Programming Paradigms

Lecture 19:

Functional Languages

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Wake-up Exercise

What does the following Scheme code evaluate to?

```
(let ((a 3))
 (let ((a 4)
      (b a))
    (+ a b)))
```
Wake-up Exercise

What does the following Scheme code evaluate to?

```scheme
(let ((a 3))
  (let ((a 4)
        (b a))
    (+ a b)))
```

Result: 7

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does the following Scheme code evaluate to?

(let ((a 3))
  (let ((a 4)
        (b a))
    (+ a b)))

Result: 7
Wake-up Exercise

What does the following Scheme code evaluate to?

```
(let ((a 3))
  (let ((a 4)
        (b a))
    (+ a b)))
```

Result: 7

- `let` binds names to values
- Scope of bindings: Second argument only
Wake-up Exercise

What does the following Scheme code evaluate to?

```
(let ((a 3))
  (let ((a 4)
        (b a))
   (+ a b)))
```

Result: 7

- `let` binds names to values
- `b` takes the value of the outer `a`
- Scope of bindings: Second argument only
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., \((M N)\)
Features

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

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

Functions assigned to variables, passed as arguments, or return values
Features

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

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

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

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

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

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

- 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
Features

- 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)
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:**
  
  $$ (+ \ 3 \ 4) \quad ( (+ \ 3 \ 4) )$$
Function Application

- **Pair of parentheses: Function application**
  - First expression inside: Function
  - Remaining expressions: Arguments

- **Examples:**
  
  $$ \text{( + 3 4) } \quad \text{((+ 3 4))} $$

  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.
Creating Functions

- Evaluating a lambda expression yields a function
  - First argument to lambda: Formal parameters
  - Remaining arguments: Body of the function

- Example:
  \[
  (\text{lambda} \ (x) \ (* \ x \ x))
  \]
Creating Functions

- Evaluating a lambda expression yields a function
  - First argument to lambda: Formal parameters
  - Remaining arguments: Body of the function

- Example:

  \[(\text{lambda } (x) \ (\ast \ x \ x))\]

  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:**
  
  ```scheme
  (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:**
  
  ```lisp
  (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)
  
  (cond
   ((< 3 2) 1)
   ((< 4 3) 2)
   (else 3))

  Yields 4
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:**
  
  ```lisp
  (if (> a 0) (+ 2 3) (+ 2 "foo"))
  ```

  ```lisp
  (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:**

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

  Evaluates to 5 if \(a\) is positive; runtime type error otherwise.

  ```lisp
  (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))
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)) ❌

https://ilias3.uni-stuttgart.de/vote/0ZT9
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:
  - `(car '(2 3 4))`
  - `(cdr '(2 3 4))`
  - `(cons 2 '(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:
  
  - `(car '(2 3 4))`
  - `(cdr '(2 3 4))`
  - `(cons 2 '(3 4))`

  Yields 2

"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:

- `(car '(2 3 4))` yields 2
- `(cdr '(2 3 4))` yields `(3 4)`
- `(cons 2 '(3 4))` 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:
  - `(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:**
  ```lisp
  (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**
  - \( \textit{set!} \) for assignment to \textit{variables}
  - \( \textit{set-car!} \) for assigning head of list
  - \( \textit{set-cdr!} \) for assigning tail of list

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

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

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

- Example:

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

Prints ”hi there”
Iteration

- Several forms of loops, e.g., with \texttt{do}

- Example:

\[
((\lambda (n)
  (\texttt{do} ((i 0 (+ i 1))
    (a 0 b)
    (b 1 (+ a b)))
  ((= i n) b)
  (display b)
  (display " "))) 5)
\]
Iteration

- Several forms of loops, e.g., with \texttt{do}

- Example:

  \begin{verbatim}
  ((lambda (n)
   (do ((i 0 (+ i 1))
        (a 0 b)
        (b 1 (+ a b)))
      ((= i n) b)
      (display b)
      (display " ")))) 5)
  \end{verbatim}

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 \texttt{do}
- Example:

\[
((\lambda (n)
  \begin{array}{l}
  (\texttt{do } ((i 0 (+ i 1)))
  (a 0 b)
  (b 1 (+ a b)))
  ((= i n) b)
  (display b)
  (display " "))
\end{array}
)) \ 5)
\]

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:

  ```lisp
  ((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:

\[
\begin{align*}
&((\text{lambda } (n)) \\
&(\text{do } ((i 0 (+ i 1))) \\
&(\text{a 0 b}) \\
&(\text{b 1 (+ a b)})) \\
&(\text{((= i n) b)} \\
&(\text{display b}) \\
&(\text{display " ")})) 5)
\end{align*}
\]

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:**
  
  ```lisp
  (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**:
  
  ```lisp
  (eval (cons '+ (list '2 '3)))
  ```
  Constructs a list from the given arguments
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:
  
  \[
  \text{(eval (cons '+ (list '2 '3)))}
  \]

  Yields 5

  Constructs a list from the given arguments
Plan for Today

- Introduction
- A Bit of Scheme
- 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
(define double (lambda (x) (+ x x)))

Applicative-order

double (* 3 4)
⇒ double 12
⇒ + 12 12
⇒ 24

Normal-order

double (* 3 4)
⇒ + (* 3 4) (* 3 4)
⇒ + 12 (* 3 4)
⇒ + 12 12
⇒ 24

Doing extra work with normal-order
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
          ((= x 0) b)
          ((> x 0) c)))))

→ Doing extra work in applicative order

---

Applicative-order

(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
         ((= -1 0) 5)
         ((> -1 0) 7))
⇒ (cond (#t 3) ⇒ 3 .
       ...)
⇒ (+ 1 2)
⇒ 3

Normal-order

(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
         ((= -1 0) (+ 2 3))
         ((> -1 0) (+ 3 4)))
⇒ (cond (#t (+ 1 2)
         ...)
⇒ (+ 1 2)
⇒ 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
(define double (lambda (x) (+ x x)))
(define avg (lambda (x y) (/ (+ x y) 2)))

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

\[(\text{double } \text{avg } 2 \ 4))\]

\[\Rightarrow (\text{double } (\div (+ 2 \ 4) \ 2))\]

\[\Rightarrow (\text{double } (\div 6 \ 2))\]

\[\Rightarrow (\text{double } 3)\]

\[\Rightarrow (+ 3 \ 3)\]

\[\Rightarrow 6\]

5 steps

Normal order

\[(\text{double } \text{avg } 2 \ 4))\]

\[\Rightarrow (+ \text{avg } 2 \ 4 \ \text{avg } 2 \ 4))\]

\[\Rightarrow (+ (\div (+ 2 \ 4) \ 2) \ \text{avg } 2 \ 4))\]

\[\Rightarrow (+ (\div 6 \ 2) \ \text{avg } 2 \ 4))\]

\[\Rightarrow (+ 3 \ \text{avg } 2 \ 4))\]

\[\Rightarrow (+ 3 \ (\div (+ 2 \ 4) \ 2))\]

\[\Rightarrow (+ 3 \ (\div 6 \ 2))\]

\[\Rightarrow (+ 3 \ 3)\]

\[\Rightarrow 6\]

8 steps
Plan for Today

- Introduction
- A Bit of Scheme
- Evaluation Order