Sjekkliste for eksamen i INF2810:

* Kunne forskjell på evalueringsmetodene
  + Eager-evaluation (applicative-order evaluation)
    - Evaluerer argumentene: kall prosedyren på verdiene
    - Standard i Scheme
  + Lazy-evaluation (utsatt evaluering) / normal-order evaluation
    - Computing-on-demand
      * Vi beregner ikke verdien før vi trenger den
    - Prosedyren kalles med argumentuttrykkene: evalueres først ved behov
  + Gir oss samme resultat, men vi må forbeholde oss til at vi jobber med ren funksjonell kode.
    - Det er utrygt å bruke destruktive operasjoner hvis vi bruker normal-order evaluering
  + Normal-order evaluering er mer effektivt og bruker mindre tid/ressurser under kjøring
  + Delay gir oss mulighet til å bestemme når et uttrykk skal evalueres
    - Eks. (define bar (delay (\* 2 8))) = bar 🡪 #<promise>
  + Force kaller på delay verdi
    - (force (promise)) 🡪 (promise)
* Kunne forstå hvordan strømmer virker
  + Strømmer konstrueres mens de brukes og implementerer normal-order evaluering
    - Stream-car
    - Stream-cdr
    - Stream-null
    - The-empty-stream
    - Cons-stream
  + Strømmer støtter *random-access* 
    - Det vil si at vi kan bygge inn memoisering i delay
      * Strømelementer kan brukes flere ganger
      * Kan bygge inn memoisering i delay
    - Hvis vi memoiserer en prosedyre kan den ikke ta inn argumenter
  + Vi kan lage uendelige sekvenser av en liste ettersom vi kan bruke strømmer
    - (define (int-starting-from n)

(cons-stream n (int-starting-from (+ n 1)))) 🡪

(n . #<promise>)

* Forstå hvordan jobbe med trær
  + Lister av lister kan sees på som trær
    - Hvert element i listen er en gren
    - Liste som er et element kalles for et subtre
    - Løvnoder er atomære verdier i treet
  + Tre ting kan eksistere i et tre
    - Den tomme lista
    - En liste
    - En løvnode (atomær verdi)



Hvordan strukturere et tre:

(define (fringe tree)

(cond ((<hvis noden er null>))

((<hvis vi har kommet til et subtre>))

(else (<hva skal skje hvis vi har kommet til en løvnode>))))

* + Gir opphav til en trerekursiv prosess
    - Eksponentiell vekst (O(n))
    - Ofte mye redundante beregninger
  + Viktig at rekursjonen går ned i alle lister (sublister)

Samler opp alle løvnodene i en liste:

(define (fringe tree)

(cond ((null? tree) ‘())

((pair? tree)

(append (fringe (car tree))

(fringe (cdr tree))))

(else (list tree))))

Teller antall løvnoder i et tre:

(define (count-leaves tree)

(cond ((null? tree) ‘())

((pair? tree)

(+ (count-leaves (car tree))

(count-leaves (cdr tree)))

(else 1)))

Kan også implementere map for et tre:

(define (tree-map proc tree)

(cond ((null? tree) ‘())

((pair? tree)

(cons (tree-map proc (car tree))

(tree-map proc (cdr tree))))

(else (proc tree))))

Hvordan lage et tre:

(define (make-tree entry left right)

(list entry left right))

(define (entry tree)

(car tree))

(define (left-branch tree)

(cadr tree)) \*cadr av et tre er hele «venstresiden» av et tre

(define (right-branch tree)

(caddr tree)) \*caddr er hele «høyresiden» av et tre

* Kunne tegne omgivelsesdiagram 🡪 ok?
* Kunne tegne boks og peker diagram 🡪 ok
* Kunne lage monitor prosedyre 🡪Nei
* Kunne forskjellen mellom OOP og FP (hva kjennetegner FP)
  + FP er prosedyrebasert
  + Data/datastrukturer vil ikke bli mutert, men heller lage en ny datatype eller struktur
  + Funksjoner er førsteklasses «borgere»
  + FP forandrer ikke tilstanden til data
  + Mer modularitet (enklere å teste, generealisere, gjenbruk)
  + Mindre sideeffekter
  + I FP er data og operasjoner ikke sterkt knyttet til hverandre og er mer abstrakt. Med andre ord kan vi bruker samme prosedyre for flere typer datastrukturer
* Må kunne lage hale-rekursive (iterative) og ikke hale-rekursive prosedyrer 🡪 Ikke ok enda
* Vite hva message passing er 🡪 ok
* Må vite hva destruktive operasjoner er og hvordan det brukes
  + Bruker set! til å gi verditilordninger til variabler
    - Eks. (set! foo 42)
* Forstå hvordan rekursjon, rekursive kall fungerer (basics) og halerekursive kall (forskjellen mellom iterativ og rekursiv prosess)
* Pugge basics som hva ‘and ‘or returnerer og forskjellen mellom ‘eq og ‘equal
  + Or
    - Hvis ingen argumenter 🡪 #f
    - Evaluerer dermed neste argument og returnerer det første som ikke blir #f
  + And
    - Hvis ingen argumenter 🡪 #t
    - Hvis det bare er et argument 🡪 verdien
    - Returnerer det siste argumentet hvis ingen evaluerer til #f
* Kan lage køer og forstå hvordan de fungerer
  + Koseptet med køer:
    - Lage et cons par hvor car peker til en liste
    - cdr peker til den siste verdien i listen
  + Eks.

Abstraksjonsbarriere:

(define (make-que)

(cons ‘() ‘() ))

(define (que-empty? queue)

(null? (car queue)))

(define (que-insert! object que)

(let ((new (cons object ‘())))

(if (que-empty? que)

(set-car! que new)

(set-cdr! (cdr que) new)

(set-cdr! que new)))

(define (que-delete! q)

(if (que-empty? queque)

«Error: empty queue»

(let ((element (caar queue)))

(set-car! queue (cdar queue))

Element)))

Interaksjon:

(define q (make-que))

(que-insert! ‘(a b c) q) 🡪



(que-insert! ‘d q)



(delete-que! q) 🡪 a



Begreper som ofte blir brukt:

* Abstraksjonsbarriere
  + Vi lager en prosedyre som kan bli brukt av andre prosedyrer og som «*skjuler*» datastrukturen for prosedyrene som kaller på denne abstrakte prosedyren
  + Ofte blir slike prosedyrer brukt til et veldig generelt bruk
  + Eks.
    - (define (find-first list)

((car list))) -> Returnerer alltid første element

* Message passing
  + En prosedyre returnerer et lambdauttrykk som returnerer en verdi avhengig av hva som blir sendt med som et argument.
  + Eks.

(define (cons x y)

(lambda (message)

(cond ((= message 0) x)

((= message 1) y)))))

(define (car proc)

(proc 0)) 🡪 første verdien

((cons 1 2) 0) 🡪 1

(car (cons 1 2) 🡪 1