



Functional Programming

Lecture 1: Introduction

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Acknowledgements

This course is based on materials created by:

- Jiří Vyskočil, Michal Pěchouček
 - ČVUT, Czech Republic
 - Koen Claessen and Emil Axelsson
 - Chalmers University of Technology, Sweden
 - H. James Hoover
 - University of Alberta, Canada
 - Ben Wood
 - Wellesley College, USA
 - H. Abelson, G. J. Sussman and Julie Sussman
 - Massachusetts Institute of Technology, USA
 - Alan Borning
 - University of Washington, USA
 - R. Kent Dybvig
 - Indiana University, USA
- ...

What is functional programming?

Wikipedia: Functional programming is a **programming paradigm** that treats computation as the evaluation of mathematical functions.

Programming paradigm: a style of building the structure and elements of computer programs.

Goal of the course

1. Improve your programming skills!
 - master recursion
 - master problem decomposition
 - rethink side effects (stateless programs)
 - different perspective to the same problems
2. Learn principles of functional programming
 - has clear benefits for SOME problems
 - it is used in many other languages

Why do I care?

- quickly learn new programming languages
- programming paradigms change and develop
- no side effects is great for
 - parallelization
 - verification
 - communication with many clients
- understanding fundamentals of computation

Does anyone use it?

- Lisp: AutoCAD, Emacs, Gimp
- Haskell: Facebook, Google, Intel
- Scala: Twitter, eBay, LinkedIn
- Erlang: Facebook, Pinterest
- Clojure: Walmart, Atlassian
- Javascript: React (Redux)

Imperative vs. Declarative

- **Instructions** to change the computer's state
 - $x := x + 1$
 - `deleteFile("slides.pdf")`
- Are executed
 - have effects
- Run program by following instructions top-down
- Functions used to **declare** dependences between data values:
 - $z = g(y)$
 - $y = f(x)$
- Expressions are evaluated
 - result to a value
- Run program by evaluating dependencies

Pure functional programming

- No side effects
 - output of a function depends **only** on its inputs
 - function does not change anything in evaluation
 - can be evaluated in any order (many times, never)
- No mutable data
- More complex function based on recursion
 - no for/while cycles
 - natural problem decomposition
 - mathematical induction

Pure functional programming

- Forbids most of what you use in (C/Java)
 - we will show you do really not loose anything
 - it can be useful for many tasks
 - it often leads to more compact code !?!
- Substantially less time spent debugging
 - encapsulation, repeatability, variety of mistakes
- Focus on operations with symbols
- Easier parallelization and verification
- Generally less computationally efficient

Brief History

- Lambda calculus (1930s)
 - formal theory of computation older than TM
- Lisp = List processor (1950s)
 - early practical programming language
 - second oldest higher level language after Fortran
- ML = Meta language (1970s)
 - Lisp with types, used in compilers
- Haskell = first name of Curry (1990s)
 - standard for functional programming research
- Python, Scala, Java8, C++ 11,

What will we learn?

Lisp (Scheme)

Lambda calculus

Haskell

Why LISP?

- Extremely simple
- Reasonably popular
- Allows deriving all concepts from principles
- Directly matches lambda calculus

Why Haskell?

- Purely functional language
 - promotes understanding the paradigm
- Rich syntactic sugar (contrast to Lisp)
- Most popular functional language
- Standard for functional programming research
- Fast prototyping of complex systems
- Why not Scala?

Course organization

- Web: cw.fel.cvut.cz/wiki/courses/fup
- Lectures + Labs
- Homework – every 2 weeks (50 %)
 - 3x10 Scheme
 - 2x10 Haskell
 - must have at least 1 point from each and ≥ 25
 - Deadlines: -3 + -1 per day until +1 is left
- Programming exam (30 %)
- Test (20 %)

Suggested literature

[1] R. Kent Dybvig: The Scheme Programming Language, Fourth Edition, MIT Press, 2009.

<https://www.scheme.com/tspl4/>

[2] Greg Michaelson: An Introduction to Functional Programming Through Lambda Calculus, Dover edition, 2011.

[3] Harold Abelson and Gerald Jay Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, MIT Press, 1996.

<https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html>

For further resources see CourseWiki

Scheme

- Dialect of Lisp (such as Common Lisp, Racket)
- Created in 1970 at MIT by Steele and Sussman
- Last standard from 2007
 - The Revised⁶ Report on the Algorithmic Language Scheme (R6RS)
- Supports imperative programming
 - we will initially not use it (we want to learn FP)
- DrRacket: racket-lang.org
 - text editor + REPL (read-evaluate-print loop)

Scheme syntax

Scheme program is a collection of expressions

1. Primitive expressions
2. Compound expressions
3. Abstractions
4. Comments, conventions

Primitive expressions

Expression	Evaluates to
5	5
"abc"	"abc"
`abc	abc
#t	#t
# \A	# \A
+	#<procedure:+>
a	?

Basics data types

Numbers (infinite precision, complex, etc.)

`+, -, *, /, abs, sqrt, number?, <, >, =`

Logical values

`#t, #f, and, or, not, boolean?`

Strings

`"abc", "Hello !!!", string?, substring`

Other types

`symbol?, char?, procedure?, pair?, port?,
vector?`

Conventions

Special suffixes

? for predicates

! for procedures with side effects

-> in procedures that transform a type of an object

Prefix of character / string / vector procedures

char-, string-, and vector-

Comments

; starts a comment until the end of the line
on the line before the explained expression

;; still start a comment until the end of the line
used to comment a function or code segment

#| |# delimit block comments

Compound expressions

Infix notation

$1+2*5$

Prefix notation

$+ 1 * 2 5$

In Scheme, there are no operator preferences

$(+ 1 (* 2 5))$

S - expression

(fn arg1 arg2 ... argN)

(“operator of calling a function”

fn expression that evaluates to a *procedure*

argX arguments of the function

) end of function call

Conditional expressions

if

(if test-exp then-exp else-exp)

cond

(cond

(test-exp1 exp)

(test-exp2 exp)

(#t exp)

. . .)

Quote

Do not evaluate, just to return the argument
(quote exp)

Abbreviated by '

A quoted expression can be evaluated by *eval*
(eval (quote (+ 1 2)))

Evaluate part of the argument

(quasiquote (* 1 2 3 (unquote (+ 2 2)) 4 5))

Abbreviated by ` and , respectively

Define

Naming expressions

```
(define id exp)
```

Defining functions

```
(define (name <formals>) <body>)
```

Nested defines

```
(define (name <formals>)
  (define (fn <formals>)
    <body-using-fn>)
```

Identifiers

Keywords, variables, and symbols may be formed from the following set of characters:

the lowercase letters a through z,

the uppercase letters A through Z,

the digits 0 through 9, and

the characters ? ! . + - * / < = > : \$ % ^ & _ ~ @

cannot start with 0-9, +, -, @ (still usually works)

Recursion

A function calling itself

```
(define (fact n)
  (cond ((= 0 n) 1)
        (#t (* n (fact (- n 1))))))
  )
)
```

Avoiding infinite recursion

1. First expression of the function is a cond
2. The first test is a termination condition
3. The "then" of the first test is not recursive
4. The cond pairs are in increasing order of the amount of work required
5. The last cond pair is a recursive call
6. Each recursive call brings computations closer to the termination condition

Recursion

Tail recursion

Last thing a function does is the recursive call

Analytic / synthetic

Return value from termination condition / composed

Tree recursion

Function is called recursively multiple times (qsort)

Indirect recursion

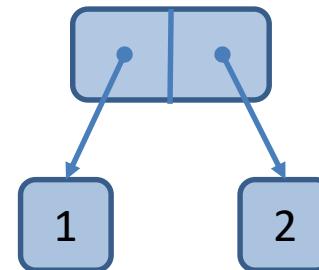
Function A calls function B which calls A

Pairs

Allow to construct compound data structures

(cons 1 2)

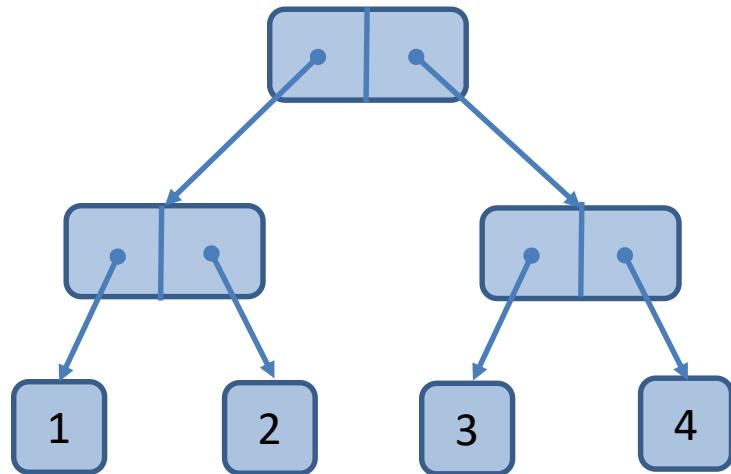
=> (1 . 2)



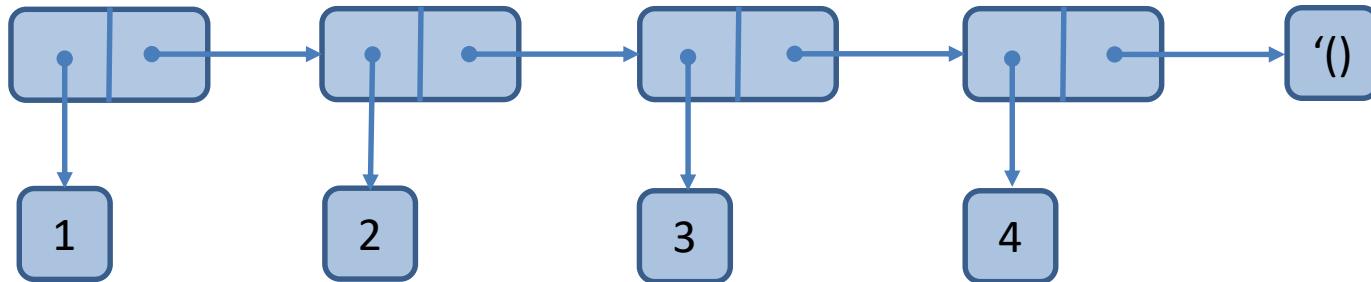
(cons

(cons 1 2)

(cons 3 4))



Lists



Lists are linked lists of pairs with ' ()' at the end

S-expressions are just lists

```
' (+ 1 2 3 4 5)
```

Lists can be created by a function cons or

```
(list item1 item2 ... itemN)
```

Lists

Pairs forming the lists can be decomposed by

car [*car*] first element of the pair

cdr [*could-er*] second element of the pair

(caddr x) shortcut for (car (cdr (cdr x)))

Empty list is a null pointer

null? tests whether the argument is the empty list

Last

Return the last element of a list

```
(define (last list)
  (cond
    ((null? (cdr list)) (car list))
    (#t (last (cdr list))))
  ))
```

What have we learned?

- Functional programming is an alternative programming paradigm
 - no side effects
 - no mutable data structures
 - focus on symbols
- Recursion is the key programming method
- Lists are a key data structure



Functional Programming

Lecture 2: Lambda abstraction

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz
xhorcik@fel.cvut.cz

Last lecture

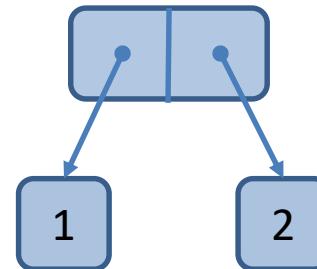
- What is (pure) functional programming
- Why do we care about it?
- Recursion is the main tool
- Scheme
 - S-expression, quote, identifiers, define, if, cond

Pairs

Allow to construct compound data structures

(cons 1 2)

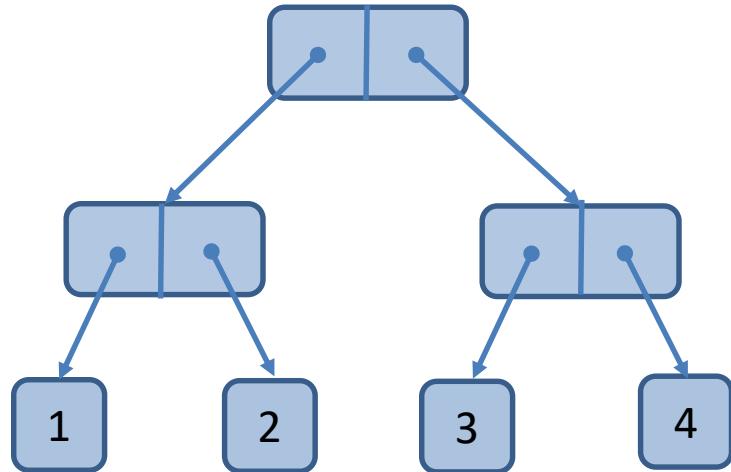
=> (1 . 2)



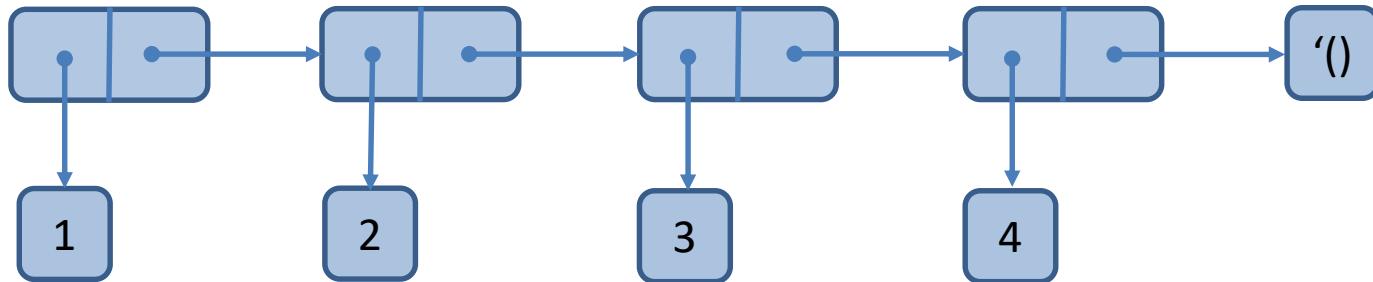
(cons

(cons 1 2)

(cons 3 4))



Lists



Lists are linked lists of pairs with '()' at the end

S-expressions are just lists

```
' (+ 1 2 3 4 5)
```

Lists can be created by a function cons or

```
(list item1 item2 ... itemN)
```

Lists

Pairs forming the lists can be decomposed by

car [*car*] first element of the pair

cdr [*could-er*] second element of the pair

(caddr x) shortcut for (car (cdr (cdr x)))

Empty list is a null pointer

null? tests whether the argument is the empty list

Append

```
;;; Append two lists
(define (append2 a b)
  (cond
    ((null? a) b)
    (else (cons (car a)
                 (append2 (cdr a) b)))))

)
```

Equality

Function = is only for numbers

Equivalence of the objects eqv?

(eqv? 1 1), (eqv? 'a 'a) ==> #t

(eqv? (list 'a) (list 'a)) ==> #f

More restrictive version is eq?

Typically the same pointer

Recursive version of eqv? on lists is

(equal? (list 'a) (list 'a)) ==> #t

Debugging Basics

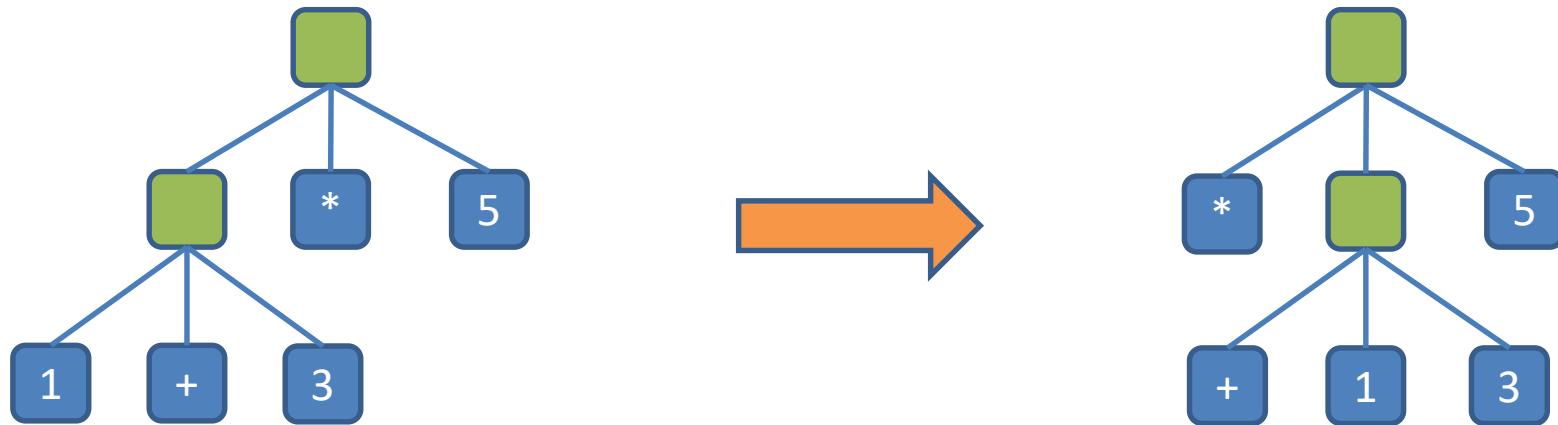
Tracing function calls and returns

```
#lang scheme  
(require racket/trace)  
(trace append2)  
(untrace append2)
```

Helper print-outs

```
(begin (display x)  
       (newline)  
       <do-work>)
```

Infix -> prefix



$((1 + 3) * 5)$

$(* (+ 1 3) 5)$

Evaluation strategy

Defines the order of evaluating the expressions
influences program termination, not the result

Evaluation of scheme is eager (or strict)

left to right

evaluate all arguments before executing a function

Evaluation of some special forms is lazy

if, and, or, **lambda**

Functions are 1st-class citizens

- They may be named by variables
- They may be passed as arguments to a function
- They may be returned by a function
- They may be included in data structures

Lambda abstraction

A construction for creating nameless procedures

(lambda (arg1 ... argN) <expr>)

Define for functions is an abbreviation

(define (<var> <formals>) <body>)

Is the same as

(define <var>
 (lambda (<formals>) <body>))

Filter

```
;; Filter a list by the given predicate
(define (my-filter pred lst)
  (cond
    ((null? lst) '())
    ((pred (car lst)))
    (cons (car lst)
          (my-filter pred (cdr lst))))
    (else (my-filter pred (cdr lst))))
  ))
```

Derivative

$$Dg(x) \approx \frac{g(x + dx) - g(x)}{dx}$$

```
(define dx 0.00001)
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x)) dx)
  )
)
```

Functions as data

```
(define (mult-fn fns x y)
  (cond
    ((null? fns) '())
    (else
      (cons
        ((car fns) x y)
        (mult-fn (cdr fns) x y))))
  )
)
```

Let

Motivation

reuse of computation/result is often required

e.g., minimum, roots from the labs

```
(let ( (<var1> <exp1>
        (<var2> <exp2>) )
              <body-using-var1-var2>)
```

$$f(x, y) = x(1 + xy)^2 + y(1-y) + (1+xy)(1-y)$$

```
;;; Local variables
(define (f x y)
  (let
    ((a (+ 1 (* x y))))
    (b (- 1 y)))
  (+ (* x a a)
     (* y b)
     (* a b))
  )
)
```

Implementing let

```
(let ((x <exp1>)
      (y <exp2>)) <body>)
```

Can be implemented as

```
((lambda (x y) <body>) <exp1> <exp2>)
```

Let as lambda

```
(define (f2 x y)
  ((lambda (a b)
    (+ (* x a a)
       (* y b)
       (* a b)))
   )
  (+ 1 (* x y)))
  (- 1 y))
)
```

Let*

We might want to use the earlier definitions in the following.

```
(let ((x <exp>))
```

```
  (let ((y <exp-with-x>)) <body-x-y>)
```

Equivalent to

```
(let* ((x <exp>)
```

```
  (y <exp-with-x>)) <body-x-y>)
```

Quicksort

```
(define (qsort lst cmp)
  (cond
    ((null? lst) '())
    (else (let*
            ((pivot (car lst))
             (smaller (lambda (x) (cmp x pivot)))
             (greater (lambda (x) (not (cmp x pivot))))))
      (append
        (qsort (filter smaller (cdr lst)) cmp)
        (list pivot)
        (qsort (filter greater (cdr lst)) cmp)))
      )
    )
  )
)
```

Scheme home assignments

Three connected assignments

Robot simulation

Population evaluation

Code synthesis

Why this assignment?

Work on your own

Submit by midnight of the day after your lecture

<https://cw.felk.cvut.cz/brute/> (in 2 weeks)



Functional Programming

Lecture 3: Higher order functions

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Last lecture

- Evaluation strategies
- Debugging
- Lambda abstraction
$$(\lambda \text{ (arg1 ... argN)} \text{ <expr>})$$
- Let, let*, append, quicksort
- Home assignment 1

Higher order functions

Functions taking other functions as arguments or returning functions as the result

- Capture and reuse common patterns
- Create fundamentally new concepts
- The reason why functional programs are compact

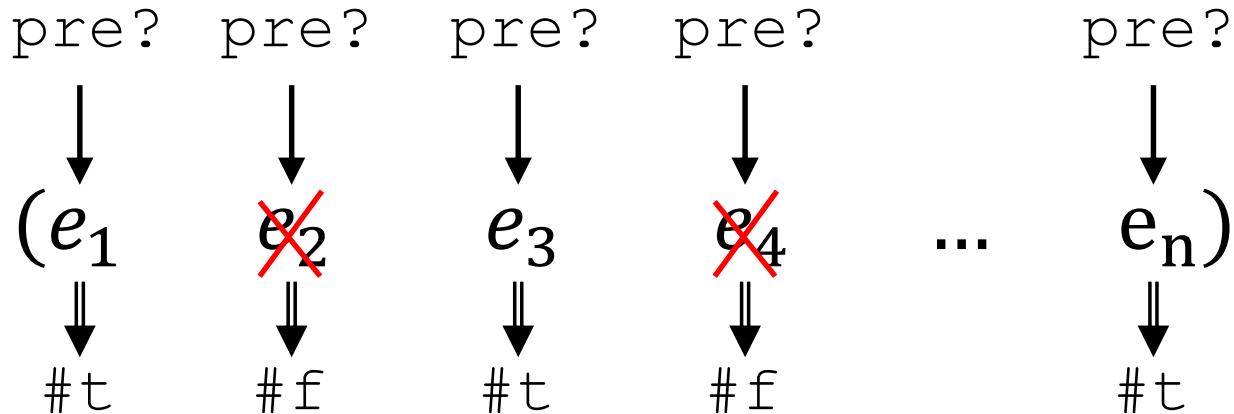
Order of data

- Order 0
 - Non function data
- Order 1
 - Functions with domain and range of order 0
- Order 2
 - Functions with domain and range of order 1
- Order k
 - Functions with domain and range of order k-1

Filter

(filter pre? list)

In the previous lecture



$(e_1 \ e_3 \ \dots \ e_n)$

Variable number of arguments

```
(define (fn arg1 arg2 . args-list) <body>)
```

After calling, the remaining arguments are in args-list.

Corresponding lambda term

```
(lambda (fn arg1 arg2 . args-list) <body>)
```

Alternatively

```
(lambda args-list <body>)
```

Apply

Applies a function to the arguments

(apply proc arg1 ... rest-args)

Example:

(apply + 1 2 3 ' (4 5))

Append

```
;;; Append arbitrary list of lists
(define (my-append . args)
  (cond
    ((null? args) args)
    (else
      (append2 (car args)
               (apply my-append (cdr args))))
    )
  )
)
```

Apply

```
(define (my-apply1 f args)
  (define (quote-all lst)
    (cond
      ((null? lst) '())
      (else (cons
              `(quote ,(car lst))
              (quote-all (cdr lst))))))
    )
  )
  (eval (cons f (quote-all args)))
)
```

Apply

```
(define (my-apply f . args)
  (define (append-last lst)
    (cond
      ((null? (cdr lst)) (car lst))
      (else (cons (car lst)
                   (append-last (cdr lst)))))
    )
  )
  (my-apply1 f (append-last args))
)
```

Compose

(compose f g)

Arguments are functions

Returns a function

```
(define (compose1 f g)
  (lambda args
    (f (apply g args)))
  )
```

Inc each / dec each

```
(define (incall list)
  (cond ((null? list) '())
        (#t (cons (+ (car list) 1)
                  (incall (cdr list))))))

(define (decall2 list)
  (cond ((null? list) '())
        (#t (cons (- (car list) 2)
                  (decall2 (cdr list))))))
```

Map

```
(define (map1 f lst)
  (cond
    ((null? lst) '())
    (else
      (cons (f (car lst))
            (map1 f (cdr lst))))
    )
  )
)
```

Map

Calls proc of N arguments on all elements of the list and returns the result as a list

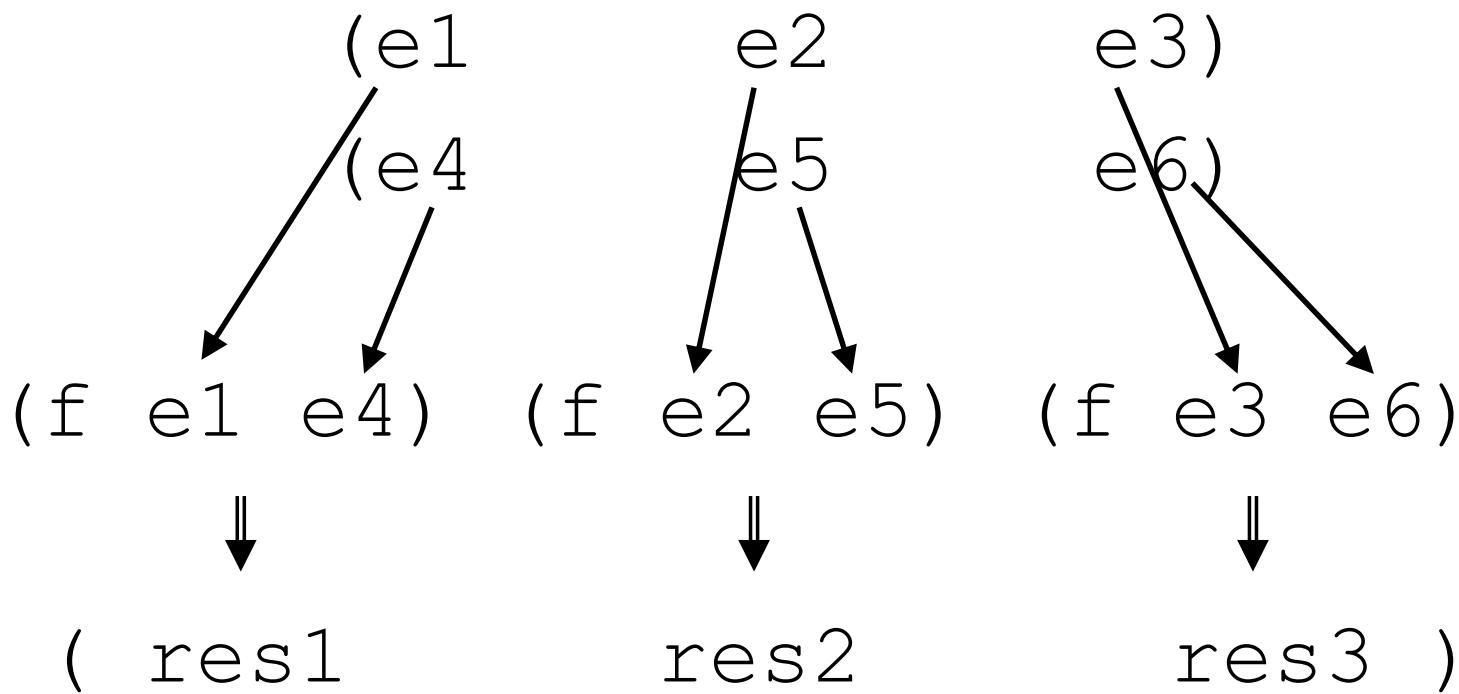
```
(map proc list1 list2 ... listN)
```

Example:

```
(map + ' (1 2 3) ' (4 5 6))
```

Map

```
(map f ' (e1 e2 e3) ' (e4 e5 e6) )
```



Map

```
(define (my-map f . args)
  (cond
    ((null? (car args)) '())
    (else
      (cons
        (apply f (map1 car args))
        (apply my-map (cons f (map1 cdr args))))
      )
    )
  )
)
```

Min / sum

```
(define (min-all list)
  (cond ((null? (cdr list)) (car list))
        (#t (min (car list) (min-all (cdr list))))))

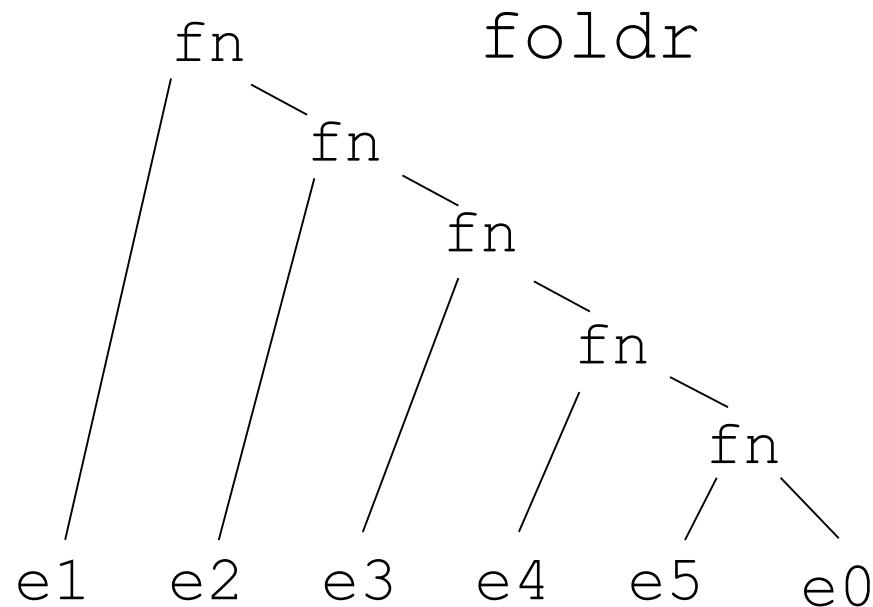
(define (sum-all list)
  (cond ((null? (cdr list)) (car list))
        (#t (+ (car list) (sum-all (cdr list))))))

(define (reduce f list)
  (cond ((null? (cdr list)) (car list))
        (#t (f (car list) (reduce f (cdr list))))))
```

Reduce

Often called foldr and foldl in scheme

(foldr fn e0 ' (e1 e2 e3 e4 e5))

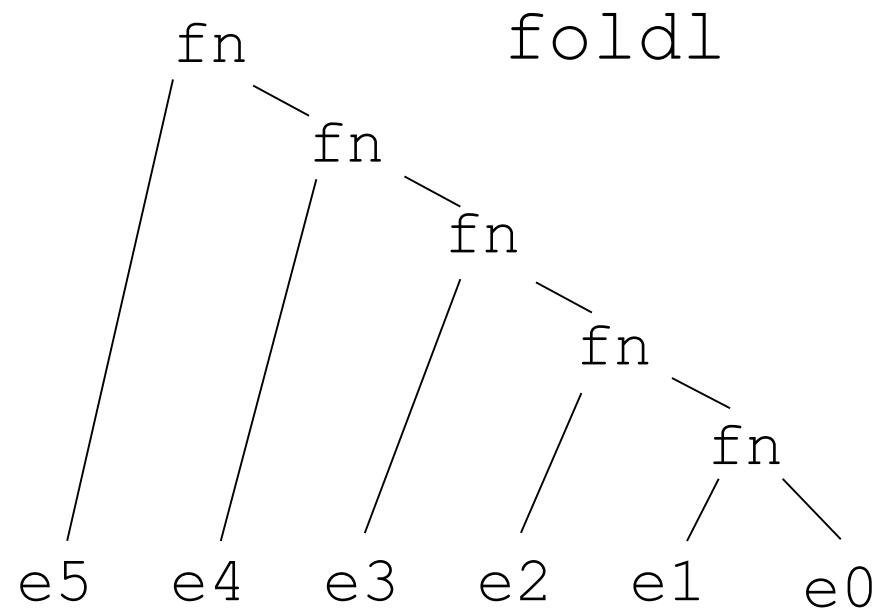


Foldr

```
(define (my-foldr f a lst)
  (cond
    ((null? lst) a)
    (else
      (f (car lst)
          (my-foldr f a (cdr lst)))
      )
    )
  )
)
```

Foldl

```
(foldl fn e0 ' (e1 e2 e3 e4 e5) )
```



Foldl

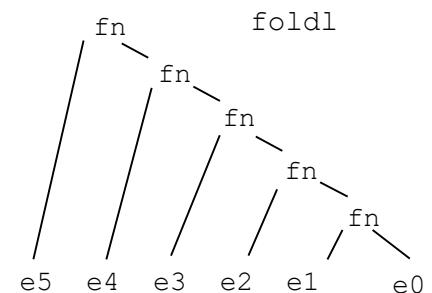
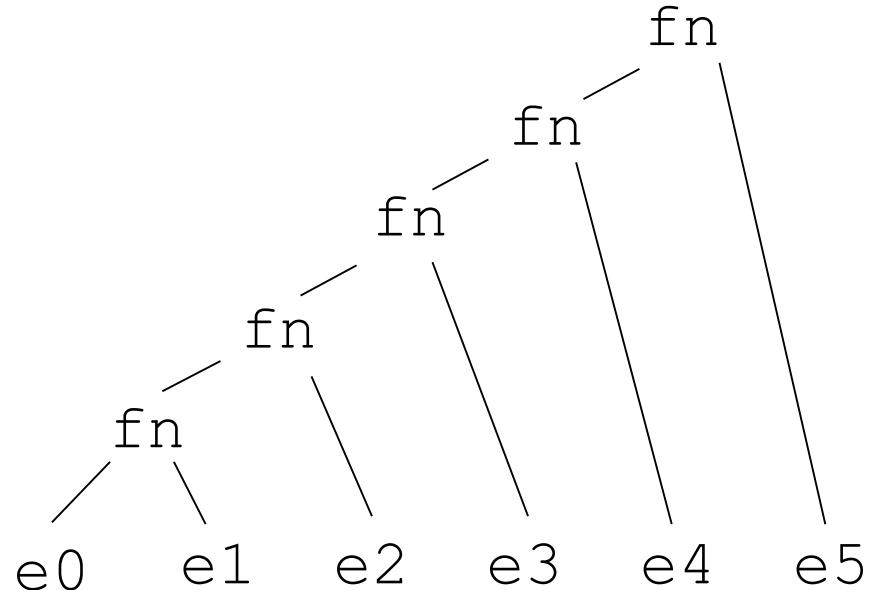
```
(define (my-foldr f a lst)
  (cond
    ((null? lst) a)
    (else (f (car lst) (my-foldr f a (cdr lst))))
    )
  )
)

(define (my-foldl f a lst)
  (cond
    ((null? lst) a)
    (else (my-foldl f (f (car lst) a) (cdr lst)))
    )
  )
)
```

Swap

```
(define (swapargs f)
  (lambda (x y) (f y x)))
```

```
(foldl (swapargs fn) e0 ' (e1 e2 e3 e4 e5))
```



Every / some

(every pred list1 ... listN)

```
(define (every1 pred lst)
  (cond
    ((null? lst) #t)
    (else (and (pred (car lst))
                (every1 pred (cdr lst))))))
  )
)

(define (some1 pred lst)
  (not
    (every1 (lambda (x) (not (pred x))) lst)))
```

Combining higher order functions

- add-only-numbers
- some
- flatten
- L2 norm
- filter
- length

Summary

- Higher order functions take functions as arguments or return functions
- Used to capture/reuse common patterns
- Create fundamentally new concepts
- Filter, apply, map, fold, swap



Functional Programming

Lecture 4: Closures and lazy evaluation

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Last lecture

- Functions with variable number of arguments
- Higher order functions
 - map, foldr, foldl, filter, compose

Binding scopes

A portion of the source code where a value is bound to a given name

- Lexical scope
 - functions use bindings available where defined
- Dynamic scope
 - functions use bindings available where executed

Ben Wood's slides

Eval

Eval is a function defined in the top level context

```
(define x 5)
(define (f y)
  (eval `(+ x y)))
```

Fails because of lexical scoping

Eval in R5RS

(eval expression environment)

Where the environment is one of

(interaction-environment) (global defines)

(scheme-report-environment x) (RXRS defines)

(null-environment x) (only special forms)

None of them allows seeing local bindings

Cons

```
(define (my-cons a b)
  (lambda (m) (m a b)))

(define (my-car p)
  (p (lambda (x y) x)))

(define (my-cdr p)
  (p (lambda (x y) y)))
```

Currying

Transforms function to allow partial application

not curried $f: A \times B \rightarrow C$

curried $f: A \rightarrow (B \rightarrow C)$

not curried $f: A \times B \times C \rightarrow D$

curried $f: A \rightarrow (B \rightarrow (C \rightarrow D))$



Picture: Haskell Curry from Wikipedia

Currying

```
(define (my-curry1 f)
  (lambda args
    (lambda rest
      (apply f (append args rest)))))
```

```
(define gmap (my-curry1 map))

(define (li str)
  (string-append "<li>" str "</li>"))

((gmap li) '("One" "Two" "Three"))
```

Lazy if using and/or

```
(define (my-if t a b)
  (or (and t a) b))

(define (my-lazy-if t a b)
  (or (and t (a)) (b)))

(my-lazy-if (< 1 2)
            (lambda () (print "true"))
            (lambda () (print "false")))
```

Syntax macros

```
(define-syntax macro-if
  (syntax-rules ()
    ((macro-if t a b)
     (my-lazy-if
      t
      (lambda () a)
      (lambda () b)))))
```

Lazy evaluation

lambda () can be used to delay evaluation

Delayed function is called "thunk"

Useful for

Lazy evaluation of function argument (call by name)

Streams – potentially infinite lists

Delay / Force

```
(define-syntax w-delay
  (syntax-rules ()
    ((w-delay expr) (lambda () expr)))))

(define-syntax w-force
  (syntax-rules ()
    ((w-force expr) (expr))))
```

Streams

```
(define (ints-from n)
  (cons n (w-delay (ints-from (+ n 1))))))

(define nats (ints-from 1))

(define (first stream) (car stream))

(define (rest stream)
  (if (pair? (cdr stream))
      (cdr stream)
      (w-force (cdr stream)))))
```

Lazy map

```
(define (lazy-map f s)
  (cond ((null? s) '())
        (else
          (cons (f (first s))
                (w-delay
                  (lazy-map f (rest s))
                  )
                )
          )
        )
      )
  )
```

Home assignment 2

Population evaluator

- evaluate a collection of programs for the robot
- (evaluate

`<prgs>`

`<pairs>`

`<threshold>`

`<stack_size>)`

Summary

- Binding scopes
 - Lexical vs. dynamic
- Closures
 - Code + environment pointer
 - Way to "store data" in a function
 - Tool for lazy evaluation
- Streams

Lexical Scope and Function Closures

Free variables

Variables used but not bound within function bodies.

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

x is a free variable in
the definition of f.

Big question:

What is the value of x when we evaluate the body
of (lambda (y) (+ x y)) here?

Visualize in DrRacket, draw environments.

Example

Demonstrates lexical scope without higher-order functions:

defines a function that, when called,
evaluates body $(+ x y)$ in an environment
where **x** is bound to 1
and **y** is bound to the argument

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

1. Looks up **f** in current environment, finding this.
2. Evaluates $(+ x y)$ in **current** environment, producing 5 .
3. Calls the function with argument 5:
 - Evaluates the body in the **old** environment, producing 6 .

Closures

revising our definition of functions

A **function definition expression** evaluates to a **function closure** value.

A **function closure** has **two parts**:

- **code** of function
- **environment** where the function was defined

Not a cons cell.
Cannot access pieces.

A **function call expression**:

- Evaluates the code of a function closure
- In the environment of the function closure

Example

Demonstrates lexical scope without higher-order functions:

Creates a closure and binds **f** to it:

Code: `(lambda (y) (+ x y))`

Environment: $f \rightarrow$ this closure, $x \rightarrow 1$

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
  (let ([x 2]
        [y 3])
    (f (+ x y))))
```

1. Looks up **f** in current environment, finding this closure.
2. Evaluates $(+ x y)$ in **current** environment, producing 5 .
3. Evaluates the closure's function body $(+ x y)$ in the closure's environment ($f \rightarrow$ the closure, $x \rightarrow 1$), extended with $y \rightarrow 5$, producing 6 .

The Rule: Lexical Scope

A function body is evaluated in the environment where the function was defined (created), extended with bindings for the arguments.

Next:

- Even taking / returning functions with higher-order functions!
- Makes first-class functions much more powerful.
 - Even if counterintuitive at first.
- Why alternative is problematic.

More examples in closures.rkt, notes. Draw...

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

x → 1

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
               [g (f 4)]
               [y 5]))
```

```
  (g 6)) )
```

eval

env

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
                [f 4]
                [g 5]))
```

```
  (g 6)) )
```

x → 1

f → env

```
(lambda (y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z)))
```

env

eval

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
               [g (f 4)]
               [y]))
```

```
  (g 6))
```

x → 1

f → env

env

```
(lambda (y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z)))
```

x → 3

eval

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

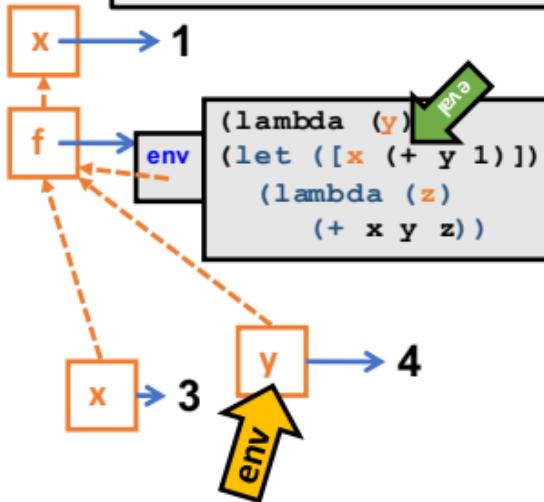
```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
               [g (f 4)]
               [y 5]))
```

```
  (g 6)) )
```



Ex: Returning a function

env pointer

shows env structure, by pointing to
“rest of environment”

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
```

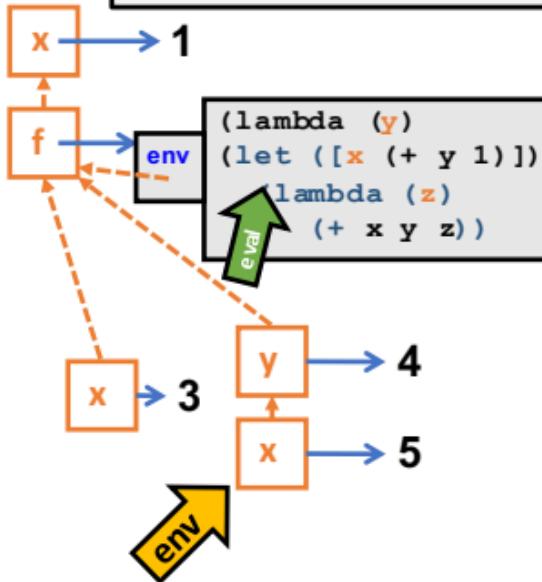
```
      (+ x y z)))
```

```
(define (g z) (let ([x 3]
```

```
                 [f 4])
```

```
                 [y 5]))
```

```
(g 6)) )
```



Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
```

```
      (+ x y z)))
```

```
(define z (let ([x 3]
```

```
           [g (f 4)]
```

```
           [y 5])
```

```
  (g 6))
```

eval

x → 1

f → env

```
(lambda (y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z)))
```

x → 3

y → 4

x → 5

y → 5

```
env
```

4

5

env

```
(lambda (z)
  (+ x y z))
```

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
               [g (f 4)]
               [y 5]))
```

eval

```
(g 6)) )
```

x → 1

f → env

```
(lambda (y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z)))
```

x → 3

y → 4

x → 5

x → 5

env

```
(lambda (z)
  (+ x y z))
```

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

```
    (lambda (z)
```

```
      (+ x y z)))
```

```
(define z (let ([x 3]
```

```
            [g (f 4)]
```

```
            [y 5]))
```

```
  (g 6)) )
```

eval

x → 1

f → env

```
(lambda (y)
  (let ([x (+ y 1)])
    (lambda (z)
      (+ x y z)))
```

x → 3

g → y

y → 5

y → 4

x → 5

env

```
(lambda (z)
  (+ x y z))
```

env

val

6

Ex: Returning a function

env pointer

shows env structure, by pointing to "rest of environment"

binding

maps variable name to value

```
(define x 1)
```

```
(define (f y)
```

```
  (let ([x (+ y 1)])
```

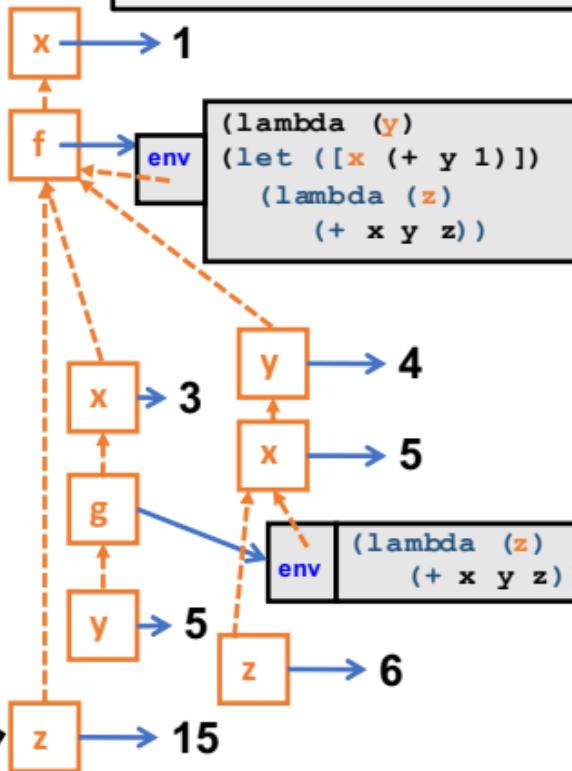
```
    (lambda (z)
      (+ x y z)))
```

```
(define z (let ([x 3]
               [g (f 4)]
               [y 5]))
```

```
  (g 6)) )
```



env



Why lexical scope?

Lexical scope: use environment where function is defined

Dynamic scope: use environment where function is called

History has shown that lexical scope is almost always better.

Here are some precise, technical reasons (not opinion).

Why lexical scope?

1. Function meaning does not depend on variable names.

Example: change body of `f` to replace `x` with `q`.

- Lexical scope: it cannot matter
- Dynamic scope: depends how result is used

```
(define (f y)
  (let ([x (+ y 1)])
    (lambda (z) (+ x y z))))
```

Example: remove unused variables.

- Dynamic scope: but maybe some `g` uses it (weird).

```
(define (f g)
  (let ([x 3])
    (g 2)))
```

Why lexical scope?

2. Functions can be understood fully where defined.

Example: dynamic scope tries to add `#f`, unbound variable `y`, and `4`.

```
(define (f y)
  (let ([x (+ y 1)])
    (lambda (z) (+ x y z)))
(define x #f)
(define g (f 7))
(define a (g 4))
```

Why lexical scope?

3. Closures automatically “remember” the data they need.

More examples, idioms later.

```
(define (greater-than-x x)
  (lambda (y) (> y x)))

(define (no-negs xs)
  (filter (greater-than-x -1) xs))

(define (all-greater xs n)
  (filter (lambda (x) (> x n)) xs))
```

Dynamic scope?

- Lexical scope definitely the right default for variables.
 - Very common across modern languages
- Early LISP used dynamic scope.
 - even though inspiration (lambda calculus) has lexical scope
 - Later "fixed" by Scheme (Racket's parent) and other languages.
- Dynamic scope is very occasionally convenient:
 - Racket has a special way to do it.
 - Perl
 - Most languages are purely lexically scoped.



Functional Programming

Lecture 5: Imperative aspects of Scheme

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Last lecture

- Binding scopes
 - Lexical vs. dynamic
- Closures
 - Code + environment pointer
 - Way to "store data" in a function
 - Tool for lazy evaluation
- Streams

Streams recap.

```
(define-syntax w-delay
  (syntax-rules ()
    ((w-delay expr) (lambda () expr)))))

(define-syntax w-force
  (syntax-rules ()
    ((w-force expr) (expr)))))

(define (lazy-map f s)
  (cond ((null? s) '())
        (else
          (cons
            (f (first s))
            (w-delay (lazy-map f (rest s)))))))
```

Stream map

```
(define (smap f . streams)
  (if (null? (car streams)) '()
      (cons (apply f (map first streams))
            (w-delay
              (apply smap f
                     (map rest streams)))))))
```

Implicitly defined streams

Imperative aspects of scheme

Until now, we did not need any mutable states

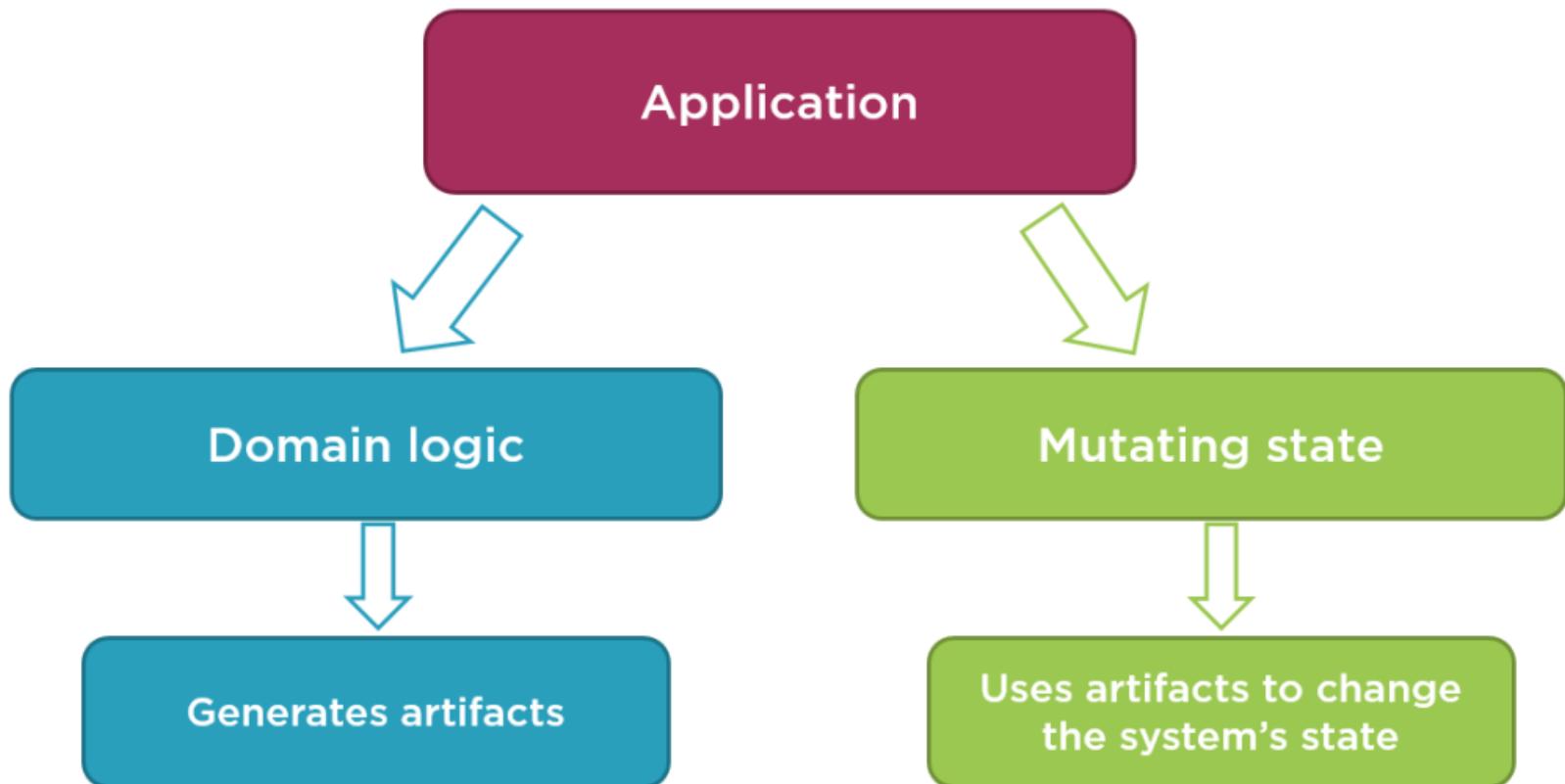
- Fully thread safe

- No exact (defensive) copies

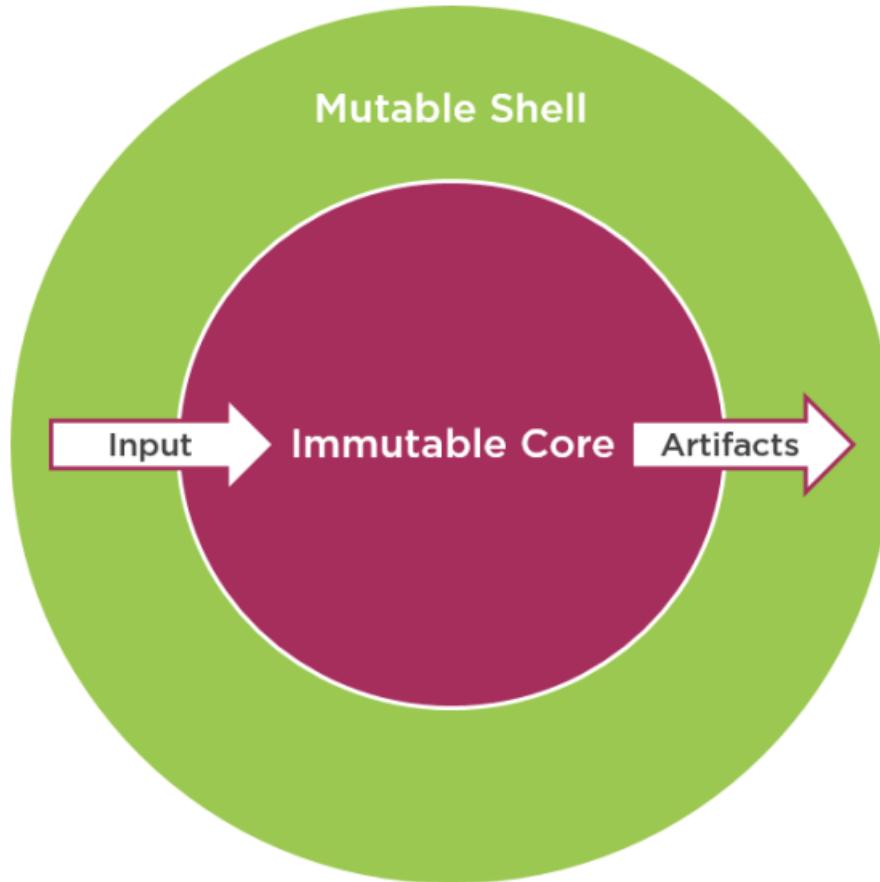
- No temporal coupling

Do not use in assignments!!!

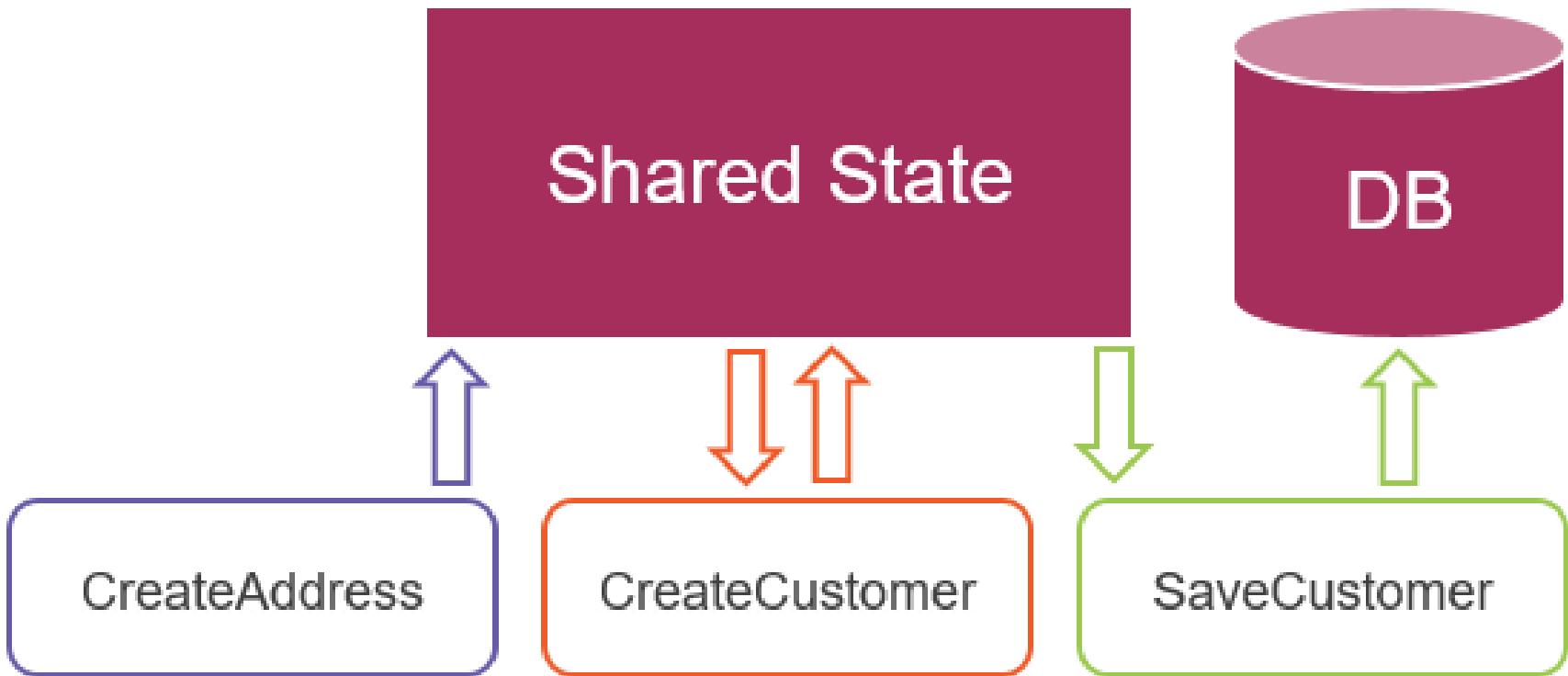
Immutable architecture



Immutable architecture



Temporal coupling



Set!

(set! id expr)

Assigns the value of expr to variable id

The id has to be already defined!

It is not the same as redefining the variable

```
(define x 1)
(define (foo)
  (define x 4)
  x)
(define (bar)
  (set! x 4)
  x)
```

We can have a state now!

```
(define counter 1)
(define (inc-counter)
  (set! counter (+ counter 1)) counter)

> (inc-counter)
> (inc-counter)
> (inc-counter)
```

Promise

Our weak delay/force may evaluate many times

State can be used to save the evaluated result

Delay with memoization :

```
(define (make-promise thunk)
  (let ((already-run? #f)
        (result #f))
    (lambda ()
      (if already-run? result
          (begin (set! result (thunk))
                 (set! already-run? #t)
                 result)))))
```

Vectors

- Heterogeneous objects
- Indexed by integers, starting from 0
- Typically faster and smaller than lists

(vector obj ...)

(make-vector k)

(make-vector k fill)

(vector-ref vector k)

(vector-set! vector k obj)

(list->vector list)

Iteration

```
(do ((<variable1> <init1> <step1>) ...)
    (<test> <expression> ...)
    <command> ...)
```

Create a vector initialized 0...length-1

```
(define (int-vec n)
  (let ((vec (make-vector n)))
    (do ((i 0 (+ i 1)))
        ((= i n) vec)
        (vector-set! vec i i))))
```

Letrec

Sometimes, we need all names available in the expressions

```
(letrec
  ((fact (lambda (n)
            (if (= n 1)
                1
                (* n (fact (- n 1)))))))
  (fact 10))
```

Letrec

```
(letrec ((x1 e1) ... (xn en)) body)
```

Is a macro expanding to

```
(let ((x1 'undefined) ... (xn 'undefined))
  (let ((t1 e1) ... (tn en))
    (set! x1 t1)
    ...
    (set! xn tn)))
  body)
```

All expressions must evaluate without evaluating

x_1, \dots, x_n

Closures in Impure Languages

Closures store data with functions

```
(define (count-clo)
  (let ((i 0))
    (lambda ()
      (begin (set! i (+ i 1))
             i))))
```

Random

```
(define random
  (let ((a 69069)
        (b 1)
        (m (expt 2 32))
        (seed 20200323))
    (lambda args
      (if (null? args)
          (begin
            (set! seed
                  (modulo (+ (* a seed) b) m))
            (/ seed m)))
          (set! seed (car args))))))
```

Summary

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
 - random access in $O(1)$
 - I/O operations
 - objects with states
 - ...



Functional Programming

Lecture 6: Imperative scheme and parallelism

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Last lecture

- We do not need to modify the state if we compute a function
- States break nice properties of pure FP
- Make the pure part of programs as large as possible
- States can sometimes be useful
 - random access in $O(1)$
 - memoization

"Classes and objects"

```
(define (make-account balance)
  (define (withdraw x)
    (if (>= balance x)
        (begin (set! balance (- balance x))
               balance)
        (error "Not enough money!!!!")))
  (define (deposit x)
    (set! balance (+ balance x))
    balance)
  (define (dispatch name)
    (cond ((eq? name 'withdraw) withdraw)
          ((eq? name 'deposit) deposit)
          (else (error "Unknown request")))))
  dispatch)
```

Lists modifications

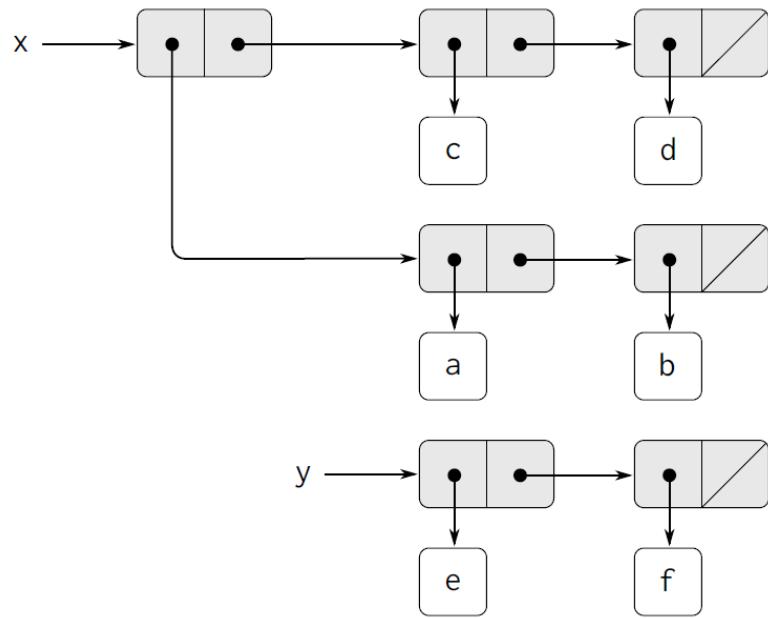
In R5RS, we can modify lists using

set-car!, set-cdr!

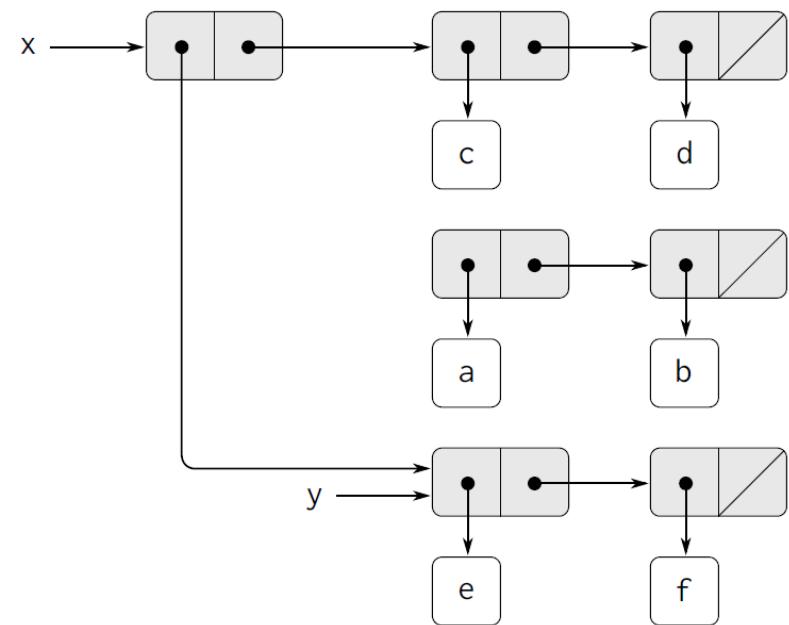
List are immutable by default with #lang scheme

Need to use mcons, mcadr, set-mcadr!, ...

(set-car! x y)

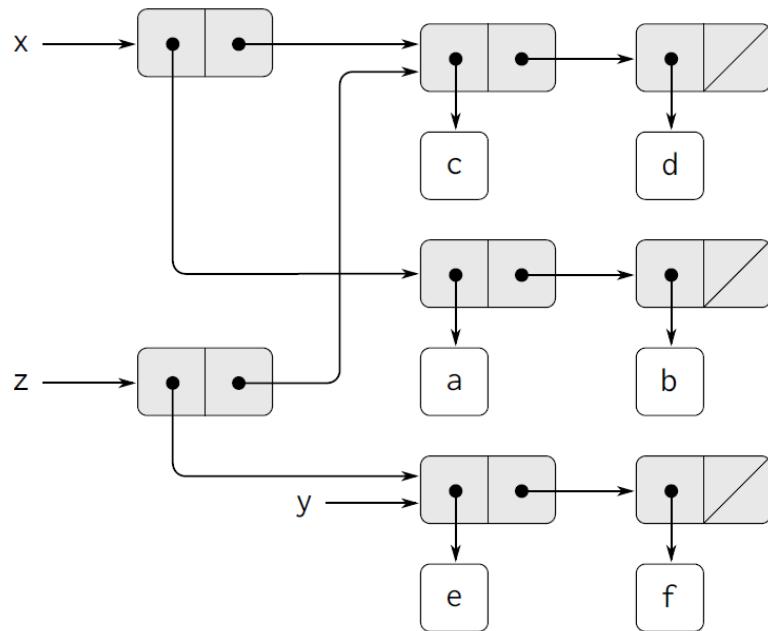


$x = ((a\ b)\ c\ d),\ y = (e\ f)$

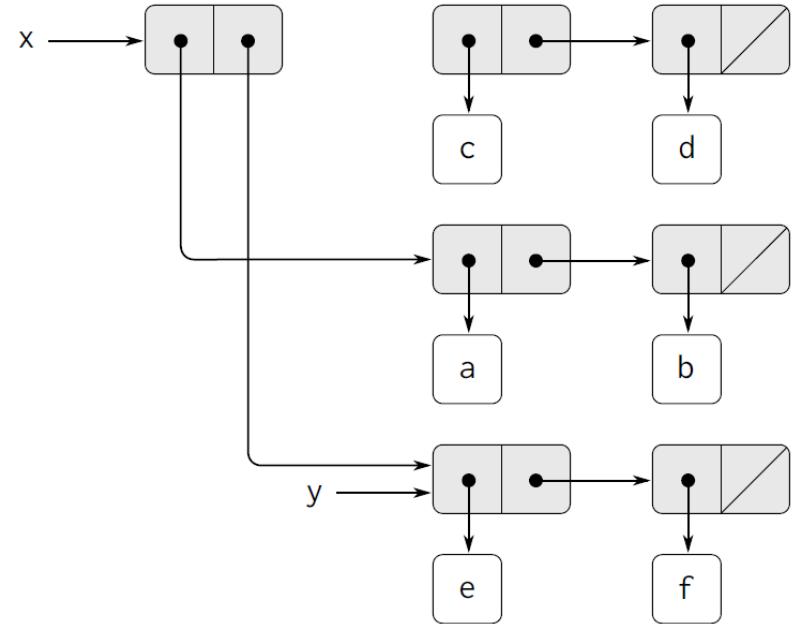


$x = ((e\ f)\ c\ d),\ y = (e\ f)$

(set-cdr! x y)

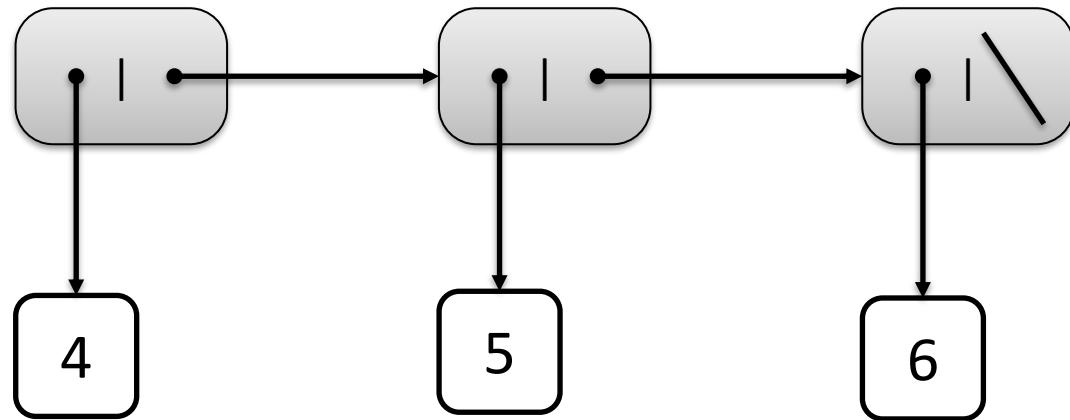
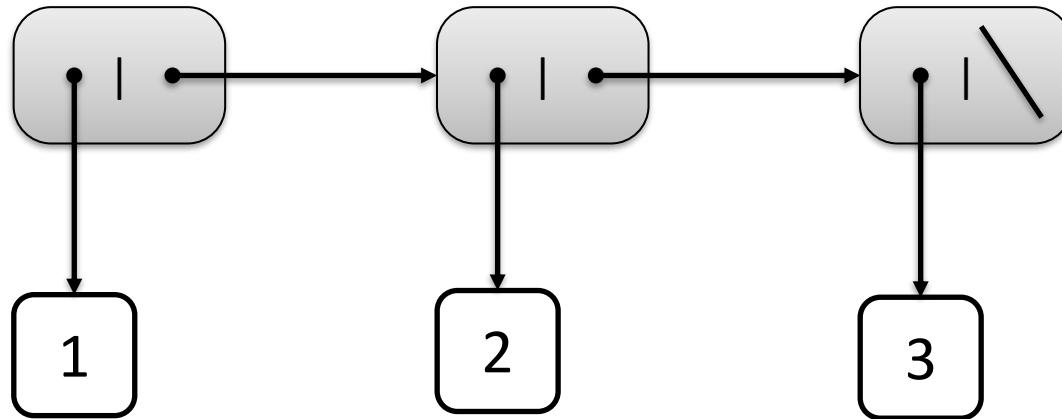


$$x = ((a \ b) \ c \ d), \ y = (e \ f)$$

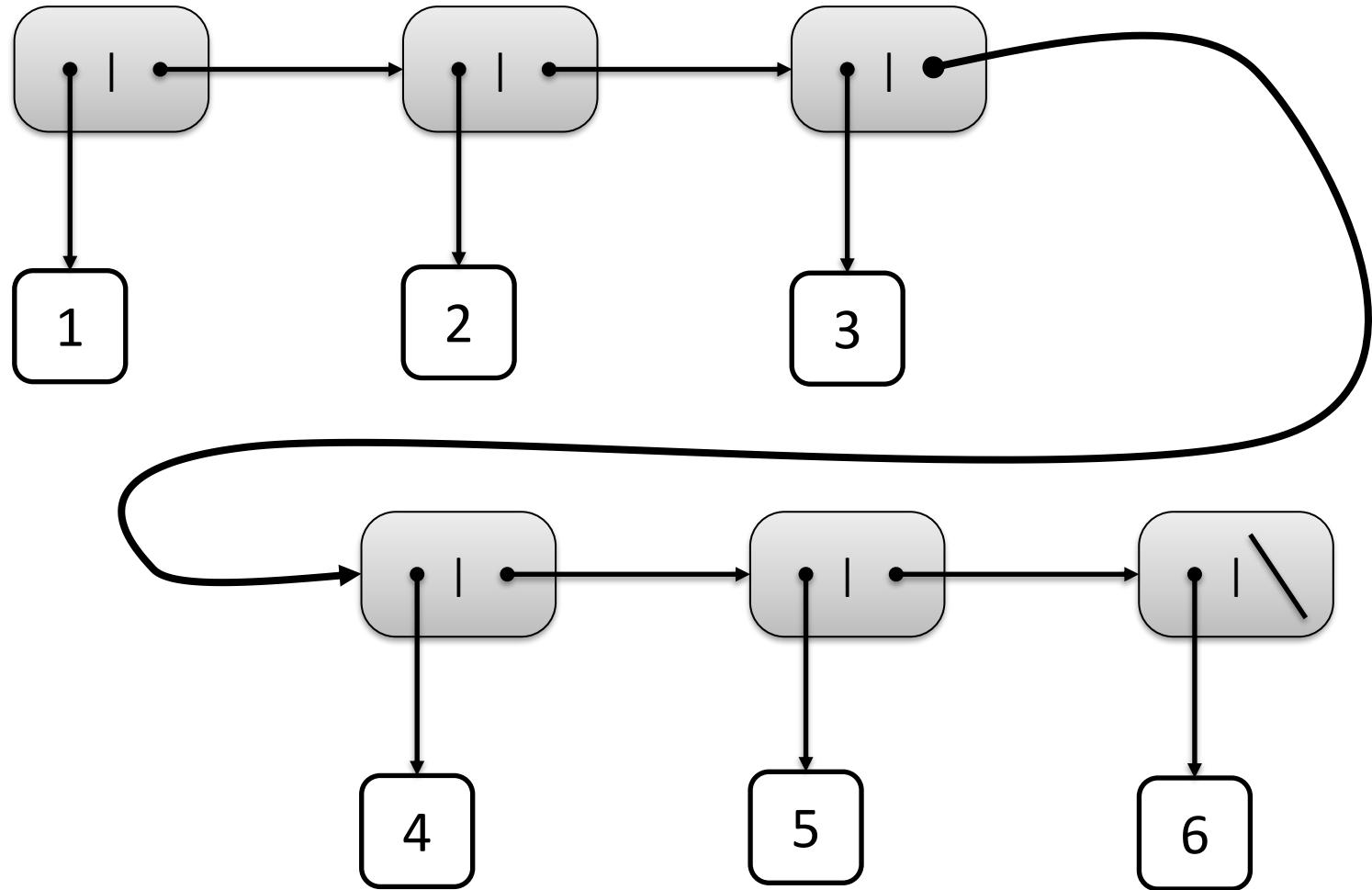


$$x = ((a \ b) \ e \ f), \ y = (e \ f)$$

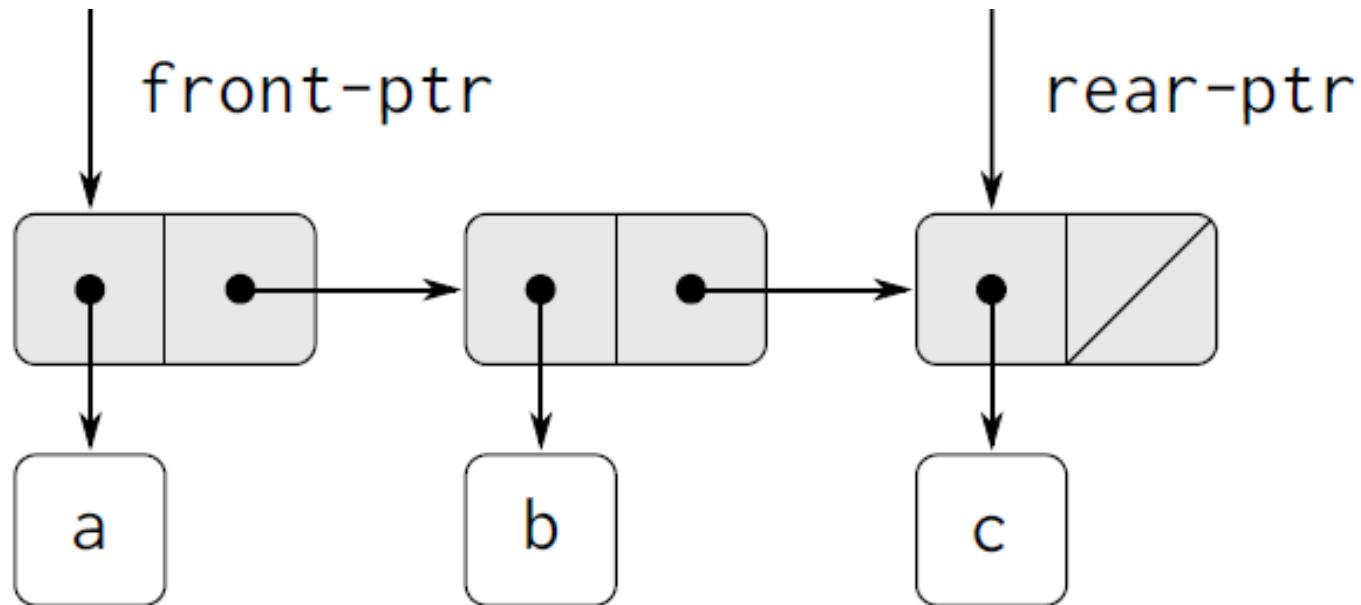
Append!



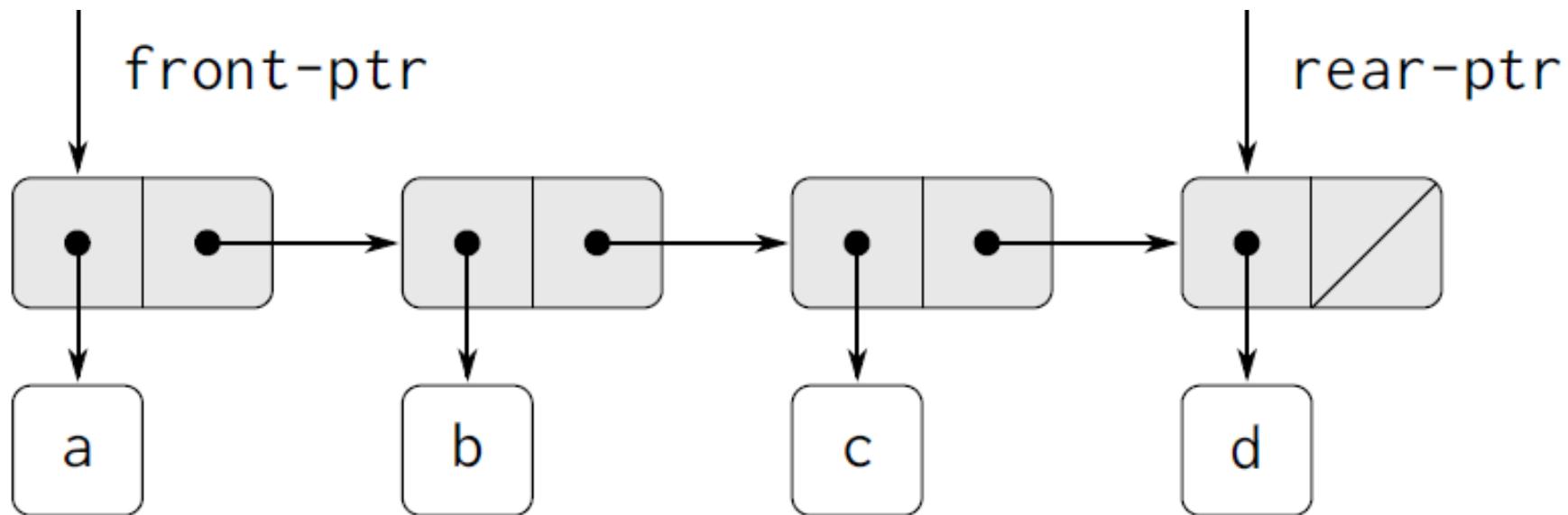
Append!



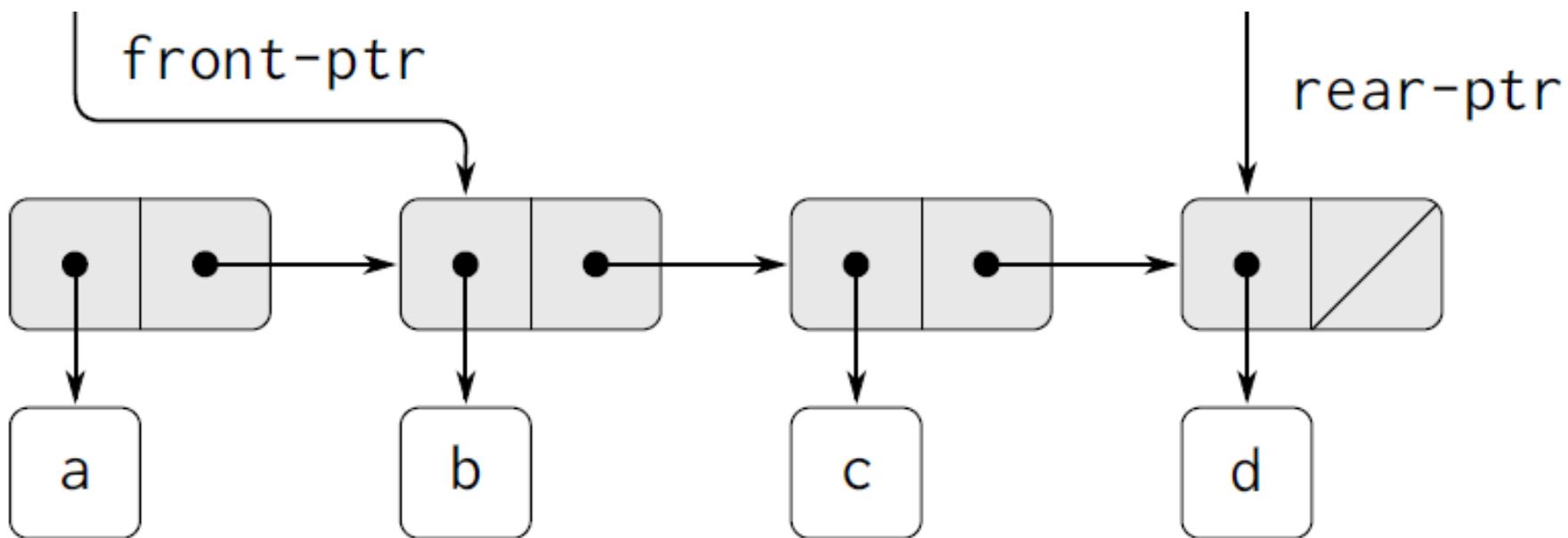
Queue



Insert



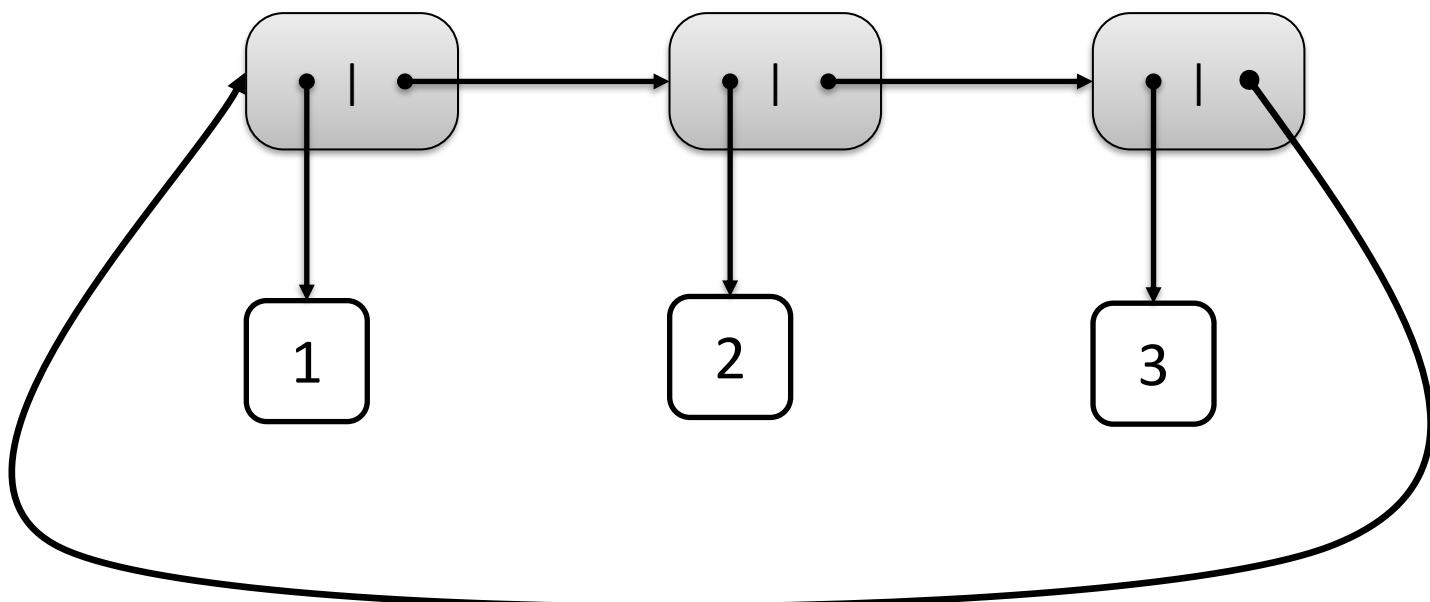
Delete



Queue

```
(define (make-q)
  (let ((front '()) (rear '()))
    (define (in x)
      (let ((new (list x)))
        (if (null? front)
            (begin (set! front new) (set! rear new))
            (begin (set-cdr! rear new) (set! rear new))))))
    (define (out)
      (let ((x (car front)))
        (set! front (cdr front))
        x))
    (define (dispatch name)
      (cond
        ((eq? name 'in) in)
        ((eq? name 'out) out)))
    dispatch))
```

Circular “lists”



Circular “lists”

```
(define (make-cyclic-list! ls)
  (define (cyc! xs)
    (if (null? (cdr xs))
        (begin (set-cdr! xs ls) ls)
        (cyc! (cdr xs)))
    ))
(cyc! ls))

(define week (make-cyclic-list!
  '(mon tue wed thu fri sat sun))))
```

Hash tables in Racket

There are many variants of hash tables

Create a hash table comparing with `equal?`
`(make-hash)`

Associate `v` with `key` in `hash`

`(hash-set! hash key v)`

`(hash-ref hash key [failure-result])`

`(hash-ref! hash key to-set)`

`hash-remove!`, `hash-update!`

Memoization

```
(define (memoize f)
  (let ((table (make-hash)))
    (lambda args
      (hash-ref! table
                 args
                 (lambda ()
                   (display "X")
                   (apply f args)))))))
```

Concurrency and Parallelism in Racket

- Thread (concurrency)
 - preempt each other without cooperation
 - share state: variables, function definitions, etc.
 - in Racket, they run on one OS thread
- Futures (parallelism)
 - evaluate an expression in parallel to the main program
 - block on operations that may not run safely in parallel
- Places (parallelism)
 - separate instances of scheme
 - communicate using message passing

Threads

Run on single OS thread

No speed-up

Waiting for slow/external event: I/O, sockets, etc.

Operations on threads

(thread thunk) **returns thread descriptor**

thread-suspend, thread-resume, kill-thread

Thread mailboxes

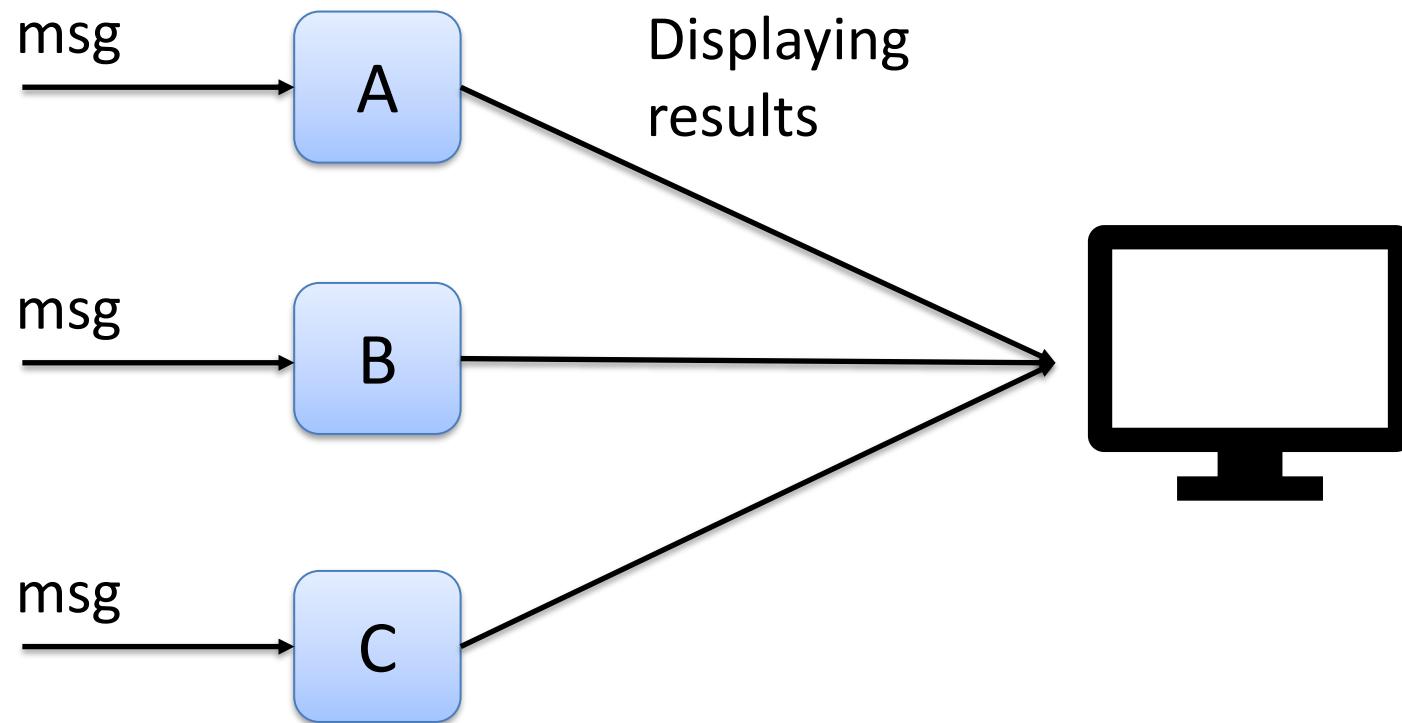
(thread-send th msg), (thread-receive)

Channels

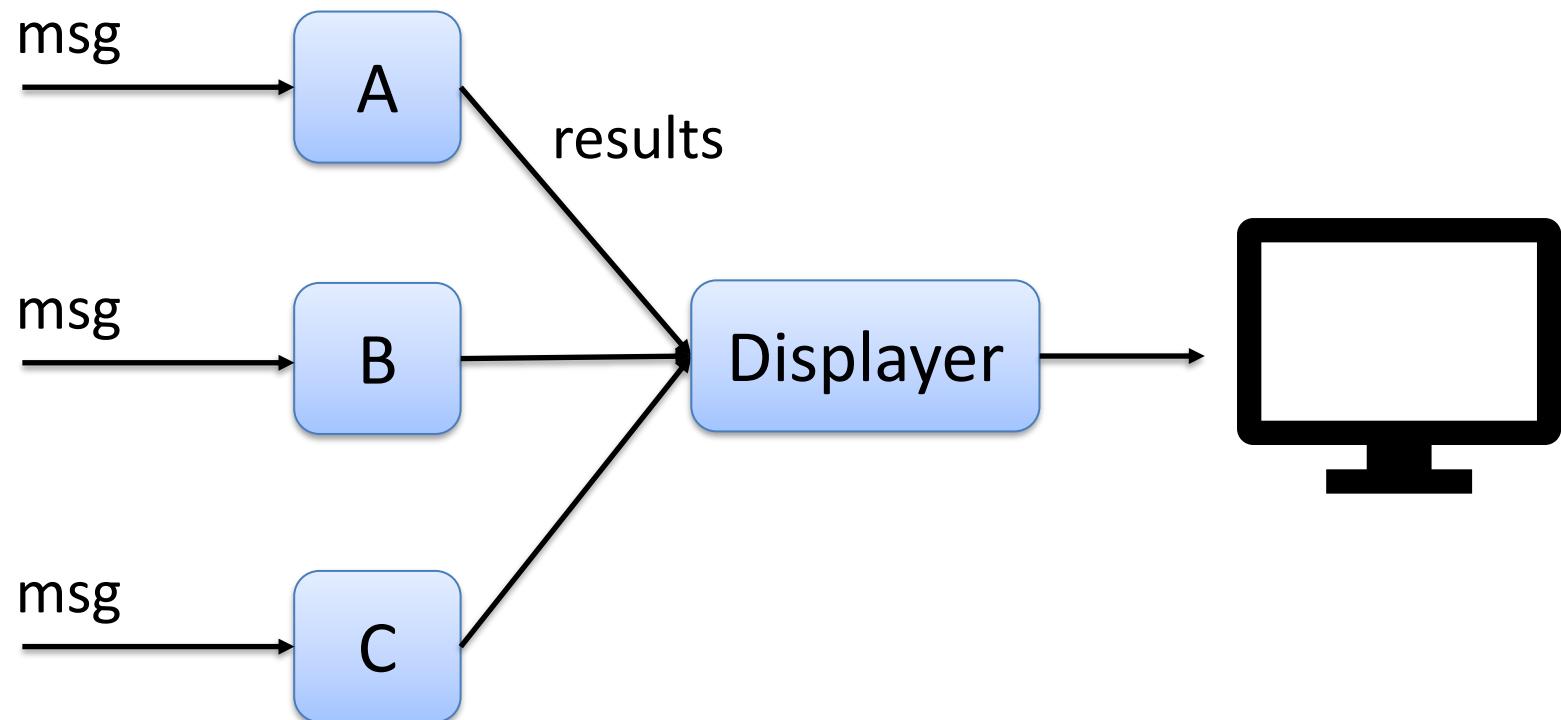
(make-channel), (channel-put ch v)

(channel-get ch), (channel-try-get ch)

Threads example



Threads example



Futures

(require racket/future)

(future thunk)

Starts evaluating an expression (given as thunk)

Blocks when an operation may not be safely executed

Returns a "future"

(touch future)

Finish evaluating the expression in the main thread

If the expression is already evaluated, return the result

As in *promise*, additional touches just return the result

Future map

Executes a given function on each element of a list in parallel and returns the results

```
(define (future-map f list)
  (let ((res
         (map (lambda (x)
                 (future (lambda () (f x))))
               list)))
    (map touch res)))
```

Futures can be visualized and analyzed using

```
(require future-visualizer)
(visualize-futures expr)
```

Home assignment 3

Genetic programming

Evolution inspired local search in structured data

Survival of the fittest!!!

Individual: program for the robot in the maze

Population: collection of the programs

New generation: selection, mutation, cross-over

Fitness function: see Home assignment 2

Summary

- We do not need to modify the state
- It breaks nice properties of FP
- It can sometimes be useful
 - random access in $O(1)$
 - objects
 - circular data structures
 - memoization
- Concurrency and parallelism



Functional Programming

Lecture 7: Lambda calculus

Viliam Lisý

Rostislav Horčík

Artificial Intelligence Center

Department of Computer Science

FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

xhorcik@fel.cvut.cz

Acknowledgement

Lecture based on:

Raúl Rojas: *A Tutorial Introduction to the Lambda Calculus*, FU Berlin, WS-97/98.

<https://arxiv.org/abs/1503.09060>

Link is also provided in courseware.

Lambda calculus

Theory developed for studying properties of effectively computable functions

Formal basis for functional programming

- as Turing machines for imperative programming

Smallest universal programming language

- function definition scheme
- variable substitution rule

Introduced by Alonzo Church in 1930s

Why to care?

- Understand that lambda and application is enough to build any program
 - without mutable state, assignment, define, etc.
 - useful for proving program properties
- Understand how numbers, conditions, recursion can be created in a purely functional way
- Think about programming yet a little differently
- Have a clue when someone mentions λ -calculus
- Understand that scheme syntax is not the worst

Syntax

A program in λ -calculus is an expression

$\langle \text{expression} \rangle := \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle$

$\langle \text{function} \rangle := \lambda \langle \text{name} \rangle. \langle \text{expression} \rangle$

$\langle \text{application} \rangle := \langle \text{expression} \rangle \langle \text{expression} \rangle$

Names, also called variables, will be x, y, z, \dots

By convention

$E_1 E_2 E_3 \dots E_n$ is interpreted as $(\dots ((E_1 E_2) E_3) \dots E_n)$

Free and bound variables

A variable in a body of a function is **bound** if it is under the scope of λ and **free** otherwise.

$\lambda x. xy, (\lambda x. x)(\lambda y. yx)$ - bold variables are free

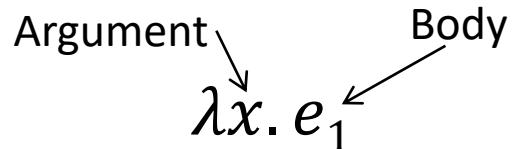
Bound variable names can be renamed anytime

$$\lambda x. x \equiv \lambda y. y \equiv \lambda z. z$$

An expression is **closed** if it has no free variables; otherwise it is **open**.

β -reduction

Lambda term represents a function:



In scheme (lambda (x) e₁)

Function can be applied to an expression e_2

$(\lambda x. e_1)e_2$ redex

It is applied by substituting free x 's in e_1 by e_2

$$(\lambda x. e_1)e_2 = [e_2/x]e_1$$

Examples of β -reduction

- $(\lambda x. x) (\lambda y. y) \equiv [(\lambda y. y)/x] x \equiv (\lambda y. y)$
- $(\lambda x. x x) (\lambda y. y) \equiv [(\lambda y. y)/x](x x)$
 $\qquad\qquad\qquad \equiv (\lambda y. y) (\lambda y. y)$
- $(\lambda x. x (\lambda x. x)) y \equiv [y/x](x (\lambda x. x))$
 $\qquad\qquad\qquad \equiv y (\lambda x. x)$

Name conflicts

Avoid name conflicts by renaming bound variables

1) do not let a substituent become bound

$$(\lambda x. (\lambda y. xy))y \text{ does } \mathbf{not} \text{ yield } \lambda y. yy$$
$$[y/x](\lambda z. xz) = \lambda z. yz$$

2) substitute only free occurrences of argument

$$\left(\lambda x. \left(\lambda y. \left(x(\lambda x. xy) \right) \right) \right) z \text{ is } \mathbf{not} \text{ } (\lambda y. (z(\lambda z. zy)))$$
$$[z/x](\lambda y. (x(\lambda x. xy))) = (\lambda y. (z(\lambda x. xy)))$$

Reduction orders

Expression may contain several redexes.

- Normal-order reduction: reduce left-most first
- Applicative-order reduction: reduce right-most first

Expression with no redex is in **normal form**.

Reduction process need not terminate!

$$(\lambda x. (x\ x))(\lambda x. (x\ x)) \equiv (\lambda x. (x\ x))(\lambda x. (x\ x))$$

Church-Rosser Theorem:

1. Normal forms are unique (independently of reduction order).
2. Normal order always finds normal form if it exists.

Non-naming of functions

Function in λ -calculus do not have names

We apply a function by writing its whole definition

We use capital letters and symbols to abbreviate this

These function names are not a part of λ -calculus

The identity function is usually abbreviated by I

$$I \equiv (\lambda x. x)$$

Conditionals

Lambda term of the form $\lambda x. \lambda y. e$ is abbreviated $\lambda xy. e$

$$T \equiv \lambda xy. x$$

$$F \equiv \lambda xy. y$$

The T and F functions directly serve as If

$$Tab = a$$

$$Fab = b$$

Logical operations

AND

$$\wedge \equiv \lambda xy. xy(\lambda uv. v) \equiv \lambda xy. xyF$$

OR

$$\vee \equiv \lambda xy. x(\lambda uv. u)y \equiv \lambda xy. xTy$$

Negation

$$\neg \equiv \lambda x. x(\lambda uv. v)(\lambda ab. a) \equiv \lambda x. xFT$$

$$\wedge TF \equiv (\lambda xy. xyF)TF \equiv TFF \equiv (\lambda ab. a)FF \equiv F$$

$$\wedge FT \equiv (\lambda xy. xyF)FT \equiv FTF \equiv (\lambda ab. b)TF \equiv F$$

Numbers

We define a "zero" and a successor function representing the next number

$$0 \equiv \lambda s. (\lambda z. z) \equiv \lambda s z. z$$

$$1 \equiv \lambda s z. s(z)$$

$$2 \equiv \lambda s z. s(s(z))$$

$$3 \equiv \lambda s z. s(s(s(z)))$$

Functional alternative of binary representation

Successor function

Increment a number by one

$$S \equiv \lambda wyx. y(wyx)$$

Increment zero to get one

$$\begin{aligned} S0 &\equiv (\lambda wyx. y(wyx))(\lambda sz. z) = \\ &\lambda yx. y((\lambda sz. z)yx) = \\ &\lambda yx. y((\lambda z. z)x) = \\ &\lambda yx. y(x) \equiv 1 \end{aligned}$$

Try: $S1, S2, \dots$

Addition

$x + y$ is applying the successor x times to y

Meaning of number n is just "apply the first argument n times to the second argument"

$$\lambda sz. s(\dots s(s(z)) \dots)$$

Therefore $2+3$ is just:

$$\begin{aligned} 2S3 &\equiv \\ (\lambda sz. s(sz))(\lambda wyx. y(wyx))(\lambda uv. u(u(v))) \\ &\equiv S(S3) \equiv S4 \equiv 5 \end{aligned}$$

Multiplication

We can multiply two numbers using

$$* \equiv (\lambda xyz. x(yz))$$

$$\begin{aligned} * 23 &\equiv (\lambda xyz. x(yz))23 = (\lambda z. 2(3z)) = \\ &(\lambda z. (\lambda xy. x(x(y))))(3z) = \\ &(\lambda z. (\lambda y. (3z)((3z)(y)))) = \\ &\left(\lambda z. (\lambda y. \left(z \left(z \left(z((3z)(y)) \right) \right) \right)) \right) = \\ &\left(\lambda zy. \left(z \left(z \left(z(z(z(y)))) \right) \right) \right) = 6 \end{aligned}$$

Conditional tests

Test if a given number is the 0

$$Z \equiv \lambda x. xF \neg F$$

$$Z0 \equiv$$

$$(\lambda x. xF \neg F)0 = 0F \neg F = \neg F = T$$

$$ZN \equiv$$

$$\begin{aligned} & (\lambda x. xF \neg F)N = NF \neg F \\ & = F(\dots F(\neg) \dots)F = IF = F \end{aligned}$$

Pairs

The pair $[a, b]$ can be represented as
 $(\lambda z. zab)$

We can extract the first element of the pair by
 $(\lambda z. zab)T$

and the second element by
 $(\lambda z. zab)F$

Predecessor

We want to create a function, which applied N times to something returns $N - 1$

The function modifies a pair (x, y) to $(x + 1, x)$

$$\Phi \equiv (\lambda p z. z(S(pT)))(pT))$$

Calling Φ on $[0,0]$ N times yields $[N, N - 1]$

$$\Phi[0,0] = [1,0] \quad \Phi[1,0] = [2,1] \quad \dots$$

Finally, we take the second number in the pair

The predecessor function is

$$P \equiv \lambda n. n\Phi(\lambda z. z00)F$$

Note than the predecessor of 0 is 0

Equality and inequality

$x \geq y$ can be represented by

$$G \equiv (\lambda xy. Z(xPy))$$

Equality is then defined based on the above as

$$\begin{aligned} E &\equiv \lambda xy. \wedge (Gxy)(Gyx) \\ &\equiv (\lambda xy. \wedge (Z(xPy))(Z(yPx))) \end{aligned}$$

Other inequalities can be defined analogically using the previously defined logical operations

Recursion

Can we create recursion without function names?

$$Y \equiv (\lambda y. (\lambda x. y(xx))) (\lambda x. y(xx)))$$

Now apply Y to some other function R

$$YR \equiv (\lambda x. R(xx)) (\lambda x. R(xx)) \equiv$$

$$R((\lambda x. R(xx)) (\lambda x. R(xx)))) \equiv$$

$$R(YR) \equiv R(R(YR)) \equiv R(\dots (R(YR) \dots))$$

Function R is called with YR as the first argument

Recursion

We can recursively sum up first n integers as

$$\sum_{i=0}^n i = n + \sum_{i=0}^{n-1} i$$

In scheme

```
(define (sum-to n)
  (if (= n 0) 0
      (+ n (sum-to (- n 1))))))
```

A corresponding recursive function is

$$R \equiv (\lambda rn. Zn0(nS(r(Pn))))$$

Recursion

$$R \equiv (\lambda rn. Zn0(nS(r(Pn))))$$

$$YR3 \equiv$$

$$R(YR)3 \equiv Z30\left(3S(YR(P3))\right) \equiv$$

$$F0\left(3S(YR(P3))\right) \equiv \left(3S(YR(P3))\right) \equiv$$

$$3S(YR2) \equiv 3S(2S(YR1)) \equiv 3S2S1S0 \equiv 6$$

Turing completeness

Turing machine

- a standard formal model of computation
- B4B01JAG Jazyky, automaty a gramatiky
- what can be solved by TM, can be solved by standard computers

A programming language Turing complete, if it can solve all problems solvable by TM

Lambda calculus is Turing complete

Summary

- Lambda calculus is formal bases of FP
- Simplest universal programming language
- Everything using lambda and application
 - conditions
 - numbers
 - pairs
 - recursion

Lecture 8: Introduction to Haskell

Viliam Lisý

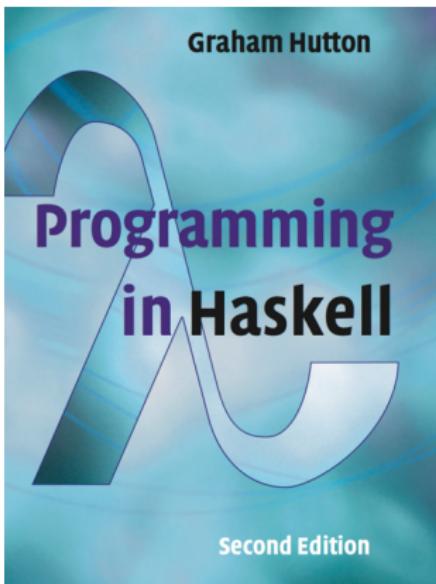
Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

Acknowledgement

Slides for next few lectures based on slides for the book:



Why Haskell?

- Purely functional language
 - promotes understanding the paradigm
- Rich syntactic sugar (contrast to Lisp)
- Rich system of types (contrast to Lisp)
- Most popular purely functional language
- Fast prototyping of complex systems
- Active user community
 - Haskell platform, packages, search
 - <https://haskell.org/>

Main properties

- Purely functional language
 - necessary exceptions (IO) wrapped as monads
- Statically typed
 - types are derived and checked at compile time
 - types can be automatically inferred
 - have a crucial role in controlling flow of the program
- Lazy
 - function argument evaluated only when needed
 - almost everything is initially a thunk

Haskell is a standardization of ideas in over a dozen of pre-existing (often proprietary) lazy functional languages

- Haskell 1.0 to 1.4
 - standardization efforts since 1990
- Haskell 98
 - first stable standard
- Haskell 2010
 - minor changes based on best practices in existing implementations
 - integration with other programming languages
 - hierarchical module names
 - pattern guards

Glasgow Haskell Compiler (GHC)

- the leading implementation of Haskell
- comprises a compiler and interpreter
- written in Haskell
- is freely available from: www.haskell.org/platform

Haskell Users Gofer System (Hugs)

- small and portable interpreter
- Windows version with simple GUI called WinHugs
- no longer in development

Starting GHCi

The interpreter can be started from the terminal command prompt \$ by simply typing ghci:

```
$ ghci  
GHCi, version X: http://www.haskell.org/ghc/ :? for help  
Prelude>
```

The GHCi prompt ">" means that the interpreter is now ready to evaluate an expression.

Basic interaction

- REPL interaction as in scheme
- Common infix syntax
- Space denotes function application
- Infix operators have priorities
 - function application is first
 - otherwise use brackets
- Left associativity for functions (as in lambda calculus)
- Up arrow recalls the last entered expression

The basic data structure

```
[1,2,3,4,5]
```

```
[1..10]
```

```
[1,3..]
```

- Build by "cons" operator : , ended by the empty list []
- All elements must be of the **same type**
- Includes all basic functions
 - take, length, reverse, ++, head, tail
- In addition, you can index by !!

Special commands

Commands to the interpreter start with ":"

- `:?` for help
- `:load <module>`
- `:reload`
- `:quit`

Can be abbreviated to the first letter

New functions are defined within a script

- Text file comprising a sequence of definitions
- Usually have a .hs suffix
- Can be loaded by

```
$ ghci <filename>
> :load <filename>
```

Defining functions

```
fact1 1 = 1
fact1 n = n * fact1 (n-1)
```

```
fact2 n = product [1..n]
```

```
power n 0 = 1
power n k = n * power n (k-1)
```

Comments

```
-- Comment until the end of the line
```

```
{-  
  A long comment  
  over multiple  
  lines.  
-}
```

Naming requirements and conventions

Function and **argument** names must begin with a lower-case letter. For example:

myFun

fun1

arg_2

x'

By convention, **list arguments** usually have an s suffix on their name. For example:

xs

ns

nss

Names with only special symbols are **infix operators**:

++++

+/+

%-

Infix operators

Can be defined in prefix notations:

```
x +/+ y = 2*x + y
```

A prefix function turns infix by ‘ ‘ and infix turns prefix by ()
‘mod’, ‘elem’, (+), (+/+)

Precedence/asociativity of infix operators set by

```
infixr <0-9> <name>
infixl <0-9> <name>
infix <0-9> <name>
```

Information about associativity, precedence, and much else

```
> :info
```

Interesting infix operators

- . \$ unary -

Pattern matching

The first LHS that matches the function call is executed

```
not False = True  
not True  = False
```

not maps False to True, and True to False

Pattern matching

Functions can often be defined in many different ways using pattern matching.

```
True && True = True
True && False = False
False && True = False
False && False = False
```

The underscore symbol `_` is a **wildcard pattern** that matches any argument value.

```
True && True = True
_ && _ = False
```

A more efficient definition does not evaluate the second argument:

```
True && b = b
False && _ = False
```

Pattern matching

The order of the definitions matters

```
_ && _ = False  
True && True = True
```

Patterns may **not repeat variables**, due to efficiency. The following gives an error:

```
b && b = b  
_ && _ = False
```

List patterns

Functions on lists can be defined using `x:xs` patterns

```
head (x:_ ) = x  
tail (_ :xs) = xs
```

We will see later it works similarly for other composite data types.
`x:xs` patterns only match **non-empty** lists:

```
> head []  
*** Exception: empty list
```

`x:xs` patterns must be **parenthesised**, because application has priority over `(:)`. The following definition gives an error:

```
head x:_ = x
```

Tuples

Tuples are **fixed length** sequences of elements of **arbitrary types**

```
(1,2)  
('a','b')  
(1,2,'c',False)
```

Their element can be accessed by pattern matching

```
first  (x,_,_) = x  
second (_ ,x,_) = y  
third   (_ ,_ ,x) = x
```

Pattern matching can be nested

```
f (1, (x:xs), 'a', (2,y)) = x:y:xs
```

As pattern

Sometimes it is useful to repeat larger parts of patterns from RHS on the LHS

```
copyfirst x:xs = x:x:xs
```

A part of the pattern can be assigned a name

```
copyfirst s@(x:xs) = x:s
```

Let / where

```
dist1 (x1,y1) (x2,y2) =  
    let d1 = x1-x2  
        d2 = y1-y2  
    in sqrt(d1^2+d2^2)
```

```
dist2 (x1,y1) (x2,y2) = sqrt(d1^2+d2^2)  
where d1 = x1-x2  
      d2 = y1-y2
```

The layout rule

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c where  
    b = 1  
    c = 2
```

means

```
a = b + c where {b=1; c=2}
```

The layout rule

Keywords (such as `where`, `let`, etc.) start a block:

- The first word after the keyword defines the **pivot column**.
- Lines exactly on the pivot define a new entry in the block.
- Start a line after the pivot to continue the previous lines.
- Start a line before the pivot to end the block.

Conditional expressions

```
abs n = if n >= 0 then n else -n
```

Conditional expressions can be nested:

```
signum n = if n < 0 then -1 else
            if n == 0 then 0 else 1
```

The must **always** have an else branch.

Guarded equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n >= 0      = n
      | otherwise = -n
```

Definitions with multiple conditions are then easier to read:

```
signum n | n < 0      = -1
          | n == 0      = 0
          | otherwise = 1
```

otherwise is defined in the prelude by otherwise = True

Set comprehensions

In mathematics, the **comprehension** notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1 \dots 5\}\}$$

The set $\{1, 4, 9, 16, 25\}$ of all numbers x^2 such that x is an element of the set $\{1 \dots 5\}$.

List comprehensions

In Haskell, a similar comprehension notation can be used to construct new **lists** from old lists.

```
[x^2 | x <- [1..5]]
```

`x <- [1..5]` is called a **generator**.

Comprehensions can have **multiple** generators

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]  
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Generator order

Changing the **order** of the generators changes the order of the elements in the final list:

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]  
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Multiple generators are like nested loops, with later generators as more deeply **nested loops** whose variables change value more frequently.

Dependent generator

Later generators can **depend** on the variables that are introduced by earlier generators.

```
[ $(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]$ ]
```

Using a dependant generator we can define the library function that **concatenates** a list of lists:

```
concat xs = [x | xs  $\leftarrow$  xs, x  $\leftarrow$  xs]
```

Infinite generator

Generators can be infinite (almost everything is lazy)

```
[x^2 | x <- [1..]]
```

The order then matters even more

```
[x^y | x <- [1..], y <- [1,2]]
```

vs.

```
[x^y | y <- [1,2], x <- [1..]]
```

Guards

List comprehensions can use **guards** to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

Using a guard we can define a function that maps a positive integer to its list of **factors**:

```
factors n = [x | x <- [1..n], mod n x == 0]
```

Example: primes

A prime's only factors are 1 and itself

```
prime n = factors n == [1,n]
```

List of all primes

```
[x | x <- [2..], prime x]
```

Example: quicksort

```
quicksort [] = []
quicksort (x:xs) = quicksort [a | a <- xs, a < x ]
                  ++ [x] ++
                  quicksort [a | a <- xs, a >= x]
```

```
quicksort [] = []
quicksort (x:xs) = quicksort smalls ++ [x] ++ quicksort larges
  where
    smalls = [a | a <- xs, a <= x]
    larges = [b | b <- xs, b > x ]
```

Haskell is the unified standard for FP

- purely functional, lazy, statically typed

It has rich 2D syntax to write compactly

Functions are defined by pattern matching

Lecture 9: Haskell Types

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

What is a Type?

A **type** is a name for a collection of related values (e.g., basic, composed, functions, etc.). For example, in Haskell the basic type

Bool

contains the two logical values:

True

False

Type errors

Applying a function to one or more arguments of the wrong type is called a **type error**.

```
Prelude> 1 + False
```

```
... error ...
```

1 is a number and False is a logical value, but + requires two numbers.

Errors prevented by types

- Applying a function to the wrong number of arguments
- Permuting the arguments of a function
- Forgetting the application of a (conversion) function

Types in functional languages

Java types: A function defined to return a string may return

- a String
- null
- an exception
- in addition to that, it can also return a "modified state of the world" (e.g. it can print a line of text to the console)
- not return at all (e.g. run into an infinite loop)

Functional languages can be (and often are) completely rigorous about types

- If it says it returns a String, it does that and only that
- No side effects!

Types in functional languages often more expressive

- They allow representing more complex properties of programs

- Types are checked without executing a function
 - E.g., type error in an unreachable branch is still detected
 - infinite stream can have a clear type
- All type errors are found at compile time
 - + **safer**: if it compiles, there is not type mismatch
 - + **faster**: no need for type checks at run time
 - + **clearer?**: can serve as documentation / specification
 - sometimes more verbose code
 - slower compilation
 - more complex compiler implementation

Types in Haskell

If evaluating an expression e would produce a value of type t , then
 e has type t , written

$e :: t$

Every well-formed expression has a type automatically calculated at
compile time using a process called **type inference**.

In GHCi, the `:type` command calculates the type of an expression,
without evaluating it:

```
> :type not False
not False :: Bool
```

Haskell has a number of **basic types**, including:

Bool	logical values
Char	single characters
String	strings of characters
Int	fixed-precision integers
Integer	arbitrary-precision integers
Float	floating-point numbers

Tuple types

A **tuple** is a sequence of values of possibly **different** types:

```
(False,'a',True) :: (Bool,Char,Bool)  
(False,True) :: (Bool,Bool)
```

The type of n-tuples whose i-th element has type t_i is
 (t_1, t_2, \dots, t_n)

- The type of a tuple encodes its size
- The type of the components is unrestricted

List types

A **list** is a sequence of values of the **same** type:

```
[False, True, False] :: [Bool]  
['a', 'b', 'c', 'd']   :: [Char]
```

In general, for any type a

[a] is the type of lists with elements of type a

- The type of a list says nothing about its length
- The type of the elements can be arbitrary (not only basic)

Function types

Types of functions are denoted using \rightarrow

```
add :: (Int, Int) -> Int
```

```
add (x, y) = x + y
```

```
zeroto :: Int -> [Int]
```

```
zeroto n = [0..n]
```

- The argument and result types are unrestricted
- It is encouraged to write types above each function

Curried functions

Functions with multiple arguments are also possible by returning **functions as results**:

```
add :: (Int, Int) -> Int  
add (x, y) = x+y
```

```
add' :: Int -> (Int -> Int)  
add' x y = x+y
```

add and add' produce the same final result, but add take arguments in a different form

Curried functions

It transparently works for multiple arguments

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```

`mult` takes an integer `x` and returns a function `mult x`, which in turn takes an integer `y` and returns a function `mult x y`, which finally takes an integer `z` and returns the result `x*y*z`.

Partial function application

Curried functions are more flexible than functions on tuples, because useful functions can often be made by **partially applying** a curried function.

```
add' 1 :: Int -> Int  
  
take 5 :: [Int] -> [Int]  
  
drop 5 :: [Int] -> [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two conventions are adopted:

- 1) The arrow operator \rightarrow associates to the **right**.

`Int → Int → Int → Int`

means `Int → (Int → (Int → Int))`

- 2) As a consequence, it is then natural for function application to associate to the **left**.

`mult x y z` means `((mult x) y) z`

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic functions

A function is called **polymorphic** (of many forms) if its type contains type variables.

```
length :: [a] -> Int
```

Type variables can be instantiated to different types in different circumstances

```
> length [False,True]      -- a = Bool  
2  
> length [1,2,3,4]        -- a = Int  
4
```

Polymorphic functions

Many of the functions defined in the standard prelude are polymorphic.

```
fst :: (a,b) -> a
```

```
head :: [a] -> a
```

```
take :: Int -> [a] -> [a]
```

```
zip :: [a] -> [b] -> [(a,b)]
```

```
id :: a -> a
```

Overloaded functions

A polymorphic function is called **overloaded** if its type contains one or more class constraints.

```
(+) :: Num a => a -> a -> a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.

Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2           -- a = Int  
3  
> 1.0 + 2.0      -- a = Float  
3.0  
> 'a' + 'b'      -- Char is not numeric  
ERROR
```

Type classes

Haskell has a number of type classes, including:

Num	Numeric types
Eq	Equality types
Ord	Ordered types

For example, you can verify by calling :type:

```
(+)  :: Num a => a -> a -> a  
  
(==) :: Eq a => a -> a -> Bool  
  
(<)  :: Ord a => a -> a -> Bool
```

- When defining a new function in Haskell, it is useful to begin by writing down its type;
- Within a script, it is good practice to state the type of every new function defined;
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Type declarations

In Haskell, a new **name** for an **existing type** can be defined using a **type declaration**.

```
type String = [Char]
```

Type declarations make other types easier to read.

```
type Pos = (Int, Int)
```

```
left :: Pos -> Pos
```

```
left (x,y) = (x-1,y)
```

Parametrized types

Like function definitions, type declarations can also have **parameters**. With

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int -> Int
mult (m,n) = m*n

copy :: a -> Pair a
copy x = (x,x)
```

Type declarations

Type declarations can be nested:

```
type Pos = (Int, Int)  
  
type Trans = Pos -> Pos
```



However, they cannot be recursive:

```
type Tree = (Int, [Tree])
```



Data declarations

Define a completely new type by specifying its values

```
data Bool = False | True
```

Values False and True are the **constructors** for the type

Type and constructor names begin with a **capital letter**

Data declarations

Values of new types can be used in the same ways as those of built-in types. Given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes, No, Unknown]

flip :: Answer -> Answer
flip Yes      = No
flip No       = Yes
flip Unknown = Unknown
```

Parametric constructors

The constructors in a data declaration can have parameters. Given

```
data Shape = Circle Float | Rect Float Float
```

we can define:

```
square :: Float -> Shape
square n = Rect n n
```

Circle and Rect can be viewed as **functions** that construct values of type Shape

New composed data types can be decomposed by **pattern matching**

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Parametric data declarations

One of the most common Haskell types

```
data Maybe a = Nothing | Just a
```

allows defining safe operations.

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive types

New types can be declared in terms of themselves. That is, types can be **recursive**. (just not with type keyword)

```
data Nat = Zero | Succ Nat
```

A value of type Nat is either Zero, or Succ n where n :: Nat.
Nat contains infinite sequence of values:

Zero

Succ Zero

Succ (Succ Zero)

...

Recursive types

We can use pattern matching and recursion to translate from Int to Nat and back.

```
nat2int :: Nat -> Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Recursive types

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero      n = n
add (Succ m) n = Succ (add m n)
```

Records

Purely positional data declarations are impractical with a large number of fields. Therefore, the fields can be named:

```
data Person = Person { firstName :: String,  
                      lastName :: String,  
                      age :: Int,  
                      height :: Float,  
                      phone :: String,  
                      address :: String}
```

This allows to define records in arbitrary order

```
defaultPerson = Person {lastName="Smith",  
                       firstName="John",...}
```

And access fields using automatically generated functions, e.g.,

```
firstName :: Person -> String
```

Example: Arithmetic expressions

Recursive types can represent tree structures, such as **expressions** from numbers, plus, multiplication.

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

$$1+2*3$$

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Example: Arithmetic expressions

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr -> Int
size (Val n)    = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr -> Int
eval (Val n)    = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Homework assignment 4

- Everything has a type known in compile time
 - basic values
 - functions
 - data structures
- Types are key for data structures in Haskell
- **Algebraic types**
 - compose complex types from simpler as products and unions
- Types can be instances of classes
 - polymorphic functions

Lecture 10: Type Classes and Miscellaneous

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

Example: Arithmetic expressions

Recursive types can represent tree structures, such as **expressions** from numbers, plus, multiplication.

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

$$1+2*3$$

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Example: Arithmetic expressions

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr -> Int
size (Val n)    = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr -> Int
eval (Val n)    = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Type Classes

Collection of types that can be used with the same functions Eq, Ord, Show. Functions required by a class can be accessed by

```
:info <classname>.
```

```
> :info Eq

class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Functions can often be implemented based on other. Only minimal complete definition (one of the above) is required.

Show Class

A class values convertible to a readable string

```
class Show a where
    showsPrec :: Int -> a -> ShowS
    show :: a -> String
    showList :: [a] -> ShowS
```

```
type ShowS = String -> String
```

This allows constant-time concatenation of results using function composition (optimization)

Minimal complete definition: `showsPrec | show`

Instance of a Class

A new instance can be added to a class by

```
instance Show Nat where
    show n = "N" ++ show (nat2int n)
```

```
instance Show Expr where
    show (Val n) = show n
    show (Add e1 e2) = "(+ " ++ show e1 ++ " "
                           ++ show e2 ++ ")"
    show (Mul e1 e2) = "(* " ++ show e1 ++ " "
                           ++ show e2 ++ ")"
```

Class Contexts

Remember the definition

```
data Maybe a = Nothing | Just a
```

To make Maybe an instance of Eq, a has to be in Eq

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    (Just x) == (Just x') = x == x'
```

Deriving Classes

Obvious definition of instances are automated

```
data Shape = Circle Float
           | Rect Float Float
deriving (Show, Eq)
```

The implemented function bodies determine the minimum required functions

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Functor Class

Class of structures you can map over

```
class Mapable f where
    mmap :: (a -> b) -> f a -> f b
```

```
instance Mapable [] where
    mmap = map
```

```
instance Mapable Maybe where
    mmap f (Just x) = Just (f x)
    mmap f Nothing = Nothing
```

Types of types and type constructors

* A specific type

* \rightarrow * A type that given a type creates a type

Constraint A constructor of a type constraint

:k

Types Summary

- Everything has a type known in compile time
 - basic values
 - functions
 - data structures
- Types are key for data structures in Haskell
- Types can be instances of classes
 - overloaded functions
- "Types" of types are kinds

Higher Order Functions

The same functions as in scheme are available

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

```
map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
```

Foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

$$\begin{aligned} \text{sum } [1, 2, 3] &= \\ \text{foldr } (+) \ 0 \ [1, 2, 3] &= \\ \text{foldr } (+) \ 0 \ (1 : (2 : (3 : []))) &= \\ 1 + (2 + (3 + 0)) &= \\ 6 & \end{aligned}$$

`fold fn el list` can be interpreted as:
replace each `(:)` by ‘fn’ and `[]` by `el`.

Lambda Expressions

Functions can be constructed without naming the functions by using **lambda expressions**.

$$\lambda x \rightarrow x + x$$

As in scheme,

$$\text{add } x \ y = x + y$$

means

$$\text{add} = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

We also have the automated currying

$$\text{add} = \lambda x \ y \rightarrow x + y$$

Lambda Expressions

We can use lambda expressions and local functions interchangeably

```
odds n = map f [0..n-1]
  where
    f x = x*2 + 1
```

can be simplified to

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

The earlier may be better if the local function has a natural name

Operator Sections

An **infix** operator can be converted into a curried **prefix** function by using parentheses.

```
> (+) 1 2  
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

```
> (1+) 2  
3  
> (+2) 1  
3
```

If \oplus is an operator then (\oplus) , $(x\oplus)$ and $(\oplus y)$ are called sections.

Custom Data Constructors

Begin with :

:#, :+, :::

```
infixr :+
data MList a = Empty | a :+ MList a deriving Show
```

Haskell program is a collection of modules

- name spaces, abstract data declarations
- module names start with upper-cased character
- filenames must match module names in GHC
- `module <name> (<exported>, <symbols>) where`
- without exported symbols, everything is exported
- data constructors exported with type name
- `Tree(Leaf,Branch)`, can be abbreviated to `Tree(..)`

Example Module

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)           = [x]
fringe (Branch left right) =
                           fringe left ++ fringe right
```

Importing Modules

Imports must be at the beginning of a module

Prelude module is loaded by default

We can choose names to import and hide

```
import Tree
import Tree hiding (tree1)
import Tree (tree1, fringe)
import qualified Tree as T hiding (tree1)
:m + Tree
```

Advanced Pattern Matching

Data constructors can be matched nested

(1, (x:xs), 'a', (2, Just y:ys))

but not x:x:xs

Top-down, left-right

Matching can succeed, fail, diverge

Refutable patterns: [], Tree x l r

Irrefutable patterns: _, x, a, ~(x:xs).

Lazy pattern \sim pat is irrefutable (always matches)

- The variable pat is bound only when used
- $\sim(x:xs)$ on LHS is equivalent to using head/tail on RHS
- $\sim(x,y)$ on LHS is equivalent to using fst/snd on RHS

$(\backslash \sim(a,b) \rightarrow 1) \bot$

A new instance can be added to a class by

Case Expressions

```
f p11 ... p1k = e1  
...  
f pn1 ... pnk = en
```

where each p_{ij} is a pattern, is semantically equivalent to:

```
f x1 x2 ... xk = case (x1, ..., xk) of  
  (p11, ..., p1k ) -> e1  
  ...  
  (pn1, ..., pnk ) -> en
```

Pattern Matching Divergence

Assume the infinite recursion `bot = bot`

Pattern matching diverges if it tries to match `bot`

Order of definitions influences pattern matching failure

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take1 (n-1) xs
```

- Type and type classes essential for Haskell
- Unnecessary, but pleasant Haskell features
 - higher order functions
 - lambda functions
 - infix operator sections
 - modules

Lecture 11: Haskell IO

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

May, 2020

Haskell is Purely Functional

- Functions have no side effects
 - outputs depend only on inputs
 - calling function with same arguments multiple times produces the same output
 - order of executing independent functions is arbitrary
 - **Haskell functions** cannot change files or print
- Pseudo-functions like rand() or getchar() in C
 - return different value each call
 - change files, network, content of the screen

Haskell is Purely Functional

- Optimizations are pure function transformations
 - rearrange calls, cache results
 - omits calling functions, unless their results are used (lazy)
 - might automatically parallelize (but granularity)
 - easier to prove correctness of optimizations
- Optimization in C must be more conservative
 - We want to keep purely functional nature
 - But we want to be able to interact, change files, etc.

IO Actions

- Haskell separates the part of the program with side effects using values of special types
- $(\text{IO } a)$ is an action, which when executed produces a value of type a

```
getChar :: IO Char  
getLine :: IO String  
putStrLn :: String -> IO ()
```

- IO actions can be passed from function to function, but are not executed in standard evaluation

Haskell program executes an action returned by function `main` in module `Main`

```
main :: IO ()  
main = putStrLn "Hello, World!"
```

Running the program

```
$ ghc <filename.hs>; ./<filename>  
$ runghc <filename.hs>
```

Sequencing actions

In order to call multiple functions, they need to provide arguments for some other function

$$g(f_1, f_2, \dots, f_n)$$

In pure functional programming

- f_i can be called in an arbitrary order
- are called only when we need the return value

When do we need the return value of `putStrLn`?

Combining actions

```
(>>) :: IO a -> IO b -> IO b  
infixl 1 >>
```

$(x >> y)$ is the action that performs x , dropping the result, then performs y and returns its result.

```
main = putStrLn "Hello" >> putStrLn "World"
```

Combining actions: bind

```
(>>=) :: IO a -> (a -> IO b) -> IO b  
infixl 1 >>=
```

`x >>= f` is the action that first performs `x`, passes its result to `f`, which then computes a second action to be performed.

```
main = putStrLn "Hello, what is your name?"  
      >> getLine  
      >>= \n -> putStrLn ("Hello, " ++ n ++ "!")
```

```
x >> y = x >>= _ -> y
```

Combining actions: return

```
return :: a -> IO a
```

Transforms a value to IO action.

Used, e.g., to define the return value of a composed action, or

```
main :: IO ()  
main = return "Viliam" >>= \name  
      -> putStrLn ("Hello, " ++ name ++ "!")
```

Did we solve the problem?

There is no function

```
unsafe :: IO a -> a
```

hence all values related to side effects are "in" IO.

Everything outside IO is safe for all optimizations.

IO can be seen as

- a flag for values that came from functions with side effects
- a container for separating unsafe operations

IO is a special case of generally useful pattern

```
class Applicative m => Monad (m :: * -> *) where
  (=>=) :: m a -> (a -> m b) -> m b
  (=>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

Based on category theory

Way of meaningfully sequencing computations

- ① Creating a (separated) boxed value
- ② Creating functions for modifying them within the boxes

do Notation

Using monads leads to long sequences of operations chained by operators `>>`, `>>=`

```
main = putStrLn "Hello, what is your name?" >>
        getLine >>= \name ->
        putStrLn ("Hello, " ++ name ++ "!")
```

Do notation just makes these sequences more readable
(it is rewritten to monad operators before compilation)

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

do Notation

do is a syntax block, such as where and let

- action on a separate line gets executed
- `v <- x` runs action `x` and binds the result to `v`
- `let a = b` defines `a` to be the same as `b` until the end of the block (no need for `in`)

Derived Primitives

Creating more complex IO actions from simpler

```
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then
                return []
            else
                do xs <- getLine
                return (x:xs)
```

Derived Primitives

The same without the do notation

```
getLine2 :: IO String
getLine2 = getChar >>= \x
    -> if x == '\n' then
        return []
    else getLine2 >>= \xs
        -> return (x:xs)
```

Derived Primitives

Writing a string to the screen:

```
putStr :: String -> IO ()  
putStr []     = return ()  
putStr (x:xs) = do putChar x  
                  putStr xs
```

Writing a string and moving to a new line:

```
putStrLn :: String -> IO ()  
putStrLn xs = do putStr xs  
                  putChar '\n'
```

IO Actions as Values

IO actions cannot be executed outside of IO

They can still be used as any other values

- return them from functions
- add them to lists

```
ioActions :: [IO ()]
ioActions = [print "Hello!",
            putStrLn "just kidding",
            getChar >> return ()]
```

Combining a list of actions

```
sequence_ :: [IO a] -> IO ()  
sequence_ [] = return ()  
sequence_ (x:xs) = do x  
                      sequence_ xs
```

```
main = sequence_ ioActions
```

Consider the following version of **hangman**:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess
- The game ends when the guess is correct.

Hangman

We adopt a **top down** approach to implementing hangman in Haskell, starting as follows:

```
hangman :: IO ()  
hangman = do putStrLn "Think of a word: "  
            word <- sgetLine  
            putStrLn "Try to guess it:"  
            play word
```

Hangman

The action `sgetLine` reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine = do x <- getCh
             if x == '\n' then
               do putStrLn x
                  return []
             else
               do putStrLn '-'
                  xs <- sgetLine
                  return (x:xs)
```

Hangman

The action `getCh` reads a single character from the keyboard, without echoing it to the screen:

```
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False
          x <- getChar
          hSetEcho stdin True
          return x
```

Hangman

The function `play` is the main loop, which requests and processes guesses until the game ends.

```
play :: String -> IO ()  
play word =  
    do putStrLn "? "  
        guess <- getLine  
        if guess == word then  
            putStrLn "You got it!"  
        else  
            do putStrLn (match word guess)  
            play word
```

Hangman

The function `match` indicates which characters in one string occur in a second string.

For example:

```
> match "haskell" "pascal"  
"-as--ll"
```

```
match :: String -> String -> String  
match xs ys =  
  [if x `elem` ys then x else '-' | x <- xs]
```

Assignment 5

- Haskell IO is separated using IO actions
 - can be executed and cause side effects
 - can be used as values in Haskell functions
 - are a monad
- Monads are general constructions, which
 - define special operators `>>`, `>>=`, `return`
 - are “containers” that often hold data
 - can be used by do notation
- We made a complete executable program in Haskell

Lecture 12: Haskell Monads

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

May, 2020

Maybe

Remember the definition

```
data Maybe a = Nothing | Just a deriving (Eq, Show)
```

Nice working with partial functions:

```
g :: Int -> Maybe Int
g 0 = Nothing
g x = Just (div 10 x)

sq :: Maybe Int -> Maybe Int
sq Nothing = Nothing
sq (Just x) = Just (x ^ 2)
```

Not really, it is a pain!

Maybe Functor

Functor: Class of structures you can map over

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

```
($) :: (a -> b) -> a -> b
```

Maybe Applicative Functor

Functors work well for unary functions, what about binary?

```
> (g 3) + (g 3)
> fmap (+) (f 3)
>:t fmap (+) (f 3)
```

No way to apply the function without pattern matching `Just`.
We want a general approach applicable for any functor.

```
class Functor f => Applicative (f :: * -> *) where
    pure :: a -> f a
    (<$>) :: f (a -> b) -> f a -> f b
```

Allows any number of arguments through currying.

```
> (*) <$> (Just 2) <*> ((+) <$> (f 3)) <*> (f 3))
```

Monad

Applicative does not constraint the order of execution. Syntax of applicative functors may not be intuitive for everyone.

```
class Applicative m => Monad (m :: * -> *) where
  (=>=) :: m a -> (a -> m b) -> m b
  (=>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

Strict generalization of Functor and Applicative (since 2014).

```
fmap f xs = xs >= (\x -> return (f x))

pure = return
fs <*> as = do f <- fs
              a <- as
              return (f a)
```

do Notation

Using monads leads to long sequences of operations chained by operators `>>`, `>>=`

```
main = putStrLn "Hello, what is your name?" >>
        getLine >>= \name ->
        putStrLn ("Hello, " ++ name ++ "!")
```

Do notation just makes these sequences more readable
(it is rewritten to monad operators before compilation)

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

Maybe Monad

```
instance Monad Maybe where
    (Just x) >>= k      = k x
    Nothing   >>= _      = Nothing

    return x  = Just x
```

Maybe Monad

Since it is a monad, we can use the do notation:

```
h :: Int -> Maybe Int
h x = do u <- g x
         v <- g 5
         return (u+v)
```

Exception Handling

Exceptions in Haskell are represented by special types

such as Maybe, Either

Explicit handling of errors makes code hard to read

the special values of the types must be handled everywhere

```
import qualified Data.Map as M

lookUp :: Char -> Either String Int
lookUp name = case M.lookup name vars of
  Just x -> Right x
  Nothing -> Left ("Variable not found: " ++ show name)
```

```
eval (Add l r) = case eval l of
  m@(Left msg) -> m
  Right x -> case eval r of
    m@(Left msg) -> m
    Right y -> Right (x + y)
```

Exception Handling

Use of monads can hide the error handling

```
Evaluator a = Ev (Either String a)
instance Monad Evaluator where
    (Ev ev) >>= k = case ev of
        Left msg -> Ev (Left msg)
        Right v -> k v
    return v = Ev (Right v)
    fail msg = Ev (Left msg)
```

Since 2014, instance of Functor and Applicative also necessary!

```
eval :: Expr -> Evaluator Int
eval (Mul l r) = do lres <- eval l
                     rres <- eval r
                     return (lres*rres)
```

https://www.schoolofhaskell.com/user/bartosz/basics-of-haskell/10_Error_Handling

Suitable for combining non-deterministic computations
can return multiple results and we want to continue with all

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= k = concat (map k xs)

return :: a -> [a]
return x = [x]
```

List Comprehensions

```
squares lst = do
    x <- lst
    return (x * x)
```

```
squares lst = lst >>= \x -> return (x * x)
```

```
squares lst = concat $ fmap k lst
  where k = \x -> [x * x]
```

List Comprehensions

```
pairs 11 12 = do
    x <- 11
    y <- 12
    return (x, y)
```

```
pairs 11 12 = [(x, y) | x <- 11, y <- 12]
```

```
pairs 11 12 = 11 >>= \x -> 12 >>= \y -> return (x,y)
```

Guards can also be added, but it requires `MonadPlus`, for more advanced combinations of computations.

Understanding IO Monad

Assume we are implementing getchar in Haskell
what type should it have?

```
getchar :: Char
```

We can then implement

```
get2chars :: String
get2chars = [getchar, getchar]
```

Haskell functions are pure, hence the compiler will

- remove the double call by caching the return value
- if it called the function twice, it would be in arbitrary order

How to solve caching?

Adding a (fake) parameter makes the calls different

```
getchar :: Int -> Char
```

```
get2chars _ = [getchar 1, getchar 2]
```

The calls can still be executed in an arbitrary order

Data dependency can order function execution
(if a result of one function is used by another function)

```
getchar :: Int -> (Char, Int)
```

```
get2chars i0 = [a,b] where (a,i1) = getchar i0
                                (b,i2) = getchar i1
```

Sequencing Through Data Dependency

The same sequencing problems would reoccur

```
get4chars = [get2chars 1, get2chars 2]
```

Hence we want

```
get4chars :: Int -> (String, Int)
```

```
get4chars i0 = ([a,b],i2) where (a,i1) = get2chars i0  
                           (b,i2) = get2chars i1
```

We are forcing a specific sequence of executing functions using data dependencies

Good intuition for how IO works

```
type IO a = RealWorld -> (a, RealWorld)  
> :i IO
```

RealWorld is a fake type serving as the Int from above

The main function is of type IO ()

```
main :: RealWorld -> ((), RealWorld)
```

All IO functions take the real world as an argument and return (a possibly modified) new version of the world

Example

Function main calling getChar two times:

```
getChar :: RealWorld -> (Char, RealWorld) -- IO Char

main :: RealWorld -> (((), RealWorld)) -- IO ()
main world0 = let (a, world1) = getChar world0
              (b, world2) = getChar world1
              in (((), world2))
```

Only main gets the RealWord. Therefore only main can execute IO actions.

Hides passing of the RealWorld value from the programmer

```
(>>) :: IO a -> IO b -> IO b
(action1 >> action2) world0 =
  let (a, world1) = action1 world0
      (b, world2) = action2 world1
  in (b, world2)
```

IO Monad

Hides passing of the RealWorld value from the programmer

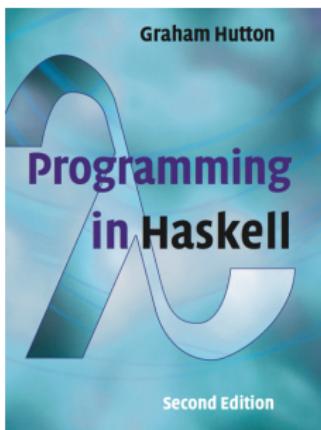
```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
  let (a, world1) = action1 world0
      (b, world2) = action2 a world1
  in (b, world2)
```

```
return :: a -> IO a
return x world0 = (x, world0)
```

Monad is **just** a convenient abstraction to do something like this!

Acknowledgements

- https://wiki.haskell.org/Introduction_to_IO
- https://wiki.haskell.org/IO_inside
- https://www.schoolofhaskell.com/user/bartosz/basics-of-haskell/10_Error_Handling
- <http://learnyouahaskell.com/functors-applicative-functors-and-monoids>



Monads Summary

- IO in pure functional programming is problematic
 - it prevents optimization possible with pure functions
 - it requires explicit ordering of pseudo-function calls
- Haskell encloses these operations to IO actions
 - no result of pseudo-function can leave the IO "container"
- Monads are a useful abstraction for
 - sequencing operations on containers
 - making operation within containers
- Build-in Monads
 - Maybe, Either e, [], IO

- Decent random numbers
 - `System.Random` (may not be installed by default in GHC)
- Cryptographically secure random numbers
 - `Crypto.Random`
- Getting random numbers generator
 - `mkStdGen <seed>`
 - `getStdGen`

- Getting a random number
 - `randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)`
- Range can be inferred from output type
 - `random :: (RandomGen g, Random a) => g -> (a, g)`
- Using the standard generator in the IO monad
 - `randomRIO (0,1)`
 - `randomRIO (0,1::Float)`
 - `randomIO :: IO Float`

Random sequence

```
myRnds :: Int -> [Float]
myRnds seed = randSeq (mkStdGen seed)
  where randSeq gen = let (v,g2) = random gen
                      in v:randSeq g2
```

Build-in variant

- `randoms <generator>`
- `randomRs <range> <generator>`

Random with IO

```
*Main> :t getStdGen
getStdGen :: IO StdGen
*Main> :t random
random :: (RandomGen g, Random a) => g -> (a, g)
```

```
import System.Random

main = do
    g <- getStdGen
    print . take 10 $ (randomRs ('a', 'z')) g
    print . take 10 $ (randomRs ('a', 'z')) g
```

Random values of custom type

Type must be an instance of class Random

```
data Coin = Heads |  
           Tails deriving (Show, Enum, Bounded)  
  
instance Random Coin where  
    randomR (a, b) g =  
        let (x, g') = randomR (fromEnum a, fromEnum b) g  
        in (toEnum x, g')  
    random g = randomR (minBound, maxBound) g
```



Functional Programming

Lecture 13: FP in the Real World

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Mixed paradigm languages

Functional programming is great
easy parallelism and concurrency
referential transparency, encapsulation
compact declarative code

Imperative programming is great
more convenient I/O
better performance in certain tasks

There is no reason not to combine paradigms

UK Job Market (May 2020)

Results 1 - 30 of 126

Skill / Job Role (Historical trends)	Rank 6 Months to 24 May 2020	Rank Change Year-on-Year	Median Salary 6 Months to 24 May 2020	Historical Permanent Job Ads	Live Job Vacancies
SQL	4	▼ -1	£50,000	17,242 (18.57%)	1,257
JavaScript	5	► 0	£52,500	16,861 (18.16%)	1,714
C#	8	▼ -1	£50,000	14,311 (15.41%)	1,401
Java	10	▼ -2	£65,000	13,149 (14.16%)	1,039
Python	13	▲ +4	£62,500	11,424 (12.30%)	1,237
PHP	64	► 0	£45,000	3,943 (4.25%)	425
C++	72	▼ -14	£55,000	3,694 (3.98%)	595
PowerShell	74	▲ +6	£52,500	3,605 (3.88%)	281
TypeScript	91	▲ +131	£57,500	3,036 (3.27%)	377
C	105	▼ -18	£52,500	2,708 (2.92%)	454
T-SQL	151	▼ -35	£47,500	2,142 (2.31%)	189
Bash Shell	164	▼ -9	£59,000	1,961 (2.11%)	197
Ruby	187	▼ -33	£62,000	1,661 (1.79%)	173
Scala	208	▲ +1	£72,500	1,485 (1.60%)	110
Java 8	216	▲ +9	£65,000	1,436 (1.55%)	102
R	231	▲ +31	£65,000	1,332 (1.43%)	66
Go	232	▲ +75	£67,500	1,331 (1.43%)	145
Kotlin	279	▲ +224	£70,000	1,051 (1.13%)	104
ES6	287	▲ +5	£57,500	999 (1.08%)	82

Results 1 - 30 of 126

Skill / Job Role (Historical trends)	Rank 6 Months to 24 May 2020	Rank Change Year-on-Year	Median Salary 6 Months to 24 May 2020	Historical Permanent Job Ads	Live Job Vacancies
F#	651	▲ +230	£90,000	246 (0.26%)	23
Solidity	879	▲ +176	£90,000	16 (0.017%)	2
Haskell	828	▲ +86	£85,000	67 (0.072%)	12
BrightScript	886	► -	£85,000	9 (0.010%)	
Lisp	894	▲ +206	£82,500	1 (0.001%)	
OCaml	893	▲ +189	£81,250	2 (0.002%)	6
Elixir	868	▲ +151	£80,000	27 (0.029%)	8
Clojure	783	▲ +87	£80,000	112 (0.12%)	14
AspectJ	871	▲ +229	£79,000	24 (0.026%)	2
ANSI SQL	891	▲ +184	£75,000	4 (0.004%)	1
ML	892	▲ +186	£75,000	3 (0.003%)	
U-SQL	863	▲ +215	£75,000	32 (0.035%)	2
Cypher	888	▲ +193	£72,500	7 (0.008%)	
Scala	208	▲ +1	£72,500	1,485 (1.60%)	110
C-shell	892	▲ +203	£72,500	3 (0.003%)	5
CoffeeScript	892	▲ +193	£72,000	3 (0.003%)	
Kotlin	279	▲ +224	£70,000	1,051 (1.13%)	104
ES7	805	▲ +92	£69,250	90 (0.097%)	11
Julia	867	▲ +211	£67,500	28 (0.030%)	1

Most popular websites

Back-end (Server-side) table in most popular websites

Websites	C#	C	C++	D	Erlang	Go	Hack	Java	JavaScript	Perl	PHP	Python	Ruby	Scala	Xhp
Google.com	No	Yes	Yes	No	No	Yes	No	Yes	No	No	Yes	Yes	No	No	No
YouTube.com	No	Yes	Yes	No	No	Yes	No	Yes	No	No	No	Yes	No	No	No
Facebook.com	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes
Yahoo	No	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Amazon.com	No	No	Yes	No	No	No	No	Yes	No	Yes	No	No	No	No	No
Wikipedia.org	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
Twitter.com	No	No	Yes	No	No	No	No	Yes	No	No	No	No	Yes	Yes	No
Bing	Yes	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No
eBay.com	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
MSN.com	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
LinkedIn.com	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
Pinterest	No	No	No	No	Yes	No	No	No	No	No	No	Yes	No	No	No
WordPress.com	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No

Source: Wikipedia

Scala

Quite popular with industry

Multi-paradigm language

- simple parallelism/concurrency
- able to build enterprise solutions

Runs on JVM

Scala vs. Haskell

- Adam Szlachta's slides

Is Java 8 a Functional Language?

Based on:

<https://jlordiales.me/2014/11/01/overview-java-8/>

Functional language

first class functions

higher order functions

pure functions (referential transparency)

recursion

closures

currying and partial application

First class functions

Previously, you could pass only classes in Java

```
File[] directories = new File(".").listFiles(new FileFilter() {  
    @Override  
    public boolean accept(File pathname) {  
        return pathname.isDirectory();  
    }  
});
```

Java 8 has the concept of method reference

```
File[] directories = new File(".").listFiles(File::isDirectory);
```

Lambdas

Sometimes we want a single-purpose function

```
File[] csvFiles = new File(".").listFiles(new FileFilter() {  
    @Override  
    public boolean accept(File pathname) {  
        return pathname.getAbsolutePath().endsWith("csv");  
    }  
});
```

Java 8 has lambda functions for that

```
File[] csvFiles = new File(".").  
    .listFiles(pathname -> pathname.getAbsolutePath().endsWith("csv"));
```

Streams

We want a list of adult users grouped by sex

```
public Map<Sex, List<User>> groupUsers(List<User> allUsers) {  
    Map<Sex, List<User>> result = new HashMap<>();  
    for (User user : allUsers) {  
        if (user.getAge() >= 18) {  
            List<User> currentUsers = result.get(user.getSex());  
            if (currentUsers == null) {  
                currentUsers = new ArrayList<>();  
                result.put(user.getSex(), currentUsers);}  
            currentUsers.add(user);  
        }  
    }  
    return result;}  
}
```

Streams

In Java 8, we can use higher order functions

```
public Map<Sex, List<User>> groupUsers(List<User> allUsers) {  
    return allUsers  
        .parallelStream()  
        .filter(user -> user.getAge() >= 18)  
        .collect(groupingBy(User::getSex));  
}
```

Declarative style (and lazy)

easier to understand

easier to parallelize

Is Java 8 a Functional Language?

Functional language

first class functions	Yes
higher order functions	Yes
pure functions (referential transparency)	No
recursion	No tail recursion optimization by default
closures	Only values, variables become final
currying and partial application	Yes

No, but it provides many of the nice FP features

FP aspect in mainstream languages

	First class functions	Higher order functions	Lambda	Closures	List comprehensions	Referential transparency	Currying/partial application	Data immutability	Pattern matching	Lazy evaluation
Haskell	+	+	+	+	+	+	+	+	+	+
Java 8	(+)	+	+	+/-	-	-	(+)	(+)	-	(+)
C++14	+	+	+	+	-	-	(+)	(+)	(+)	(+)
Python	+	+	+	+	+	-	+	(+)	(+/-)	(+)
JavaScript	+	+	+	+	+	-	+	(+)	(+/-)	(+)
MATLAB	+	+	+	+	-	-	+	(+)	-	(+)

Erlang

Haskell – complex types + concurrency support

- Immutable data
- Pattern matching
- Functional programming
- Distributed
- Fault-tolerant

Map Reduce

Distributed parallel big data processing inspired by functional programming

- John Hughes's slides

Lisp for Scripting in SW Tools

- Emacs: extensible text editor
- AutoCAD: technical drawing software
- Gimp: gnu image manipulation program

Gimp

User scripts in: `~/.gimp-2.8/scripts`

Register the function by

`script-fu-register`

`script-fu-menu-register`

`Filters → Script-Fu → Refresh Scripts`

See example source code in a separate file.

TAKE-AWAYS FROM FP

Declarative programming

- write what should be done and leave how to the **optimizer**
 - particularly interesting in distributed setting
- easier to understand, no need to go back from how to what

Minimizing Side Effects

- reusability
- predictability
- concurrency
- lower mental load (modularity/encapsulation)

It is easier than it seems!

Immutability

You can use it in any programming language to
ease parallelization
avoid defensive copying
avoid bugs in hashmaps / sets
consistent state even with exceptions
allows easier caching

It is not as inefficient as it seems!

Recursion

- Many problems are naturally recursive
 - easier to understand / analyze
 - less code, less bugs
 - combines well with immutability
- A great universal tool

Exam

Remote test

- recording screen, camera, mic.
- may be asked to explain the solution orally

Schedule

- 40 min test
 - anything hard to evaluate by programming
- 15 min break
- 3h of programming at computers (>50% points)
 - ~2 Haskell and ~2 Scheme tasks
 - upload system

Dates (tentative): 3.6. 9:00; ...