi author

- "The greatest single programming language ever designed."

  -Alan Kay, co-inventor of Smalltalk and OOP

Scheme expressions

Scheme programs consist entirely of two types of expressions.

Atomic expressions *(Atoms: primitive values that cannot be broken up into smaller parts)*

- *Self-evaluating:* numbers, booleans
  `3`, `5.5`, `-10`, `#t`, `#f`
- *Symbols:* names bound to values
  `+`, `modulo`, `list`, `x`, `foo`, `hello-world`

Combinations

`(<operator> <operand1> <operand2> ...)`

A combination is either a **call expression** or a **special form expression**.

Call Expressions

Call expressions

(<operator> <operand1> <operand2> ...)

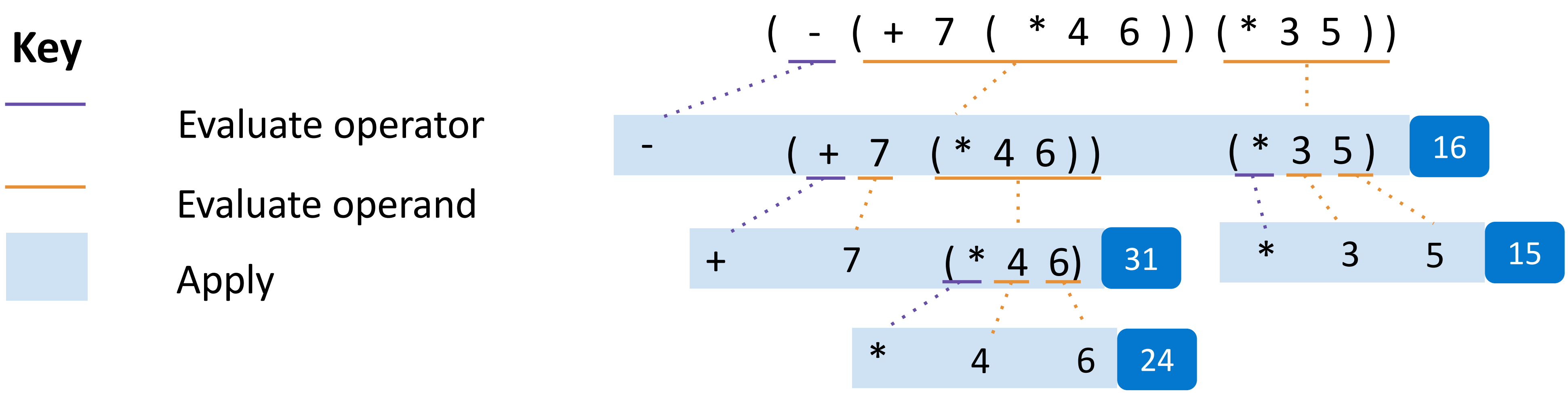
A call expression applies a procedure to some arguments.

How to evaluate call expressions:

Step 1. Evaluate the operator to get a procedure.

Step 2. Evaluate all operands left to right to get the arguments.

Step 3. Apply the procedure to the arguments.



Call Expressions

Call expressions include an operator and 0 or more operands in parentheses

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

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

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

(Demo_1)

Special Form Expressions

(<operator> <operand1> <operand2> …)

<operator> : define, if, lambda, etc.

Assigning values to names

The define special form assigns a value to a name:

`(define <name> <expr>)`

How to evaluate:
Step 1. Evaluate the given expression.
Step 2. Bind the resulting value to the given name in the current frame.
Step 3. Return the name as a symbol.

```
scm> (define x (+ 3 4))
x
scm> x
7
scm> (define x (+ x 5))
x
scm> x
12
```

Control flow

The if special form allows us to evaluate an expression based on a condition:

`(if <predicate> <if-true> <if-false>)`

#f is the only Falsy value in Scheme

How to evaluate:
Step 1. Evaluate the `<predicate>`.
Step 2. If `<predicate>` evaluates to anything but `#f`, evaluate `<if-true>` and return the value. Otherwise, evaluate `<if-false>` if provided and return the value.

```
scm> (if #t 3 5)
3
scm> (if 0 (+ 1 0) (/ 1 0))
1
scm> (if (> 10 1) (* 5 6))
30
scm> (if (not 4) 1 (if #f 5 6))
6
```

Defining functions with names

The second version of define is a shorthand for creating a function with a name:

`(define (<name> <param1> <param2> ...) <body>)`

How to evaluate:

Step 1. Create a lambda procedure with the given parameters and body.
Step 2. Bind it to the given name in the current frame.
Step 3. Return the function name as a symbol.

```
scm> (define (square x) (* x x))
square
scm> square
(lambda (x) (* x x))
scm> (square 4)
16
scm> (square -10)
100
```

(Demo_2)

Lambda Expressions

The lambda special form returns a lambda procedure.

`(lambda (<param1> <param2> …) <body>)`

How to evaluate:
Step 1. Create a lambda procedure with the given parameters and body.
Step 2. Return the lambda procedure.

The body expression is evaluated when the lambda procedure is applied.

```
scm> (lambda (x) (* x x))
(lambda (x) (* x x))
scm> ((lambda (x) (* x x)) 5)
25
scm> (define square (lambda (x) (* x x)))
square
scm> (square 4)
16
```

Lambda Expressions

Two equivalent expressions:

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

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

Lambda Expressions

Two equivalent expressions:

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

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

An operator can be a call expression too:

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

Evaluates to the $x+y+z^2$ procedure

Lambda Expressions

Two equivalent expressions:

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

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

An operator can be a call expression too:

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

Evaluates to the $x+y+z^2$ procedure

黑板时间

Check Your Understanding

What would Scheme display for the following expressions?

```
scm> (define x 5)
 x
scm> (lambda (x y) (print 2))
 (lambda (x y) (print 2))
scm> ((lambda (x) (print x)) 1)
 1
scm> (define f (lambda () #f))
 f
scm> (if f x (+ x 1))
 5
scm> (if (f) (print 5) 6)
 6
scm> (+ (if 1 2 3) (if 4 5 6))
 7
```

`(define <name> <expr>)`

Step 1. Evaluate the given expression.
Step 2. Bind the value to the given name.
Step 3. Return the name as a symbol.

`(lambda (<p1> <p2> …) <body>)`

Step 1. Create a procedure with the given parameters and body.
Step 2. Return the procedure.

`(if <pred> <if-true> <if-false>)`

Step 1. Evaluate the predicate.
Step 2. If the predicate isn't `#f`, evaluate `<if-true>` and return the value. Otherwise, evaluate `<if-false>` and return the value.

Example: Factorial

Recall the factorial function, which takes in an integer `n` and computes the product of all the integers from 1 to `n`.

Let's try to write it in Scheme!

Scheme has no special form that allows for iteration, so we have to use recursion.

Ideas:

1. *Base case:* if n is 0 or 1, just return 1
2. *Recursive case:* Return the factorial of the previous number multiplied by n
3. Use the if special form to capture our two cases:

```
(if <pred> <if-true> <if-false>)
```

Try it out!

Combinations can be split across multiple lines

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

No explicit return statement!

Example: Counting up

Let's write a function count-up that takes in an integer n and prints all the integers from 1 to n.

Ideas:

1. We need to keep track of the current element, k. k starts at 1.
2. Since we have to use recursion, we can write a helper function to keep track of k.
3. Print k at the beginning of every call and only make a recursive call to print more numbers if k is less than n.

```
(define (count-up n)
    (define (counter k)
        (print k)
        (if (< k n)
            (counter (+ k 1))))
    (counter 1))
```

If there is more than one expression in the body, the function returns the value of the *last* expression.

The X You Need To Understand In This Lecture

- Scheme programs consist only of expressions, all of which can be categorized into either **atomic expressions** or **combinations**.
- Combinations are either **call expressions** or **special form expressions**, and they differ in the value of the operator.
- Scheme call expressions are evaluated just like they are in Python, but each special form has its own rules of evaluation.
- The special forms we learned today are `if`, `define`, and `lambda`.
- Writing some procedures in Scheme will require recursion; there is no special form for iteration.