Programming Paradigms

Functional Languages (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Plan for Today

- 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)
```

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

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

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.

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

Bindings

- **Names bound to values with `let`**
  - First argument: List of name-value pairs
  - Second argument: Expressions to be evaluated in order
- **Example:**

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
    (sqrt (plus (square a) (square b))))
```

Bindings

- **Names bound to values with `let`**
  - First argument: List of name-value pairs
  - Second argument: Expressions to be evaluated in order
- **Example:**

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
    (sqrt (plus (square a) (square b))))
```

Yields 5.0

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))
```

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))
```

Yields 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 3) (+ 2 4))

; Program 3
(9)

; Program 4
((lambda (x y) (- x y)) (+ 10 0) (- 4 2))
```

Please vote via Ilias.

Quiz: Functions in Scheme

Which of the following yields 9?

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

; Program 2
(- (+ 12 3) (+ 2 4)) ✔

; Program 3
(9) ✘

; Program 4
((lambda (x y) (- x y)) (+ 10 0) (- 4 2)) ✘
```

Please vote via Ilias.

Lists

- **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))
```

Lists

- **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:**

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

```
(car '(2 3 4))    (cdr '(2 3 4))    (cons 2 '(3 4))
```

Lists

- **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:**

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

```
(car '(2 3 4))     (cdr '(2 3 4))     (cons 2 '(3 4))
```

Yields 2

Lists

- **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))
```

Yields 2

Yields (3 4)

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

Lists

- **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:**

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

`(car '(2 3 4))` **Yields 2**

`(cdr '(2 3 4))` **Yields (3 4)**

`(cons 2 '(3 4))` **Yields (2 3 4)**

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)
      (l '(a b)))
  (set! x 3)
  (set-car! l '(c d))
  (set-cdr! l '(e))
  (cons x l))
```

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)
      (l '(a b)))
  (set! x 3)
  (set-car! l '(c d))
  (set-cdr! l '(e))
  (cons x l))
```

Yields (3 (c d) e)

Sequencing

- Cause interpreter to evaluate multiple **expressions one after another** with `begin`
- Example:

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

Sequencing

- Cause interpreter to evaluate multiple expressions one after another with `begin`
- Example:

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

Prints "hi there"

Iteration

- 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)
```

Iteration

- 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)
```

List of triples that each

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

Iteration

- 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)
```

List of triples that each

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

Termination condition and expression to be returned

Iteration

- 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)
```

List of triples that each

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

Termination condition and expression to be returned

Body of the loop

Iteration

- 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)
```

List of triples that each

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

Termination condition and expression to be returned

Body of the loop

Computes first n Fibonacci numbers

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`**
- **Example:**

Constructs a list from the given arguments

```
(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`**
- **Example:**

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

Constructs a list from the given arguments

Yields 5