Functional Programming
Lecture 4

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Pattern matching

Pattern matching

Pattern matching allows deconstructing a data structure and based on its components to branch the computation.

```
(match exp
    [pattern1 exp1]
    [pattern2 exp2]
    ...)
```

Similar to `cond` matching the expression `exp` against the patterns.

The patterns might include variables. If a pattern matches `exp`, its variables are bound to the corresponding values in `exp`.

Matching literals and predicates

```
(struct point (x y))

(match exp
  [0 'zero]
  [1 'one]
  [2 'two]
  ["abc" 'abc]
  [(point 0 0) 'point]
  [(? string?) 'string]
  [(and (? number? x) (? positive?))
   (format "positive num ~a" x)]
  [_ 'other])
```

Matching lists

```
(match lst
  [(list) 'empty]
  [(list x) (format "singleton (~a)" x)]
  [(list 'fn ys ...)
   (format "fn and rest ~a" ys)]
  [(list (list 'fn args ...) ys ...)
   (format "fn with ~a and rest ~a" args ys)]
  [(list 1 ys ... z)
   (format "1, rest ~a and last ~a" ys z)]
  [(list x ys ...)
   (format "~a and rest ~a" x ys)]
  [_ 'other])
```

Lazy evaluation

Postponing evaluation

Scheme/Racket uses almost always applicative/strict evalution order, i.e. before evaluating `(fn a1 a2 ...)` it evaluates all subexpressions `fn, a1, ...` from left to right.

Exceptions are syntactic forms like `if,cond,and,or`:

```
(if (< 0 1) 0 (/ 1 0)) => 0
```

Can we force the interpreter to postpone the argument evaluations?

```
(define (my-if c a b) (if c a b))
(my-if (< 0 1) 0 (/ 1 0)) => /: division by zero
```

Thunk

Defining a function creates a closure but its body is not evaluated.

```
(my-if (< 0 1)
       (lambda () 0)
       (lambda () (/ 1 0))) => #<procedure>
```

```
(define (my-lazy-if c a b) (if c (a) (b)))
```

```
(lambda () exp) = (thunk exp)
```

```
(my-lazy-if (< 0 1)
            (thunk 0) (thunk (/ 1 0))) => 0
```

Applications of thunks

- Special forms like `if, cond, and, or`
- Passing an expression to be evaluated later, e.g. `(thread (thunk exp))`
- Programs can use potentially large or infinite data structures, e.g. streams. This can make the code more efficient or improve its modularity.

Streams

Streams

Streams are ordered sequences of elements that are evaluated when needed (lazily), can be infinite.

```
(delay exp) ; called a promise, similar as thunk
```

```
(force exp) ; similar as (exp)
```

Moreover, `force` caches the result of `(exp)` so `exp` is not evaluated again by repeated forcing.

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

(define nats (ints-from 0))
(force (cdr nats)) => (1 . #<promise:...
```

Functions for streams

Functions creating and manipulating streams are implemented in Racket (not Scheme)

streams	lists
`stream-cons`	`cons`
`stream`	`list`
`stream-first`	`car`
`stream-rest`	`cdr`
`stream-empty?`	`null?`
`stream-filter`	`filter`
`stream-map`	`map`
`stream-take`	`take`

Any list can be used as a stream. Finite stream can be converted to a list by `stream->list`.

Streams defined explicitly

We can specify a stream by defining a generating function computing a next element from previous ones.

```
(define (nats n)
  (stream-cons n (nats (+ n 1))))
```

We can construct an infinite stream for any function *f* computing the next element from the current one:

```
(define (repeat f a0)
  (stream-cons a0 (repeat f (f a0))))
```

Streams defined implicitly

The stream of ones: $\overline{1} = 1, \overline{1}$.

```
(define ones (stream-cons 1 ones))
```

The stream of $a, b, a, b, a, \ldots$: $\overline{ab} = a, b, \overline{ab}$.

```
(define ab (stream-cons 'a (stream-cons 'b ab)))
```

```
(define stream-days
        (stream* 'mon 'tue 'wed
                 'thu 'fri 'sat 'sun
                 stream-days))
```

Streams defined implicitly

`stream-map` works only for a single stream.

```
(define (add-streams s1 s2)
  (stream-cons (+ (stream-first s1)
                  (stream-first s2))
               (add-streams (stream-rest s1)
                            (stream-rest s2))))
```

$$\overline{\mathbb{N}} = 0, \overline{1} + \overline{\mathbb{N}}$$

	1	1	1	1	1
+	0	1	2	3	4
0	1	2	3	4	5

```
(define nats2
  (stream-cons 0 (add-streams ones nats2)))
```

Applications of streams

Streams are useful when we need a somethink like a Python's iterator. Compare

```
(stream-fold + 0 (in-range 10000000))
(foldl + 0 (range 10000000))
```

Lazily evaluated data structure provide better modularity because we can separate generating code from further processing functions like pruning or searching.

Generate → Prune → Search

Newton-Raphson

$n \mapsto \sqrt{n}$

g_0 – an initial guess

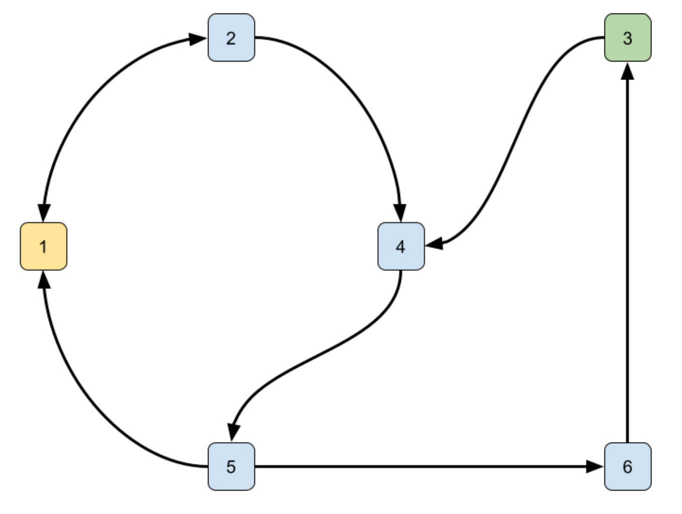
$g_{k+1} = \frac{1}{2}(g_k + n/g_k)$ — the next guess

When $g_k = g_{k+1} = \frac{1}{2}(g_k + n/g_k)$, we have

$$\begin{aligned} 2g_k &= g_k + n/g_k \\ g_k &= n/g_k \\ g_k &= \sqrt{n} \end{aligned}$$

In practice, we test $|1 - \frac{g_k}{g_{k+1}}| \leq \varepsilon$

Lazy tree



What have we learned?

- Pattern matching allows simultaneous computation branching and data structure deconstruction.
- We can control the evaluation order by delaying the evaluation.
- Streams are lazy lists which can be even infinite.
- Lazily evaluated structures provide better modularity.