IN2040: Funksjonell Programmering

Introduksjon

Martin Steffen

Universitetet i Oslo

21. august 2023

Tema for i dag

- Introduksjon
- Om digital undervisning
- Lærebok
- Hva skal vi lære?
- Praktisk informasjon
  - Grupper
  - Obliger

Fysiske forelesninger 2023

- Forelesningene i IN2040 høsten 2023 blir i auditorium
- screencast for hver forelesning (ly + foiler), fra 2020
- vi lenker til YouTube fra forelesningsplannen

Hei!

- Forelesere: Martin Steffen (`msteffen`)
- På screencastene også: Erik Velldal (`erikve`) og Stephan Oepen (`oe`)

Hei!

- Forelesere: Martin Steffen (msteffen)
- På screencastene også: Erik Velldal (erikve) og Stephan Oepen (oe)

Gruppelærere (og rettere)

Anton Zelentsov, Maham Iftikhar, Jakub
Mikolaj Wasylkow, Oskar Haukebøe
({antonz|mahami|jakubmw|oskah}
@ifi.uio.no)

- Gruppe 1, tue. 10–12
- Gruppe 2, man. 14–16
- Gruppe 3, tue. 8–10
- Gruppe 4, tue. 14-16

Hei!

- Forelesere: Martin Steffen (msteffen)
- På screencastene også: Erik Velldal (erikve) og Stephan Oepen (oe)

Gruppelærere (og rettere)

Anton Zelentsov, Maham Iftikhar, Jakub
Mikolaj Wasylkow, Oskar Haukebøe
({antonz|mahami|jakubmw|oskah}
@ifi.uio.no)

- Gruppe 1, tue. 10–12
- Gruppe 2, man. 14–16
- Gruppe 3, tue. 8–10
- Gruppe 4, tue. 14-16

Ekstra rettere

- Lise Chen
  (liseche@ifi.uio.no)
- Emir Kukuruzovic
  (emirkuk@ifi.uio.no)
- Mathias Strømberg Durkis
  (mathisdu@math.uio.no)
- Dan Ngyen
  (danng@ifi.uio.no)
- Sergey Jakobsen
  (sergeyj@ifi.uio.no)
- Elise Møllerstedt Gunnestad
  (elisegun@ifi.uio.no)

Opplegg for gruppetimene

Generelt

- Første halvdel av timen: ofte felles tavleundervisning.
- Andre halvdel: selvstendig arbeid og hjelp med koding.
- Mer fokus på selvstendig arbeid siste uke før en innlevering.
- Innimellom skal vi også poste noen (frivillige) ukesoppgaver.

Første gruppetime

- Starter neste uke (man. 28.8.2023). Tema:
- Gjennomgang av programmeringsomgivelse.
  - Scheme i DrRacket.
- Arbeid med innlevering 1a (legges ut snart med frist om to uker).

Innleveringer

- Tre obliger, hver i to deler (‘a’ + ‘b’), dvs. seks innleveringer tilsammen.
- Man kan oppnå opptil 10 poeng per innlevering
- Kreves minst 12 poeng (av 20 mulige) per oblig (‘a’ + ‘b’) for å bestå.
- Alle tre obligene må vurderes som bestått for å kunne ta eksamen.
- NB: Innleveringsfrister utsettes bare ved egenmelding eller legeerklæring; ingen omlevering.
- Obligfrister legges alltid til en fredag kl 23:59 (annenhver uke)
- Oblig (1) må løses individuelt, mens (2) og (3) kan løses i grupper av to eller tre studenter sammen.

Innleveringer

- Tre obliger, hver i to deler ('a' + 'b'), dvs. seks innleveringer tilsammen.
- Man kan oppnå opptil 10 poeng per innlevering
- Kreves minst 12 poeng (av 20 mulige) per oblig ('a' + 'b') for å bestå.
- Alle tre obligene må vurderes som bestått for å kunne ta eksamen.
- NB: Innleveringsfrister utsettes bare ved egenmelding eller legeerklæring; ingen omlevering.
- Obligfrister legges alltid til en fredag kl 23:59 (annenhver uke)
- Oblig (1) må løses individuelt, mens (2) og (3) kan løses i grupper av to eller tre studenter sammen.
- Konkurranse: de som samler inn flest poeng tilsammen gjennom semesteret får premie; den tradisjonsrike rekursjons-tskjorta!

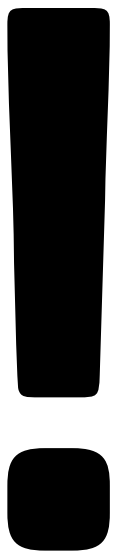
Spørsmål og hjelp

- Gruppetimene: Gruppelærerne er der for å hjelpe og veilede.
- Diskusjonsforum
  Gitub issues (Piazza ikke lenger tillatt ved UiO):

  `https://github.uio.no/IN2040/h23/issues?q=is%3Aissue`
- (Astro Discourse) (???)
- Felles adresse til fag- og-gruppelærere: `in2040-hjelp@ifi.uio.no`

Beskjeder

- Husk å sjekke beskjeder på `https://minestudier.uio.no/` eller emnets semesterside, og UiO-eposten din.



Lange tradisjoner

- IN2040 er et emne med lange tradisjoner.
- Har i ulike varianter blitt undervist ved UiO i flere tiår,
- i likhet med en rekke andre universiteter rundt i verden.
- Springer ut av et emne ved MIT på tidlig 80-tall,
  som også ga opphavet til læreboka.
- Lecture 1A in MIT 6.001, by Hal Abelson, 1986:
  https://youtu.be/2Op3QLzMgSY

Læreboka

- *Structure and Interpretation of Computer Programs*, 2. utg., Abelson & Sussman.
- En klassiker innen informatikk
- 1. utgave 1984, 2. utg. 1996
- Brukes fortsatt på en rekke universiteter
- Gammel men ikke støvete.
- Hvordan kan den fortsatt være relevant?

(Sussman)

(Abelson)

Læreboka

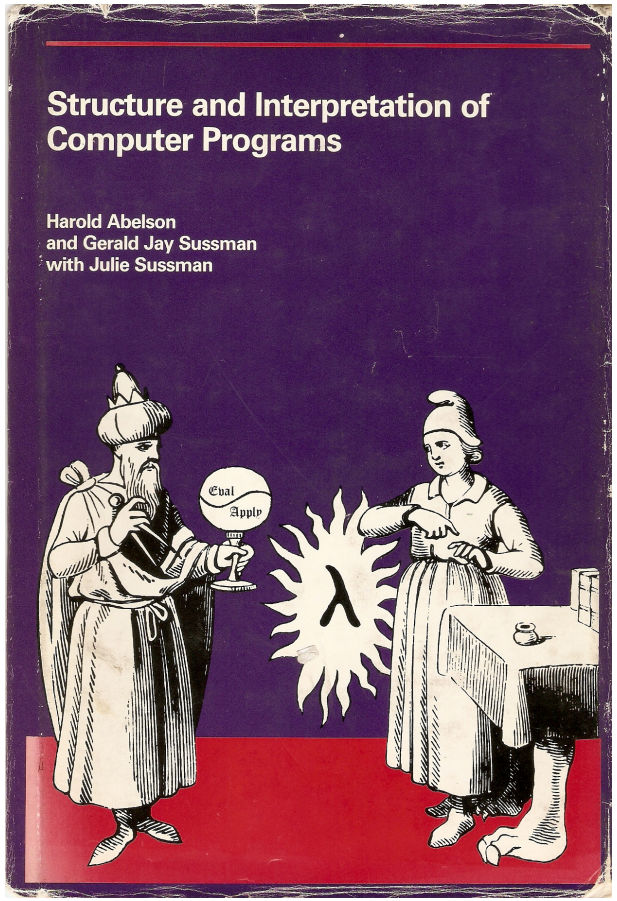
- *Structure and Interpretation of Computer Programs*, 2. utg., Abelson & Sussman.
- En klassiker innen informatikk
- 1. utgave 1984, 2. utg. 1996
- Brukes fortsatt på en rekke universiteter
- Gammel men ikke støvete.
- Hvordan kan den fortsatt være relevant?
- Som med klassikere flest er den tidløs.
  - Basert på Scheme, men ikke fokus på lavnivå-detaljer ved språk og syntaks.
  - Generelle teknikker for å kontrollere kompleksiteten i programmer.

(Sussman)

(Abelson)

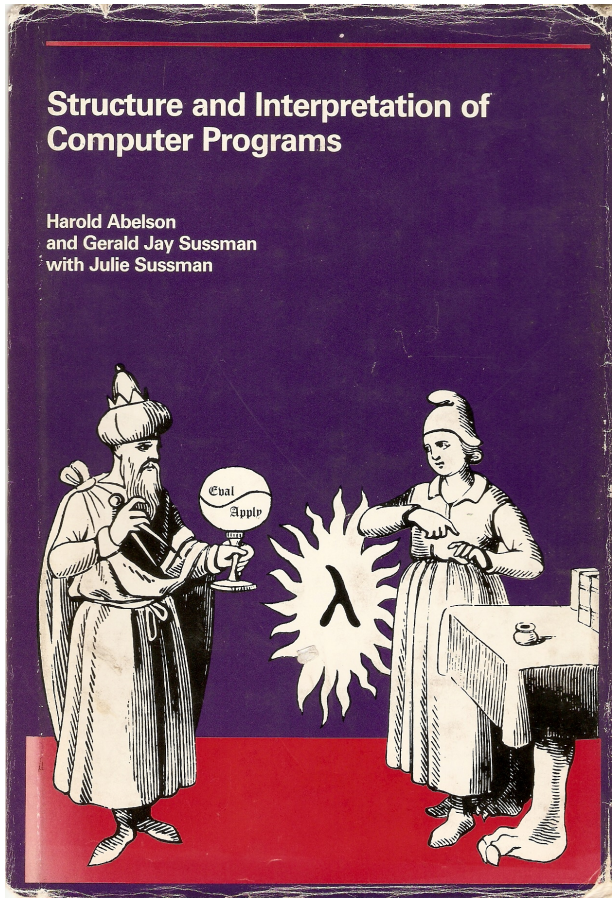
Læreboka (forts.)

- Aka *SICP* eller *the Wizard book*
- Papirversjonen får du på Akademika.
- Mange gratis versjoner på nett, bl.a.:
- https://web.mit.edu/6.001/6.037/sicp.pdf
  PDF versjon fra MIT



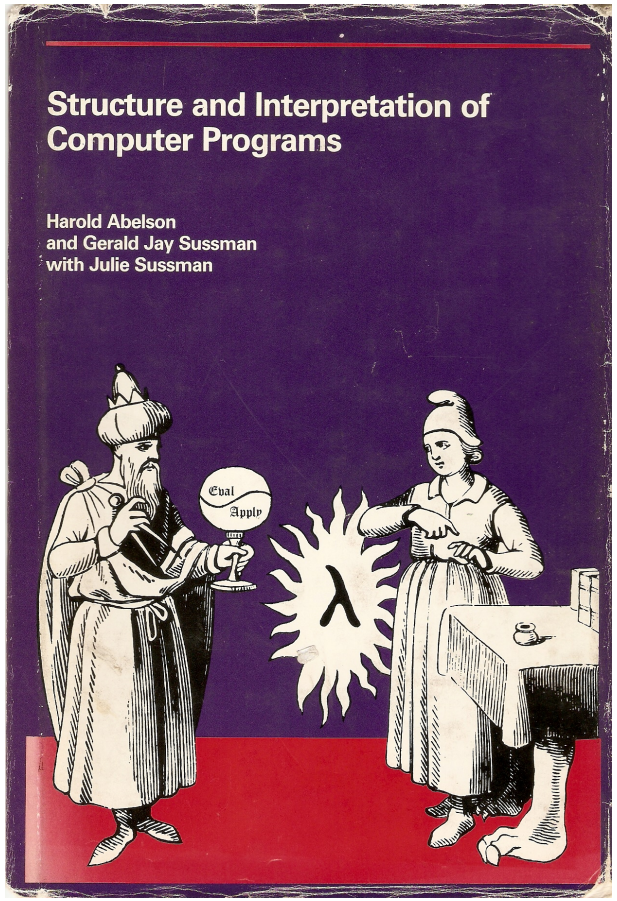
Læreboka (forts.)

- Aka *SICP* eller *the Wizard book*
- Papirversjonen får du på Akademika.
- Mange gratis versjoner på nett, bl.a.:
- https://web.mit.edu/6.001/6.037/sicp.pdf
  PDF versjon fra MIT
- PDF-versjon (ulike versjoner tilpasset lesebrett, tablets, mobil, etc):
  sicpebook.wordpress.com/ebook/



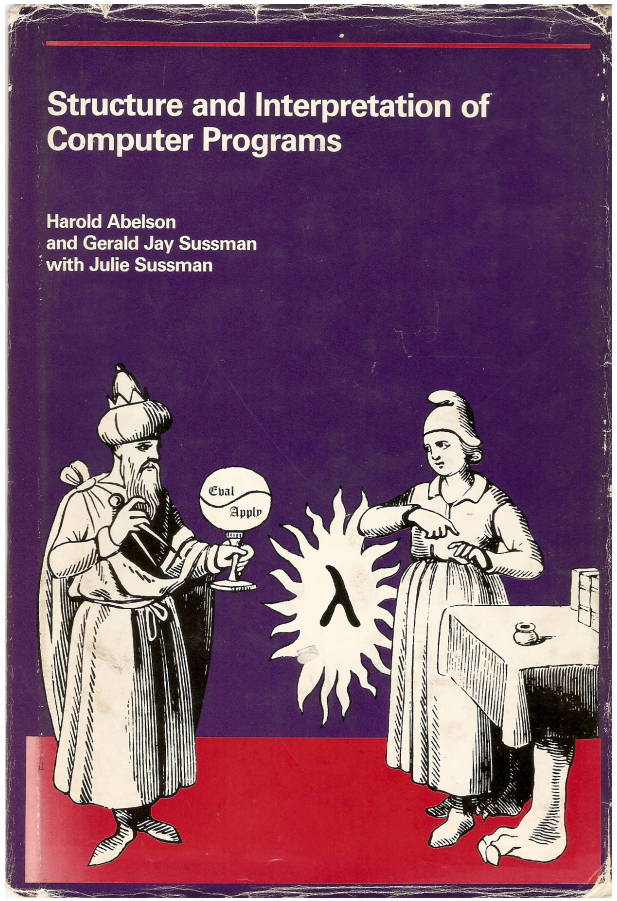
Læreboka (forts.)

- Aka *SICP* eller *the Wizard book*
- Papirversjonen får du på Akademika.
- Mange gratis versjoner på nett, bl.a.:
- https://web.mit.edu/6.001/6.037/sicp.pdf
  PDF versjon fra MIT
- PDF-versjon (ulike versjoner tilpasset lesebrett, tablets, mobil, etc):
  sicpebook.wordpress.com/ebook/
- Interaktiv HTML: xuanji.appspot.com/isicp/



Læreboka (forts.)

- Aka *SICP* eller *the Wizard book*
- Papirversjonen får du på Akademika.
- Mange gratis versjoner på nett, bl.a.:
- https://web.mit.edu/6.001/6.037/sicp.pdf
  PDF versjon fra MIT
- PDF-versjon (ulike versjoner tilpasset lesebrett, tablets, mobil, etc):
  sicpebook.wordpress.com/ebook/
- Interaktiv HTML: xuanji.appspot.com/isicp/
- JavaScript Edition ut April 2022;
  mitpress.mit.edu/books/structure-and-interpretation-computer-programs-1



Noe av det vi skal se på i kurset

- Rekursjon
  - prosedyrer som kaller på seg selv

Noe av det vi skal se på i kurset

- Rekursjon
  - prosedyrer som kaller på seg selv
- Prosedyrer som førsteklasses objekter
  - prosedyrer er en datatype på linje med tall, symboler, strenger, lister, etc.
- Høyere-ordens prosedyrer
  - prosedyrer som tar andre prosedyrer som input-argument og/eller
  - prosedyrer som returnerer andre prosedyrer som output-verdi

Noe av det vi skal se på i kurset

- Rekursjon
  - prosedyrer som kaller på seg selv
- Prosedyrer som førsteklasses objekter
  - prosedyrer er en datatype på linje med tall, symboler, strenger, lister, etc.
- Høyere-ordens prosedyrer
  - prosedyrer som tar andre prosedyrer som input-argument og/eller
  - prosedyrer som returnerer andre prosedyrer som output-verdi
- Objektorientering via prosedyrer
  - *innkapsling* av data i prosedyrer for å skjule tilstandsvariabler

Noe av det vi skal se på i kurset

- Rekursjon
  - prosedyrer som kaller på seg selv
- Prosedyrer som førsteklasses objekter
  - prosedyrer er en datatype på linje med tall, symboler, strenger, lister, etc.
- Høyere-ordens prosedyrer
  - prosedyrer som tar andre prosedyrer som input-argument og/eller
  - prosedyrer som returnerer andre prosedyrer som output-verdi
- Objektorientering via prosedyrer
  - *innkapsling* av data i prosedyrer for å skjule tilstandsvariabler
- Strømmer og utsatt evaluering
  - definering av uendelige sekvenser og realisering av de enkelte elementene i disse ved utsatt evaluering etter behov

Noe av det vi skal se på i kurset

- Rekursjon
  - prosedyrer som kaller på seg selv
- Prosedyrer som førsteklasses objekter
  - prosedyrer er en datatype på linje med tall, symboler, strenger, lister, etc.
- Høyere-ordens prosedyrer
  - prosedyrer som tar andre prosedyrer som input-argument og/eller
  - prosedyrer som returnerer andre prosedyrer som output-verdi
- Objektorientering via prosedyrer
  - *innkapsling* av data i prosedyrer for å skjule tilstandsvariabler
- Strømmer og utsatt evaluering
  - definering av uendelige sekvenser og realisering av de enkelte elementene i disse ved utsatt evaluering etter behov
- Programmer som opererer på programmer
  - program = data
  - enkelt å gjøre i Lisp, siden programkoden er basert på språkets innebygde datastrukturer: lister

Viktige egenskaper ved funksjonell programmering

- Prosedyrer påvirker ikke ekstern tilstand.
  - Prosedyrer kalles for deres returverdi; uten bieffekter (*side-effects*).
- Prosedyrer er uavhengige av ekstern tilstand.
  - Kallet på en prosedyre gir alltid samme resultat med samme argument.

Viktige egenskaper ved funksjonell programmering

- Prosedyrer påvirker ikke ekstern tilstand.
  - Prosedyrer kalles for deres returverdi; uten bieffekter (*side-effects*).
- Prosedyrer er uavhengige av ekstern tilstand.
  - Kallet på en prosedyre gir alltid samme resultat med samme argument.
- Gjennomsiktig semantikk (*referential transparency*)
  - Semantikken til et uttrykk er uavhengig av hvor og når det brukes.
  - Funksjonskall kan erstattes med returverdi uten at programmets semantikk endres.

Viktige egenskaper ved funksjonell programmering

- Prosedyrer påvirker ikke ekstern tilstand.
  - Prosedyrer kalles for deres returverdi; uten bieffekter (*side-effects*).
- Prosedyrer er uavhengige av ekstern tilstand.
  - Kallet på en prosedyre gir alltid samme resultat med samme argument.
- Gjennomsiktig semantikk (*referential transparency*)
  - Semantikken til et uttrykk er uavhengig av hvor og når det brukes.
  - Funksjonskall kan erstattes med returverdi uten at programmets semantikk endres.
- Enklere
  - å lese og vedlikeholde
  - å teste og debugge; enkeltfunksjoner kan testes i isolasjon.
  - å parallelisere.
- Ofte ansett for å gi økt sikkerhet (p.g.a. alt over).

Viktige egenskaper ved funksjonell programmering

- Prosedyrer påvirker ikke ekstern tilstand.
  - Prosedyrer kalles for deres returverdi; uten bieffekter (*side-effects*).
- Prosedyrer er uavhengige av ekstern tilstand.
  - Kallet på en prosedyre gir alltid samme resultat med samme argument.
- Gjennomsiktig semantikk (*referential transparency*)
  - Semantikken til et uttrykk er uavhengig av hvor og når det brukes.
  - Funksjonskall kan erstattes med returverdi uten at programmets semantikk endres.
- Enklere
  - å lese og vedlikeholde
  - å teste og debugge; enkeltfunksjoner kan testes i isolasjon.
  - å parallelisere.
- Ofte ansett for å gi økt sikkerhet (p.g.a. alt over).
- Funksjoner danner byggeklosser. Kan være sammensatte / kan settes sammen. Legger til rette for nedenfra-og-opp design og modularitet.
- Prosedyrer er data.

Ikke fokus på *ren* funksjonell programmering

- Hverken IN2040 eller SICP prediker rendyrket funksjonell programmering.
- Scheme er ikke et rent funksjonelt språk; inneholder også støtte for "destruktive operasjoner" og muterbare data.

Ikke fokus på *ren* funksjonell programmering

- Hverken IN2040 eller SICP prediker rendyrket funksjonell programmering.
- Scheme er ikke et rent funksjonelt språk; inneholder også støtte for "destruktive operasjoner" og muterbare data.
- Poenget er ikke at det ene eller andre paradigmet alltid er bedre enn det andre...
- men at du blir en bedre programmerer ved å kjenne til flere ulike teknikker og tenkemåter.

Spilleliste for uke 1 og fremover

- YouTube-kanalen til emnet vil ha en spilleliste for hver uke.
- Legger ut lenke på forelesningsplanen en uke i forveien.
- Resten av spillelista for uke 1 gir en kort intro til Scheme og Lisp!

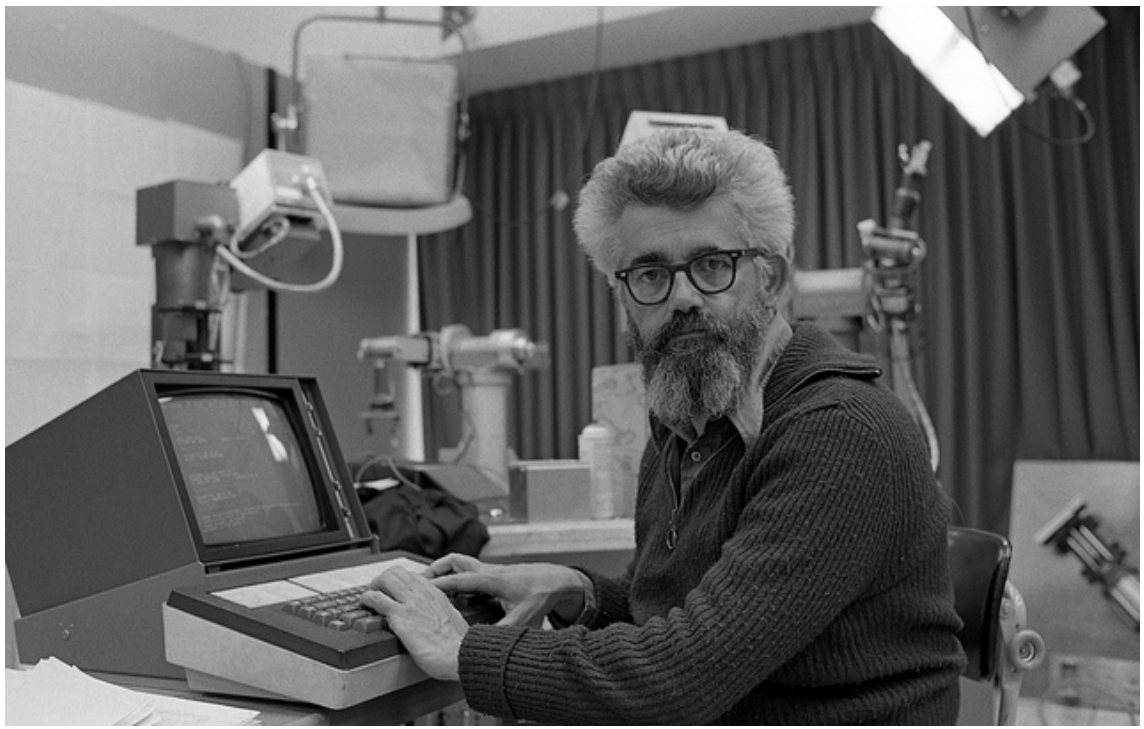
Scheme og Lisp

Lisp

- Kraftig høynivå-språk med lange tradisjoner.
- Som SICP er også Lisp i seg selv en klassiker.
- 'Oppdaget' av John McCarthy allerede i 1958.
  - Først bare ment som en matematisk formalisme (for rekursjonsteori og symbolsk algebra).
  - En av studentene hans, Steve Russell, implementerte så en interpreter for formalismen og dermed var programmeringsspråket Lisp født.
- Snarere enn at Lisp har blir utdatert har tendensen vært at andre språk stadig beveger seg nærmere Lisp (f.eks. `lambda` i Java og Python).

Lisp og AI

- Har hatt en tett kobling til forskningen på kunstig intelligens.
- Ofte beskrevet som "AI-språket".
- Bare delvis sant, men det er flere grunner til koblingen;
- AI: introdusert av McCarthy på midten av 50-tallet.
- Lisp: introdusert av McCarthy på midten av 50-tallet.

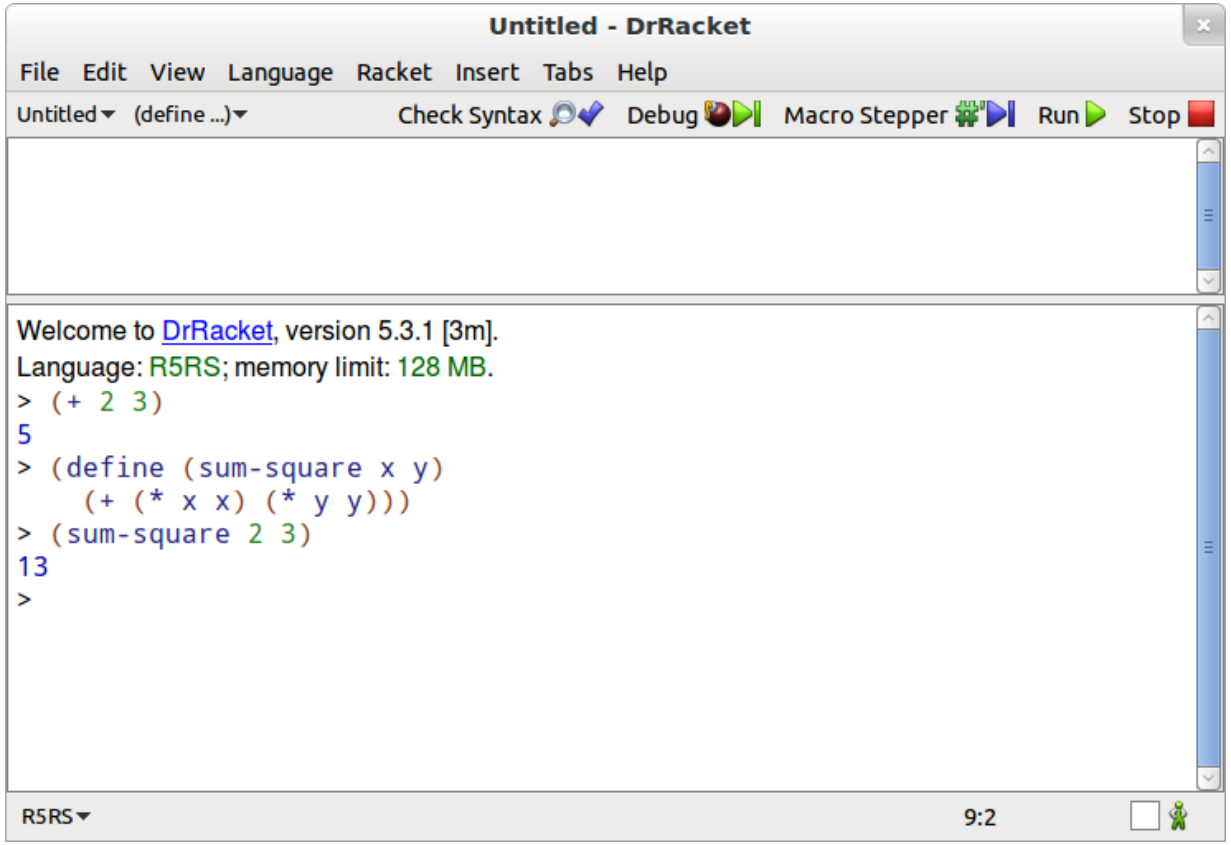


Koding i IN2040

- Programmeringspråk: Scheme
- Minimalistisk dialekt av Lisp.
- Andre dialekter: Common Lisp, Clojure, Emacs Lisp, Racket (dialekt av Scheme), etc.
- Lisp har en enkel semantikk, og en ekstremt enkel syntaks.
- Vi skal lære Scheme på noen minutter; og så bruke resten av kurset på å mestre noen av de avanserte teknikkene det støtter.
- Programmeringsomgivelse: DrRacket

DrRacket

- Se racket-lang.org
- Innstalert på alle IFI-maskiner.
- Viktig: DrRacket støtter flere standarder; sett språk til R^5RS.
- Se kurssidene for en innføring i bruk.



- I det nederste tekstområdet kan du teste og utvikle kode interaktivt:
- Skriv et uttrykk og trykk enter; uttrykket evalueres; resultatet printes:
- Såkalt read-eval-print loop eller REPL.

Syntaks og semantikk

Nå skal vi bruke noen minutter på å lære Scheme

- Vi tester noen uttrykk mot REPL'en.
- (= den interaktive Scheme-omgivelsen)
- Vi bruker ? for representere REPL-promptet og → for hva et uttrykk evalueres til.
- Datatyper som tall, strenger og boolske konstanter er eksempler på primitive / atomære uttrykk.
- Slike uttrykk evaluerer til seg selv.

Eksempler

```
? "dette er en streng"
→ "dette er en streng"
```

Nå skal vi bruke noen minutter på å lære Scheme

- Vi tester noen uttrykk mot REPL'en.
- (= den interaktive Scheme-omgivelsen)
- Vi bruker ? for representere REPL-promptet og → for hva et uttrykk evalueres til.
- Datatyper som tall, strenger og boolske konstanter er eksempler på primitive / atomære uttrykk.
- Slike uttrykk evaluerer til seg selv.

Eksempler

```
? "dette er en streng"
→ "dette er en streng"

? 42
→ 42
```

Nå skal vi bruke noen minutter på å lære Scheme

- Vi tester noen uttrykk mot REPL'en.
- (= den interaktive Scheme-omgivelsen)
- Vi bruker ? for representere REPL-promptet og → for hva et uttrykk evalueres til.
- Datatyper som tall, strenger og boolske konstanter er eksempler på primitive / atomære uttrykk.
- Slike uttrykk evaluerer til seg selv.

Eksempler

```
? "dette er en streng"
→ "dette er en streng"

? 42
→ 42

? 3.14159
→ 3.14159
```

Nå skal vi bruke noen minutter på å lære Scheme

- Vi tester noen uttrykk mot REPL'en.
- (= den interaktive Scheme-omgivelsen)
- Vi bruker ? for representere REPL-promptet og → for hva et uttrykk evalueres til.
- Datatyper som tall, strenger og boolske konstanter er eksempler på primitive / atomære uttrykk.
- Slike uttrykk evaluerer til seg selv.

Eksempler

```
? "dette er en streng"
→ "dette er en streng"

? 42
→ 42

? 3.14159
→ 3.14159

? #t
→ #t
```

Nå skal vi bruke noen minutter på å lære Scheme

- Vi tester noen uttrykk mot REPL'en.
- (= den interaktive Scheme-omgivelsen)
- Vi bruker ? for representere REPL-promptet og → for hva et uttrykk evalueres til.
- Datatyper som tall, strenger og boolske konstanter er eksempler på primitive / atomære uttrykk.
- Slike uttrykk evaluerer til seg selv.

Eksempler

```
? "dette er en streng"
→ "dette er en streng"

? 42
→ 42

? 3.14159
→ 3.14159

? #t
→ #t

? #f
→ #f
```

Prosedyrer og navngitte variabler

- "Parentesbasert prefiksnotasjon".
- Scheme-kode består av lister.
- Første element (prefikset) er operatoren (prosedyre).
- Resten av lista består av operandene (argumentene / parameterene).

Eksempler

```
? (+ 1 2)
→ 3
```

Prosedyrer og navngitte variabler

- "Parentesbasert prefiksnotasjon".
- Scheme-kode består av lister.
- Første element (prefikset) er operatoren (prosedyre).
- Resten av lista består av operandene (argumentene / parameterene).
- Lett å anvende en prosedyre på vilkårlig mange argumenter med prefiksnotasjon.

Eksempler

```
? (+ 1 2)
→ 3

? (+ 1 2 10 7 5)
→ 25
```

Prosedyrer og navngitte variabler

- "Parentesbasert prefiksnotasjon".
- Scheme-kode består av lister.
- Første element (prefikset) er operatoren (prosedyre).
- Resten av lista består av operandene (argumentene / parameterene).
- Lett å anvende en prosedyre på vilkårlig mange argumenter med prefiksnotasjon.
- `define`: er en såkalt *special form*, ikke en vanlig prosedyre. Brukes for å introdusere en leksikalsk variabel: symbol som bindes til en verdi.

Eksempler

```
? (+ 1 2)
→ 3

? (+ 1 2 10 7 5)
→ 25

? (define foo 42)
```

Prosedyrer og navngitte variabler

- "Parentesbasert prefiksnotasjon".
- Scheme-kode består av lister.
- Første element (prefikset) er operatoren (prosedyre).
- Resten av lista består av operandene (argumentene / parameterene).
- Lett å anvende en prosedyre på vilkårlig mange argumenter med prefiksnotasjon.
- `define`: er en såkalt *special form*, ikke en vanlig prosedyre. Brukes for å introdusere en leksikalsk variabel: symbol som bindes til en verdi.

Eksempler

```
? (+ 1 2)
→ 3

? (+ 1 2 10 7 5)
→ 25

? (define foo 42)

? foo
→ 42

? (+ foo 58)
→ 100

? (+ foo bar)
⇝ error; bar undefined
```

Sammensatte uttrykk

```
? (* 10 (+ foo 58))
→ 1000

? (/ (+ 10 20) 2)
→ 15
```

- Lister kan omslutte hverande for å bygge opp sammensatte uttrykk.
- Parentesene gir strukturen; ingen tvetydighet mht. (operator)presedens.

Sammensatte uttrykk

```
? (* 10 (+ foo 58))
→ 1000

? (/ (+ 10 20) 2)
→ 15

? (* (+ foo 58) (- (/ 8 2) 2))
→ 200
```

- Lister kan omslutte hverande for å bygge opp sammensatte uttrykk.
- Parentesene gir strukturen; ingen tvetydighet mht. (operator)presedens.

Sammensatte uttrykk

```
? (* 10 (+ foo 58))
→ 1000

? (/ (+ 10 20) 2)
→ 15

? (* (+ foo 58) (- (/ 8 2) 2))
→ 200

? (* (+ foo 58)
     (- (/ 8 2)
        2))
→ 200
```

- Lister kan omslutte hverande for å bygge opp sammensatte uttrykk.
- Parentesene gir strukturen; ingen tvetydighet mht. (operator)presedens.
- Ved lange uttrykk bruker vi indentering for å gjøre koden mer lesbar. (Bruk `TAB` i DrRacket / Emacs.)

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
?  (* (+ foo 58)
      (- (/ 8 2)
         2))

⇒ (* (+ 42 58)
      (- (/ 8 2)
         2))

⇒ (* 100
      (- (/ 8 2)
         2))
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))

⇒ (* 100 2)
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))

⇒ (* 100 2)

→ 200
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

- Vi har beskrevet evalueringen rekursivt; regelen kaller på seg selv (1c).

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))

⇒ (* 100 2)

→ 200
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

- Vi har beskrevet evalueringen rekursivt; regelen kaller på seg selv (1c).
- Scheme og Lisp har en enkel semantikk; alt oppsummeres med regelen over.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))

⇒ (* 100 2)

→ 200
```

Når Scheme leser sammensatte utrykk

Schemes evalueringsregel:

1) Evaluér enkelt-uttrykkene i sammensetningen
   a) Atomære uttrykk: evalueres til seg selv.
   b) Variabler: evalueres til verdien de refererer til.
   c) Sammensatte utrykk: anvend 1) igjen.
2) Anvend operatoren (det første uttrykket i lista) på verdiene til de andre uttrykkene.

- Vi har beskrevet evalueringen rekursivt; regelen kaller på seg selv (1c).
- Scheme og Lisp har en enkel semantikk; alt oppsummeres med regelen over.
- Unntaket er *special forms* (som `define`), som hver har sine egne evalueringsregler.

Eksempel

```
? (* (+ foo 58)
     (- (/ 8 2)
        2))

⇒ (* (+ 42 58)
     (- (/ 8 2)
        2))

⇒ (* 100
     (- (/ 8 2)
        2))

⇒ (* 100 (- 4 2))

⇒ (* 100 2)

→ 200
```

Oppsummering om datatypene vi har sett til nå

- Tall, “booleans”, og strenger er selv-evaluerende.

```
? 3.1415 → 3.1415
? "foo" → "foo"
? #t → #t
```

Oppsummering om datatypene vi har sett til nå

- Tall, “booleans”, og strenger er selv-evaluerende.
  - `? 3.1415` → `3.1415`
  - `? "foo"` → `"foo"`
  - `? #t` → `#t`
- Symboler evalueres til verdien de navngir (om noen):
  - `? foo` → `42`
  - `? +` → `#<procedure:+>`

Oppsummering om datatypene vi har sett til nå

- Tall, "booleans", og strenger er selv-evaluerende.
  - `? 3.1415` → `3.1415`
  - `? "foo"` → `"foo"`
  - `? #t` → `#t`
- Symboler evalueres til verdien de navngir (om noen):
  - `? foo` → `42`
  - `? +` → `#<procedure:+>`
- Lister er prosedyrekall. Det første elementet er en operator som kalles med *verdiene* til de resterende elementene som argumenter.
  - `? (* 10 (+ 2 3))` → `50`

Oppsummering om datatypene vi har sett til nå

- Tall, "booleans", og strenger er selv-evaluerende.
  - `? 3.1415` → `3.1415`
  - `? "foo"` → `"foo"`
  - `? #t` → `#t`
- Symboler evalueres til verdien de navngir (om noen):
  - `? foo` → `42`
  - `? +` → `#<procedure:+>`
- Lister er prosedyrekall. Det første elementet er en operator som kalles med *verdiene* til de resterende elementene som argumenter.
  - `? (* 10 (+ 2 3))` → `50`
- Men lister kan også være data:
  - `? (list 1 2 3)` → `(1 2 3)`
  - `? (list foo 2 "bar")` → `(42 2 "bar")`

Program = data

Kommentar / frampek

- I de første to forelesningen skal vi jobbe med enkle matematiske eksempler, men senere i kurset kommer vi til å jobbe masse med lister og symbolske data.
- Lister har en spesiell rolle i Lisp og Scheme: kan brukes til å representere både data og program.
- LISP = LISt Processing
- Lister = data = program.

Opprette prosedyrer

Å opprette egne prosedyrer med `define`

- Vi har sett at `define` kan brukes for å binde leksikalske variabler til en verdi. Men verdien kan også være en prosedyre.
- Prosedyrer lages ved såkalte lambda-uttrykk.

Å opprette egne prosedyrer med `define`

- Vi har sett at `define` kan brukes for å binde leksikalske variabler til en verdi. Men verdien kan også være en prosedyre.
- Prosedyrer lages ved såkalte lambda-uttrykk.

Generell form

```
(define ⟨navn⟩
  (lambda (⟨argumenter⟩)
    ⟨kropp⟩))
```

Eksempel

```
? (define snitt
    (lambda (x y)
      (/ (+ x y) 2)))
```

Å opprette egne prosedyrer med `define`

- Vi har sett at `define` kan brukes for å binde leksikalske variabler til en verdi. Men verdien kan også være en prosedyre.
- Prosedyrer lages ved såkalte lambda-uttrykk.

Generell form

```
(define ⟨navn⟩
  (lambda (⟨argumenter⟩)
    ⟨kropp⟩))
```

Eksempel

```
? (define snitt
    (lambda (x y)
      (/ (+ x y) 2)))

? (snitt 10 20) → 15

? (snitt (snitt 10 20)
         (snitt 10 30))
→ 17.5
```

Å opprette egne prosedyrer med `define`

- Vi har sett at `define` kan brukes for å binde leksikalske variabler til en verdi. Men verdien kan også være en prosedyre.
- Prosedyrer lages ved såkalte lambda-uttrykk.

Generell form

```
(define ⟨navn⟩
  (lambda (⟨argumenter⟩)
    ⟨kropp⟩))
```

- Det finnes også en kortform som sløyfer `lambda`:

```
(define (⟨navn⟩ ⟨argumenter⟩)
  ⟨kropp⟩)
```

Eksempel

```
? (define snitt
    (lambda (x y)
      (/ (+ x y) 2)))

? (snitt 10 20) → 15

? (snitt (snitt 10 20)
         (snitt 10 30))
→ 17.5

? (define (snitt x y)
    (/ (+ x y) 2))
```

Predikater og kondisjonale uttrykk

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) →
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.
- De logiske konnektivene `and` og `or` er *special forms*.
- Sammen med kondisjonale uttrykk som f.eks `if` og `cond` kan vi bruke predikater og konnektiver til å kontrollere flyten i et program.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f

? (and (< 3 4)
       foo)
→
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.
- De logiske konnektivene `and` og `or` er *special forms*.
- Sammen med kondisjonale uttrykk som f.eks `if` og `cond` kan vi bruke predikater og konnektiver til å kontrollere flyten i et program.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f

? (and (< 3 4)
       foo)
→ 42
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.
- De logiske konnektivene `and` og `or` er *special forms*.
- Sammen med kondisjonale uttrykk som f.eks `if` og `cond` kan vi bruke predikater og konnektiver til å kontrollere flyten i et program.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f

? (and (< 3 4)
       foo)
→ 42

? (or foo
      (negative?  foo))
→
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.
- De logiske konnektivene `and` og `or` er *special forms*.
- Sammen med kondisjonale uttrykk som f.eks `if` og `cond` kan vi bruke predikater og konnektiver til å kontrollere flyten i et program.

Predikater og kondisjonale uttrykk

Eksempler

```
? (define foo 42)

? (even?  foo) → #t

? (zero?  foo) →  #f

? (not (>= foo 42)) →  #f

? (and (< 3 4)
       foo)
→ 42

? (or foo
      (negative?  foo))
→ 42
```

- Predikater er prosedyrer eller uttrykk som returnerer sant / usant (`#t` / `#f`).
- F.eks `even?`, `not`, `>`, `=`, etc.
- Husk at alt unntatt `#f` teller som sant.
- De logiske konnektivene `and` og `or` er *special forms*.
- Sammen med kondisjonale uttrykk som f.eks `if` og `cond` kan vi bruke predikater og konnektiver til å kontrollere flyten i et program.

Predikater og kondisjonale uttrykk (forts.)

Eksempler

```
? (if (number?  42)
      "number"
      "something else")
```

$\rightarrow$ *"number"*

Generell form

(if ⟨*test*⟩
    ⟨*utfall-hvis-sant*⟩
    ⟨*utfall-hvis-usant*⟩)

Predikater og kondisjonale uttrykk (forts.)

Eksempler

```
? (if (number?  42)
      "number"
      "something else")
```

→ *"number"*

```
? (cond ((< 42 3) "less")
        ((> 42 3) "greater")
        (else "equal"))
```

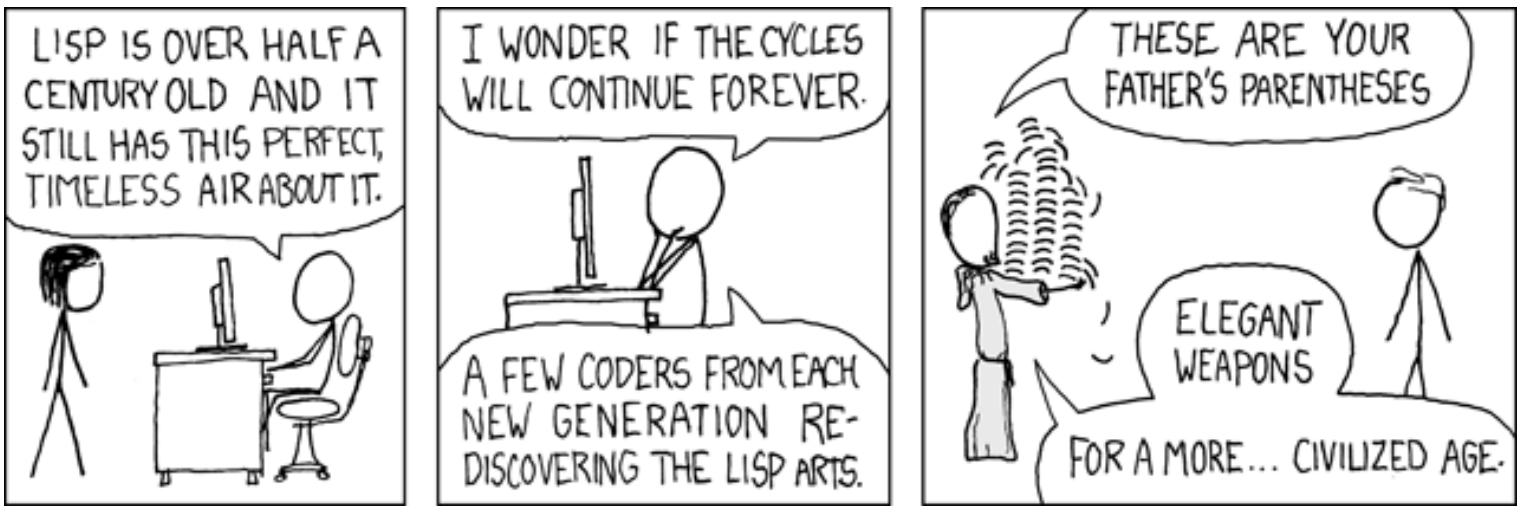
→ *"greater"*

Generell form

(if ⟨*test*⟩
    ⟨*utfall-hvis-sant*⟩
    ⟨*utfall-hvis-usant*⟩)

(cond (⟨$test_1$⟩ ⟨$utfall_1$⟩)
      (⟨$test_2$⟩ ⟨$utfall_2$⟩)
      (⟨$test_n$⟩ ⟨$utfall_n$⟩)
      (else ⟨*utfall*⟩))

xkcd on Lisp



http://xkcd.com/297/