Programming Paradigms Functional Languages

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart Summer 2022

Overview

Introduction

- A Bit of Scheme
- Evaluation Order

What does the following Scheme code evaluate to?

What does the following Scheme code evaluate to?

Result: 6

Wake-up Quiz

What does the following Scheme code evaluate to?

(let ((foo 4)) let binds names
 (let ((foo 2) to values
 (bar foo))
 (+ foo bar)))

Result: 6

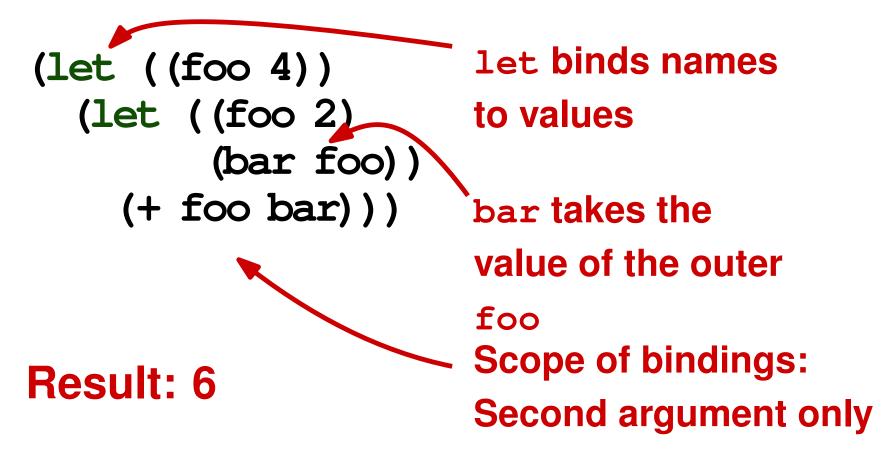
Wake-up Quiz

What does the following Scheme code evaluate to?

let binds names (let ((foo 4)) (let ((foo 2) to values (bar foo)) (+ foo bar))) Scope of bindings: **Result: 6** Second argument only

Wake-up Quiz

What does the following Scheme code evaluate to?



Functional Languages

- Functional paradigm: Alternative to imperative PLs
 - Output: Mathematical function of input
 - □ No internal state, no side effects
- In practice: Fuzzy boundaries
 - □ "Functional" features in many "imperative" PLs
 - E.g., higher-order functions
 - □ "Imperative features" in many "functional" PLs
 - E.g., assignment and iteration

Historical Origins

Lambda calculus

Alonzo Church, 1930s

Express computation based on

Abstraction into functions

- E.g., $(\lambda x.M)$
- Function application
 - E.g., (*M N*)

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators
- Structured function returns
- Constructors for structured objects
- Garbage collection

- First-class function values and higher-order function Funct
- Extensive polymorphism
- List types and operators
- Functions assigned to variables, passed as arguments, or return values
- Structured function returns
- Constructors for structured objects
- Garbage collection

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators

Use a function on different kinds of values, e.g., using type inference

- Structured function returns
- Constructors for structured objects
- Garbage collection

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators

Ideal for recursion (handle first element and then recursively the remainder)

- Structured function returns
- Constructors for structured objects
- Garbage collection

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators

Functions can return any structured data, e.g., lists and functions

- Structured function returns
- Constructors for structured objects
- Garbage collection

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators
- Structured function returns
- Constructors for structured objects
- Garbage collection

Construct aggregate objects inline and all-at-once

- First-class function values and higher-order function
- Extensive polymorphism
- List types and operators
- Structured function returns
- Constructors for structured objects
- Garbage collection

Necessary because evaluation tends to create lots of temporary data

Purely Functional PLs

Functions depend only on their parameters

- □ Not on any other global or local state
- Order of evaluation is irrelevant
 - Eager and lazy evaluation yield same result

E.g., Haskell

- □ By Philip Wadler et al., first released in 1990
- □ Actively used as a research language

Non-Pure Functional PLs

- Mix of functional features with assignments
- E.g., Scheme
 - Dialect of Lisp
 - □ By Guy Steele and Gerlad Jay Sussman (MIT)

E.g., OCaml

- Extends ML with OO features
- Developed at INRIA (France)

Overview

- Introduction
- A Bit of Scheme
- Evaluation Order

Function Application

Pair of parentheses: Function application

□ First expression inside: Function

□ Remaining expressions: Arguments

Examples:

(+ 3 4) ((+ 3 4))

Function Application

Pair of parentheses: Function application

□ First expression inside: Function

□ Remaining expressions: Arguments

Examples:

(+ 3 4) ((+ 3 4))

Applies + function to 3 and 4. Evaluates to 7.

Function Application

Pair of parentheses: Function application

□ First expression inside: Function

Remaining expressions: Arguments

Examples:

(+ 3 4)

Applies + function to 3 and 4. Evaluates to 7. ((+ 3 4))

Tries to call 7 with zero arguments. Gives runtime error. 10-3

Creating Functions

Evaluating a lambda expression yields a function

□ First argument to lambda: Formal parameters

□ Remaining arguments: Body of the function

Example:

(lambda (x) (* x x))

Creating Functions

Evaluating a lambda expression yields a function

First argument to lambda: Formal parameters

□ Remaining arguments: Body of the function

Example:

(lambda (x) (* x x))

Yields the "square" function

Names bound to values with let

□ First argument: List of name-value pairs

 Second argument: Expressions to be evaluated in order

Example:

Names bound to values with let

□ First argument: List of name-value pairs

 Second argument: Expressions to be evaluated in order

Example:

Conditional Expressions

Simple conditional expression with if

- □ First argument: Condition
- Second/third argument: Value returned if condition is true/false
- Multiway conditional expression with cond
- Examples:

(if (< 2 3) 4 5)

(cond ((< 3 2) 1) ((< 4 3) 2) (else 3))

Conditional Expressions

Simple conditional expression with if

- □ First argument: Condition
- Second/third argument: Value returned if condition is true/false
- Multiway conditional expression with cond
- Examples:

(if (< 2 3) 4 5)

Yields 4

(cond ((< 3 2) 1) ((< 4 3) 2) (else 3))

13 - 2

Conditional Expressions

Simple conditional expression with if

- □ First argument: Condition
- Second/third argument: Value returned if condition is true/false
- Multiway conditional expression with cond
- Examples:

(if (< 2 3) 4 5)

Yields 4

Yields 3 (cond ((< 3 2) 1) ((< 4 3) 2) (else 3))

13 - 3

Dynamic Typing

- Types are determined and checked at runtime
- Examples:

(if (> a 0) (+ 2 3) (+ 2 "foo"))

(define min (lambda (a b) (if (< a b) a b)))

Dynamic Typing

- Types are determined and checked at runtime
- Examples:

(if (> a 0) (+ 2 3) (+ 2 "foo"))
 Evaluates to 5 if a is positive;
 runtime type error otherwise

(define min (lambda (a b) (if (< a b) a b)))

Dynamic Typing

- Types are determined and checked at runtime
- Examples:

(if (> a 0) (+ 2 3) (+ 2 "foo"))

Evaluates to 5 if a is positive; runtime type error otherwise

(define min (lambda (a b) (if (< a b) a b))) Implicitly polymorphic: Works both for integers and floats.

Quiz: Functions in Scheme

Which of the following yields 9?

; Program 1 ((lambda (x) (* x x)) 3)

; Program 2 (- (+ 12 4) (+ 2 4))

; Program 3 (9)

; Program 4 ((lambda (x y) (-x y)) (+ 10 0) (- 4 1))

Quiz: Functions in Scheme

Which of the following yields 9?

; Program 1 ((lambda (x) (* x x)) 3) 🖌

; Program 2 (- (+ 12 4) (+ 2 4)) 💢

; Program 3 (9) 🗙

; Program 4 ((lambda (x y) (-x y)) (+ 10 0) (-4 1)) 🔀



Central data structure with various operations

□ car extracts first element

cdr extracts all elements but first

cons joins a head to the rest of a list

Examples:

(car'(234)) (cdr'(234)) (cons 2'(34))



Central data structure with various operations

□ car extracts first element

cdr extracts all elements but first

cons joins a head to the rest of a list

Examples:

(car ' (2 3 4))

(cdr ' (2 3 4)) (

(cons 2 ' (3 4))

"Quote" to prevent interpreter from evaluating (i.e., a literal)



Central data structure with various operations

car extracts first element

cdr extracts all elements but first

cons joins a head to the rest of a list

prevent interpreter from evaluating (i.e., a literal)

"Quote" to

Examples:

(car ' (2 3 4))

) (cdr ' (2

(cdr ' (2 3 4))

(cons 2 ' (3 4))

Yields 2



Central data structure with various operations

□ car extracts first element

cdr extracts all elements but first

cons joins a head to the rest of a list

(cdr'(234))

Examples:

(car'(2 3 4))

Yields 2Yields (3 4)

"Quote" to prevent interpreter from evaluating (i.e., a literal)

(cons 2 ' (3 4))



Central data structure with various operations

□ car extracts first element

□ cdr extracts all elements but first

 \Box cons joins a head to the rest of a list

Examples:

(car ' (2 3 4))

Yields 2

"Quote" to prevent interpreter from evaluating (i.e., a literal)

(cdr'(234))(cons 2 ' (3 4)) **Yields (234) Yields (3 4)**

Assignments

Side effects via

□ set! for assignment to variables

□ set-car! for assigning head of list

(cons x 1))

□ set-cdr! for assigning tail of list

Example: (let ((x 2) (l ' (a b))) (set! x 3) (set-car! l ' (c d)) (set-cdr! l ' (e))

Assignments

Side effects via

□ set! for assignment to variables

□ set-car! for assigning head of list

□ set-cdr! for assigning tail of list

Example: (let ((x 2)

Yields (3 (c d) e) 17 - 2

$$x : 3 \\ \mathcal{L}: (a b) \\ \mathcal{L}: ((c d) b) \\ \mathcal{L}: ((c d) b) \\ \mathcal{L}: ((c d) e) \\ \mathcal{L}: (c d) e)$$



Cause interpreter to evaluate multiple expressions one after another with begin

Example:

```
(let
   ((n "there"))
  (begin
   (display "hi ")
   (display n)))
```



 Cause interpreter to evaluate multiple expressions one after another with begin

Example:

```
(let
   ((n "there"))
  (begin
   (display "hi ")
   (display n)))
```

Prints "hi there"

Several forms of loops, e.g., with do

Example:

```
((lambda (n)
  (do ((i 0 (+ i 1))
        (a 0 b)
        (b 1 (+ a b)))
        ((= i n) b)
        (display b)
        (display " "))) 5)
```

Several forms of loops, e.g., with do

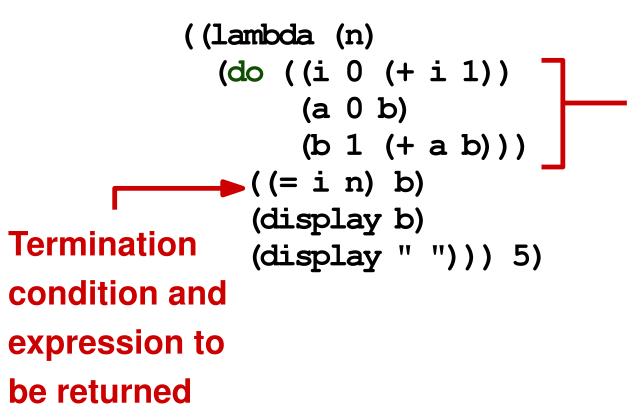
Example:

List of triples that each

- specify a new variable
- its initial value
- expression to compute next value

Several forms of loops, e.g., with do

Example:

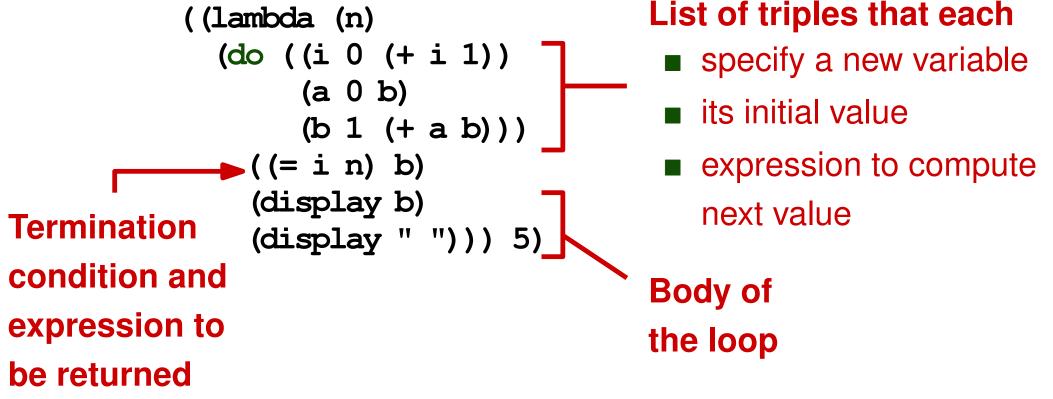


List of triples that each

- specify a new variable
- its initial value
- expression to compute next value

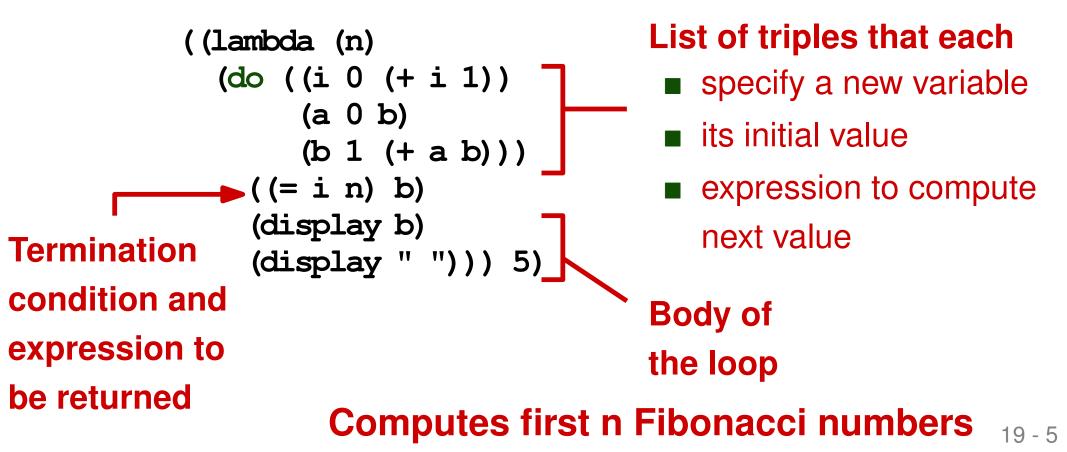
Several forms of loops, e.g., with do

Example:



Several forms of loops, e.g., with do

Example:



Programs as Lists

Programs and lists: Same syntax

- Both are S-expressions: String of symbols with balanced parentheses
- Construct and manipulate an unevaluated program as a list
- Evaluate with eval
- Example:

(eval (cons '+ (list '2 '3)))

Programs as Lists

Programs and lists: Same syntax

- Both are S-expressions: String of symbols with balanced parentheses
- Construct and manipulate an unevaluated program as a list
- Evaluate with eval Constructs a list from
 - **/**the given arguments

Example:

(eval (cons '+ (list '2 '3)))

Programs as Lists

Programs and lists: Same syntax

- Both are S-expressions: String of symbols with balanced parentheses
- Construct and manipulate an unevaluated program as a list
- Evaluate with eval Constructs a list from
 - **/**the given arguments

Example:

(eval (cons '+ (list '2 '3)))

Yields 5

Overview

- Introduction
- A Bit of Scheme
- Evaluation Order

Evaluation Order

In what order to evaluate subcomponents of an expression?

- Applicative-order: Evaluate arguments before passing them to the function
- Normal-order: Pass arguments unevaluated and evaluate once used
- Scheme uses applicative-order

$$\begin{pmatrix} define & double & (lambda & (x) & (+x \times)) \end{pmatrix} \\ \hline Applicative-order \\ (double & (+3 +)) \\ \Rightarrow (double & 12) \\ \Rightarrow (+ 12 + 12) \\ \Rightarrow 24 \end{pmatrix} = (+ (+ 3 +) (+ 3 +)) \\ \Rightarrow (+ 12 + 12) \\ \Rightarrow 24 \\ \hline Doing extra work with normal order ! \\ \hline Doing extra work with normal order ! \\ \hline double & (+ x \times)) \\ \Rightarrow (+ 12 + 12) \\ \Rightarrow 24 \\ \hline Doing extra work with normal order ! \\ \hline Doing extra work with normal work with normal order ! \\ \hline Doing extra work with normal work with norma$$

$$(define switch (lambda (x a b c)
(cond ((< x 0) a)
((= x 0) b)
((> x 0) c))))

Applicative-order
(switch -1 (+12) (+23) (+34))
 \Rightarrow (switch -1 3 (+23) (+34))
 \Rightarrow (switch -1 3 (+23) (+34))
 \Rightarrow (switch -1 3 5 7)
 \Rightarrow (switch -1 3 5 7)
 \Rightarrow (switch -1 3 5 7)
 $((> -1 0) 3)$ =>
 $((> -1 0) 7)$ =>$$

$$\frac{Normal - Stud:}{(switch - 1 (+12) (+23) (+34))}$$

(switch - 1 (+12) (+23))
((2 - 1 0) (+12))
((2 - 1 0) (+23))
((2 - 1 0) (+34))
((2 - 1 0) (+34))
((2 - 1 0) (+34))
((2 - 1 0) (+34))
:
:
:
:
:
:
:
:

=) 3

⇒... ⇒3

Impact on Correctness

- Evaluation order also affects correctness
- E.g., runtime error when evaluating an "unneeded" subexpression
 - Terminates program in applicative-order
 - Not noticed in normal-order

Lazy Evaluation

- Evaluate subexpressions on-demand
- Avoid re-evaluating the same expression
 - Memorize its result
- Transparent to programmer only in PL without side effects, e.g., Haskell
 - □ In PLs with side effects, e.g., Scheme:
 - Programmer can explicitly ask for lazy
 - evaluation with delay

Quiz: Evaluation Order

(define double (lambda (x) (+ x x))) (define avg (lambda (x y) (/(+ x y) 2)))

How many evaluation steps are needed to evaluate

(double(avg 2 4))

under applicative-order and normal-order evaluation?

Quiz: Evaluation Order

(define double (lambda (x) (+ x x))) (define avg (lambda (x y) (/(+ x y) 2)))

How many evaluation steps are needed to evaluate

(double(avg 2 4))

under applicative-order and normal-order evaluation?

5 and 8

Applicative - order (double (avg 24)) $\Rightarrow (double (24) 2))$ => (double (/ 6 2)) =) (double 3) =) (+ 3 3) ⇒ 6 5 steps

$$\frac{Normal-order}{(double (av_j 2 4))}$$
=) (+ (av_j 2 4) (av_j 2 4))
=) (+ (/ (+ 2 4) 2) (av_j 2 4))
=) (+ (/ 6 2) (av_j 2 4))
=) (+ 3 (av_j 2 4))
=) (+ 3 (/ (+ 2 4) 2))
=) (+ 3 (/ (+ 2 4) 2))
=) (+ 3 (/ 6 2))
=) (+ 3 3)
=) 6
& steps

Overview

- Introduction
- A Bit of Scheme
- Evaluation Order

