Komma igång med Allegro Common Lisp

Första gången

Det Lisp-system som vi i kommer att använda för laborationerna heter Allegro Common Lisp. Det är en kommersiell programvara från företaget Franz Inc. som är en av de största tillverkarna av utvecklingsmiljöer för Lisp. Allegro körs inuti texteditorn Emacs som en integrerad del. Vi använder för närvarande version 8.0 av Allegro Common Lisp.

Via kursens webbsidor kan du komma åt en manual för Common Lisp. Där finns alla funktioner och variabler dokumenterade och det kan vara bra att ha en webbläsare med denna sida uppslagen bredvid ditt Emacs-fönster som referens.

För att få tillgång till Allegro när du sitter inloggad på IDA:s Sun-stationer måste du lägga till modulen `prog/allegro`. Detta gör du genom att öppna ett skalfönster och skriva följande:

```
zaza7 <14> module initadd prog/allegro
zaza7 <15> module add prog/allegro
```

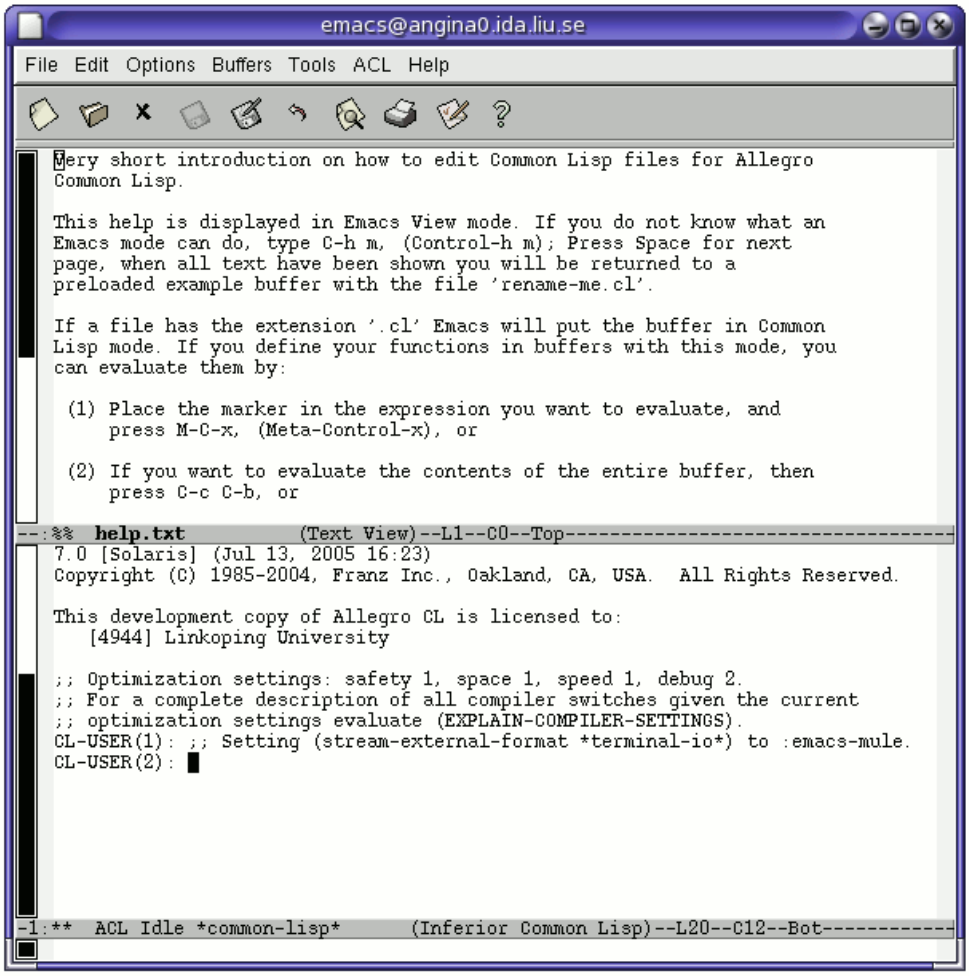
Den första raden lägger till modulen i dina uppstartfiler så att den finns med nästa gång du loggar in. Den andra raden lägger till modulen till det aktuella skalfönstret så att du kan köra Allegro utan att logga ut och logga in igen. Vid nästa labbtillfälle behöver du inte lägga till modulen igen, utan kan starta Allegro direkt. (Observera att det endast är texten i fetstil som du ska skriva in. Övrig text är ett exempel på hur prompten i skalfönstret kan se ut.)

Du kan starta Allegro på två olika sätt, antingen från ett skalfönster genom att skriva

```
zaza7 <16> emacs-allegro-cl
```

eller genom att välja motsvarande alternativ i menyerna i skrivbordsmiljön. Det senare alternativet fungerar dock inte första gången, eftersom du måste logga ut och logga in igen för att menyerna ska genereras om.

När du startar Allegro visas ett delat Emacs-fönster. I den nedre bufferten **`*common-lisp*`** körs Allegro och här kan du vid Lisp-prompten skriva in Lisp-uttryck som beräknas när du trycker Enter. Den övre bufferten används normalt för att editera källkod. Från början visas dock en hjälptext. Denna kan du få bort genom att, när markören är i den övre bufferten, trycka **`q`**. Figuren på nästa sida visar hur Allegro ser ut vid uppstart.


emacs@angina0.ida.liu.se
File Edit Options Buffers Tools ACL Help
Very short introduction on how to edit Common Lisp files for Allegro
Common Lisp.
This help is displayed in Emacs View mode. If you do not know what an
Emacs mode can do, type C-h m, (Control-h m); Press Space for next
page, when all text have been shown you will be returned to a
preloaded example buffer with the file 'rename-me.cl'.
If a file has the extension '.cl' Emacs will put the buffer in Common
Lisp mode. If you define your functions in buffers with this mode, you
can evaluate them by:
(1) Place the marker in the expression you want to evaluate, and
press M-C-x, (Meta-Control-x), or
(2) If you want to evaluate the contents of the entire buffer, then
press C-c C-b, or
--:%% help.txt (Text View)--L1--C0--Top
7.0 [Solaris] (Jul 13, 2005 16:23)
Copyright (C) 1985-2004, Franz Inc., Oakland, CA, USA. All Rights Reserved.
This development copy of Allegro CL is licensed to:
[4944] Linkoping University
;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the current
;; optimization settings evaluate (EXPLAIN-COMPILER-SETTINGS).
CL-USER(1): ;; Setting (stream-external-format *terminal-io*) to :emacs-mule.
CL-USER(2):
-1:** ACL Idle *common-lisp* (Inferior Common Lisp)--L20--C12--Bot

Laboration 1
Komma igång med programmering

1.1. Fokus

Syftet med denna första laborationsomgång är att introducera programmering i allmänhet och programspråket Lisp i synnerhet. I början är det extra viktigt att du lägger ner mycket tid på att göra övningar för att snabbt skaffa dig en uppfattning om hur Lisp fungerar. Programmering är ett hantverk som man lär sig bäst genom att skriva många program, snarare än genom att enbart läsa en bok.

Du bör försöka göra så många som möjligt av de övningar som finns i laborationen. För att bli godkänd ska du dels skriftligen redovisa de två uppgifter som finns i slutet av laborationen, dels genomföra den dugga (kortare skriftligt prov) som ges vid det redovisningstillfälle som finns angivet i schemat.

Till denna laboration hör kapitel 1-5, samt avsnitt 6.1.1-6.1.6 i läroboken.

1.3. Uppmjukning

Övning 101: Läs kapitel 2 Uttryck och funktioner i läroboken. Skriv sedan in följande uttryck vid Lisp-prompten och försök att i förväg räkna ut vad Lisp-tolken kommer att svara. Gör gärna om uttrycket till infix-form först om det känns lättare.

```
12
-12
1/3                     Bråk
(+ 12 45)
(+ 12 45 78)            Godtyckligt antal argument
(+ 12 (- 45))           Unärt minus
(+ 12 (- 45 78))        Binärt minus
(1+ (* 2 3))
(1- (+ 2 3))
(/ 35 13)
(/ 35.0 13.0)
(/ 35 5)                Heltal
(/ 34 5)                Bråk
(float (/ 34 5))        Flyttal
(- (/ 1 (/ (+ 2 3) (* 4 5))) 6)
```

1.4. Infixform till prefixform

Övning 102: Nu ska vi öva på omvandling åt andra hållet. Gör om följande aritmetiska uttryck från infixform till prefixform. Skriv sedan in uttrycken vid Lisp-prompten och kontrollera att resultatet blir rätt.

Infixform (normal)	Prefixform (Lisp)
1+2·3	
(1+2)·3	
2·4+(6-8)	

$$\frac{5 + (-3 + 4)}{6 + \frac{2}{5}}$$

1.5. Definition av funktioner

I de första två övningarna har vi använt Lisp ungefär som en avancerad miniräknare. Nu är det dags att införa egna funktioner med hjälp av konstruktionen `defun`. Du kan definiera funktioner på två sätt i Allegro. Antingen kan du definiera en eller flera funktioner i den övre bufferten och evaluera alla med tangentkombinationen `C-c C-b`, eller så kan du definiera en funktion i taget genom att skriva in dem vid prompten i den undre bufferten.

Övning 103: Definiera en funktion `medelvärde` med två argument `x` och `y` som returnerar medelvärdet av x och y.

```
CL-USER(1): (medelvärde 3 7)
5
CL-USER(2): (medelvärde 1 2)
3/2
```

Övning 104: Definiera en funktion `störst` med två argument `x` och `y` som returnerar det största av argumenten. Det finns redan en sådan funktion i Lisp som heter `max`, men tanken är givetvis att du ska skriva en egen funktion utan att använda dig av `max`.

```
CL-USER(3): (störst 3 7)
7
```

Övning 105: Definiera en funktion `störst-av-3` som tar tre argument och returnerar det största talet av dessa tre. Tips: Använd dig av funktionen `störst`.

```
CL-USER(4): (störst-av-3 10 -3 8)
10
```

1.6. Manipulering av tal

För att avrunda ett flyttal till ett heltal kan man använda funktionen `truncate`. Funktionen avrundar egentligen inte, utan delar bara upp flyttalet i två delar: en heltalsdel och en decimaldel. Om du använder `truncate` vid Lisp-prompten skrivs två värden ut, men det är bara det första som används om du bakar in `truncate` i ett annat uttryck.[1]

```
CL-USER(4): (truncate 10.7)
10
0.6999998
CL-USER(5): (+ (truncate 5.4) (truncate 6.7) 10)
21
```

Av exemplet ovan kan man dra slutsatsen att Lisp i normalfallet inte har särskilt hög precision för lagring av flyttal. Decimaldelen av 10,7 borde ju vara 0,7.

[1] Funktionen `truncate` returnerar faktiskt två värden. Du kan fånga upp båda dessa med att använda t.ex. `multiple-value-bind` (se s. 270 i läroboken). Detta tas upp senare i kursen.

För att göra mer riktig avrundning kan du använda funktionen `round`. Den avrundar ett flyttal till närmaste heltal. Om flyttalet är mitt emellan två heltal avrundas det till det närmaste jämna heltalet. Precis som för `truncate` returneras två värden, men bara det första används.

```
CL-USER(6): (round 10.4)
10
0.39999962
CL-USER(7): (round 10.5)
10
0.5
CL-USER(8): (round -10.9)
-11
0.10000038
```

För att få fram resten vid en heltalsdivision kan vi använda funktionen `mod`.

```
CL-USER(9): (mod 10 3)
1
CL-USER(10): (mod 25 7)
4
```

Om man försöker dela 10 med 3 får man resultatet 3 med resten 1. Om man försöker dela 25 med 7 får man resultatet 3 med resten 4.

Övning 106: Definiera en egen variant av funktionen `mod` och kalla den `my-mod`. Din funktion ska funka likadant som orginalet, men ska givetvis inte använda mod, utan räkna ut resten med hjälp av andra funktioner vi använt tidigare.

Övning 107: Vi vill kunna hantera de enskilda siffrorna i ett godtyckligt positivt heltal. Därför ska du skriva tre hjälpfunktioner som ska heta `sista-siffran`, `utan-sista-siffran` och `lägg-till-siffra-sist`. Funktionen sista-siffran ska returnera den sista siffran i talet (entalssiffran) och funktionen utan-sista-siffran ska returnera talet utan den sista siffran. Funktionen lägg-till-siffra-sist ska returnera ett nytt tal där ett gammalt tal slagits ihop med en siffra.

```
CL-USER(15): (sista-siffran 123)
3
CL-USER(16): (utan-sista-siffran 123)
12
CL-USER(17): (lägg-till-siffra-sist 5 123)
1235
```

Övning 108: Med hjälp av de tre funktionerna från föregående övning ska du konstruera en funktion `byt-plats-sista` som byter plats på de två sista siffrorna i ett tal.

```
CL-USER(18): (byt-plats-sista 1234)
1243
```

Övning 109: Med hjälp av de tre funktionerna från övning 107 ska du konstruera funktionerna `ental`, `tiotal` och `hundratal` som plockar fram motsvarande siffror ur ett tal.

1.7. Rekursion

Läs kapitel 3 Rekursion i läroboken. Två viktiga begrepp införs, *rekursiv* och *iterativ process-lösning*.

Övning 110: Definiera funktionen ”upphöjt till”. Kalla den `upphöjt-till` och låt den ta två heltalsargument. Definiera två varianter av funktionen, en som utför en rekursiv processlösning och en som utför en iterativ processlösning. För att förstå hur beräkningen genomförs kan du i huvudet eller på papper använda substitutionsmodellen (se läroboken avsnitt 2.10) eller använda kommandot `trace` för att spåra körningen (se Allegro-manualen i detta häfte, s.14). Det finns redan en ”upphöjt till”-funktion i Lisp som heter `expt`, men du ska givetvis inte använda den.

```
CL-USER(19): (upphöjt-till 2 3)
8
CL-USER(20): (upphöjt-till 5 2)
25
```

Övning 111: Definiera en funktion `summera-första` som adderar ihop de första *n* heltalen. Gör två versioner, en med rekursiv och en med iterativ processlösning. Summan av de *n* första heltalen kan beräknas med formeln $n(n+1)/2$ men i den här övningen vill vi att du summerar talen ett i taget med hjälp av rekursion.

```
CL-USER(21): (summera-första 6)
21
CL-USER(22): (summera-första 0)
0
```

Om du istället vill multiplicera de *n* första heltalen, vad behöver du ändra i funktionerna?

Övning 112: Skriv en rekursiv funktion som räknar antalet siffror i ett positivt heltal. Använd gärna hjälpfunktionerna från övning 107. Skriv också en funktion som räknar ut summan av de ingående siffrorna i ett positivt heltal. Funktionerna ska uppvisa rekursiv processlösning.

```
CL-USER(23): (antal-siffror 1234)
4
CL-USER(24): (summera-siffror 1234)
10
```

Övning 113: Skriv en rekursiv funktion som vänder på ett positivt heltal, dvs returnerar ett nytt tal där siffrorna är i omvänd ordning. Funktionen ska uppvisa iterativ processlösning.

```
CL-USER(25): (vänd-på-siffror 1234)
4321
```

Övning 114*: För att bearbeta enskilda siffror i ett tal har vi i övning 107 skapat hjälpfunktioner som går igenom talet sekvensiellt bakifrån. Definiera nu funktioner så att vi går igenom talet framifrån istället. De tre funktionerna ska fungera enligt nedanstående exempel:

```
CL-USER(26): (första-siffran 123)
1
CL-USER(27): (utan-första-siffran 123)
23
CL-USER(28): (lägg-till-siffra-först 5 123)
5123
```

Övning 115*: Om du vill lösa en lite större uppgift kan du ta dig an övning 2.8.E ur läroboken. Skriv en funktion `dagar` som räknar ut antalet dagar mellan två datum. Vi anger ett datum som ett heltal på formen *yyyymmdd*. Använd hjälpfunktioner för att ur ett sådant åttasiffrigt tal hämta delarna, dvs år, månad och dag.

```
CL-USER(29): (dagar 19931215 19940110)
26
```

Ett tips kan vara att räkna ut antalet dagar från 1900-01-01 för båda datumen och sedan ta absolutbeloppet på skillnaden.

Övning 116*: Lös uppgift 3.3.D i läroboken. Innan du tittar på lösningsförslaget, fundera på problemet. Skriv in funktionerna och testa för att komma underfund med vad som händer. Om du har tidigare erfarenheter av programmering, jämför med andra språk. Är det samma problem där?

1.8. Symboliska uttryck

Dessa övningar berör kapitel 4 Listor och symboler, kapitel 5 Lokala variabler och kapitel 6 Sekvenser och träd i läroboken. Här införs de viktigaste datatyperna i Lisp, symboler och listor. Dessutom införs quote eftersom både program och data representeras som listor. I kapitel 4 finns en stor mängd fördefinierade funktioner. I kapitel 6 beskrivs olika rekursiva mallar för bearbetning av listor och binära träd.

Övning 117: Pröva att skriva in följande Lisp-uttryck. Fundera i förväg ut vad Lisp-systemet kommer att svara.

```
(setq a 123)
a
(quote a)
'a
(+ a 17)
'(+ 1 17)
(quote (+ 1 17))
(setq a 'b)
a
'(hej hopp ditt feta nylle)
'(quote x)
''x
"Kalle Pettersson"
'Kalle\ Pettersson
'|Kalle Pettersson|
(cdr (assoc 'ett '((ett . 1) (två . 2) (tre . 3))))
```

På kursens webbsidor finns en exempelsamling med ytterligare övningar. I del B kan du hitta fler övningar kring symboliska uttryck.

1.9. Grundläggande Lisp-funktioner

Titta igenom sammanställningen av Lisp-funktioner som finns i avsnitten 4.6 och 4.10 i läroboken. Sök även efter funktionerna i on line-manualen för Lisp som du kan hitta via kursens webbsidor.

Övning 118: Vad blir resultaten om man skriver in följande Lisp-uttryck? Fundera i förväg ut vad du tror att Lisp-systemet kommer att svara och testa sedan.

```
(first '(one two three))
(car '(one two three))
(rest (first '((one two) three four)))
(cdr (car '((one two) three four)))
(cdar '((one two) three four)))

(second '(one two three))
(cadr '(one two three))
(cons '(eva lisa) '(karl sven))
(list '(eva lisa) '(karl sven))
(append '(eva lisa) '(karl sven))

(remove 'sven '(eva sven lisa sven anna))
(subst 'gittan 'birgitta '(eva birgitta lisa birgitta))
(make-list 5)
(make-list 5 :initial-element 'start)
(copy-list '(a new list))

(intersection '(a b c) '(x b z c))
(union '(a b c) '(x b z c))
(last '(karl adam nilsson))
(butlast '(karl adam nilsson))
(butlast '(a b c d e) 3)
(nthcdr 2 '(a b c d e))
(nth 2 '(a b c d e))
(assoc 'tre '((ett . one) (två . two) (tre . three)))
```

Övning 119: Vi har en lista som innehåller information om en familj:

```
CL-USER(30): (setq familj '((far karl) (mor eva) (barn lena sven lisa)))
((FAR KARL) (MOR EVA) (BARN LENA SVEN LISA))
```

Skriv Lisp-uttryck som med hjälp av grundläggande listfunktioner använder familj för att utföra följande:

- tar fram faderns namn
- skapar en lista med faderns och moderns namn
- tar fram en lista med barnens namn
- skapar en ny lista med barnen först och fadern sist
- skapar en ny lista med ytterligare ett barn peter som läggs sist bland barnen

Övning 120: Vi antar att vi har listor med information om familjer enligt föregående övning. Definiera följande funktioner:

- en funktion `skapa-familj` som tar tre argument: namn på fader, namn på moder och en lista med namn på barnen
- en funktion `barn?` som undersöker om ett barn med ett givet namn finns i familjen

```
CL-USER(31): (setq familj (skapa-familj 'bosse 'karin '(bertil beata)))
((FAR BOSSE) (MOR KARIN) (BARN BERTIL BEATA))
CL-USER(32): (barn? familj 'kalle)
NIL
```

1.10. Sekventiell bearbetning

Sekventiell bearbetning, dvs att gå igenom en lista element för element, är ett viktig område som du bör lägga ner mycket tid på. Lärobokens avsnitt 6.1 innehåller mallar för sekventiell bearbetning, samt ett flertal exempel och övningar. Utför substitutionsmodellen på några exempel för att förstå rekursionen bättre. Vid testning på dator kan du använda spårningsfunktionen `trace` för att följa de successiva rekursiva anropen med värdena på parametrarna och de beräknade värdena.

Övning 121: Skriv en funktion `summera-endast-tal` som summerar de element på en lista som är tal.

```
CL-USER(33): (summera-endast-tal '(a 1 b 2 ((b 4) 2) 3))
6
```

Övning 122: Skriv en funktion `finns-bokstav?` som returnerar ett sant värde (`T`) om en given bokstav finns i ett ord, annars ett falskt värde (`NIL`). Ordet representeras som en lista av symboler. Det finns en liknande funktion i Lisp redan som heter `member`, så givetvis ska du inte använda dig av den utan själv bearbeta listan tecken för tecken.

```
CL-USER(34): (finns-bokstav? 'u '(h u s))
T
CL-USER(35): (finns-bokstav? 'a '(b i l))
NIL
```

Övning 123: Skriv en funktion `ta-bort-vokaler` som tar en lista med bokstäver (symboler) och returnerar en ny lista med vokalerna borttagna. Det kanske går lättare om du föreställer dig att du ska skapa en ny lista med endast konsonanter snarare än att ta bort vokalerna. Tips: Använd dig av `member` eller `finns-bokstav?` som du definierat ovan!

```
CL-USER(36): (ta-bort-vokaler '(b i r g i t t a))
(B R G T T)
```

I del C exempelsamlingen på kurshemsidan finns ytterligare övningar på sekvensiell bearbetning.

Övning 124*: Definiera mängdoperationerna union, snitt och differens. Kalla funktionerna `min-union` (eftersom funktionen `union` redan finns i Lisp), `snitt` och `differens`. Definiera också funktionen `lägg-till` som lägger till ett element till en mängd, samt en funktion `medlem?` som undersöker om ett element är medlem i en mängd. Låt mängderna representeras som listor av symboler.

Som exempel kan vi skapa tre mängder med personer som går i ettan, personer som är intresserade av fotboll samt personer som är intresserade av teater.

```
CL-USER(37): (setq ettor '(sven lisa peter anna karl))
(SVEN LISA PETER ANNA KARL)
CL-USER(38): (setq fotboll '(lisa eva karl sven))
(LISA EVA KARL SVEN)
CL-USER(39): (setq teater '(karin lisa peter nisse eva))
(KARIN LISA PETER NISSE EVA)
```

Med hjälp av funktionen `min-union` kan vi nu bilda mängden av alla personer i de tre delmängderna. Notera att ingen person är med två gånger, eftersom det ju är en mängd. Personernas ordning spelar heller ingen roll.

```
CL-USER(40): (min-union ettor (min-union fotboll teater))
(ANNA LISA EVA KARL SVEN KARIN PETER NISSE)
```

Ta fram mängden av alla ettor som är intresserade av fotboll och lägg till Tomas.

```
CL-USER(41): (lägg-till 'tomas (snitt ettor fotboll))
(TOMAS SVEN LISA KARL)
```

Ta fram mängden av alla ettor som är intresserade av fotboll eller teater.

```
CL-USER(42): (snitt ettor (min-union fotboll teater)
(SVEN LISA PETER KARL)
```

Ta reda på om Eva inte är en etta, men är intresserad av åtminstone någotdera av fotboll och teater.

```
CL-USER(43): (medlem? 'eva (differens (min-union fotboll
                                                 teater)
                                       ettor))
T
```

1.11. Sammanfattning

Innan du går över till att lösa uppgifterna i den här laborationen, fundera över om du kan svara på följande frågor. Sidorna inom parentes hänvisar till läroboken.

- Vad menas med ett uttryck i Lisp? (s. 13-15)
- Vad innebär det att en funktion är rekursiv? (s. 36)
- Vad är skillnaden mellan rekursiv och iterativ processlöning? (s. 39-42))
- Vad är en datatyp och vilka datatyper i Lisp har vi hittills stött på? (s. 47)
- Vilken roll har quote-tecknet och när ska det användas? (s. 55-59)
- Vad är skillnaden mellan `cons`, `list` och `append`? (s. 53, 60)

I detta inledande avsnitt har vi stött på många nya funktioner i Lisp. Fundera över om du har förstått ungefär vad följande urval av de viktigaste funktionerna gör.

- Matematiska funktioner: `+`, `-`, `*`, `/`, `1+`, `1-`, `>`, `<`, `=`
- Styrstrukturer: `if`, `cond`
- Skapa listor: `cons`, `list`, `append`
- Undersöka listor: `first`, `rest`, `car`, `cdr`, `second`, `endp`, `last`, `butlast`, `assoc`
- Undersöka objekt i allmänhet: `atom`, `eq`, `numberp`, `symbolp`
- Funktioner vid Lisp-prompten: `trace`, `setq`

Uppgift 1A - Rekursiva funktioner

Lös minst sex av nedanstående åtta deluppgifter. Lös minst en av de första två som behandlar siffrorna i ett tal. (Återanvänd hjälpfunktionerna från övning 107.) Minst två av dina funktioner ska uppvisa en iterativ processlösning och minst två rekursiv processlösning.

1. Skriv en funktion `räkna-siffra` som räknar antalet förekomster av en viss siffra i ett positivt heltal.
2. Skriv en funktion `udda-siffror` som i ett positivt heltal tar bort alla jämna tal (så att t.ex. 1234 blir 13).
3. Skriv en funktion `räkna` som räknar antalet förkomster av en viss symbol i en lista.
4. Skriv en funktion `filtrera` som från en lista med godtyckligt innehåll skapar en ny lista med enbart de element som är tal.
5. Skriv en funktion `udda` som från en lista med enbart tal skapar en ny lista med enbart de talen som är udda
6. Skriv en funktion `summera` som tar en lista med symbolerna `ett`, `två`, ..., `nio` och summerar dessa som om de vore siffror. Antag att enbart sådana symboler som kan hanteras finns med i listan.
7. Skriv en funktion `positiva` som tar en lista med tal som kan vara både positiva och negativa och som skapar en ny lista som är likadan, fast utan minustecken. Alla negativa tal ska ha blivit utbytta mot sina absolutvärden.
8. Skriv en funktion `kvadratsumma` som summerar kvadraterna av de *n* första udda talen.

```
CL-USER(1): (räkna-siffra 1 12315)
2
CL-USER(2): (udda-siffror 123456)
135
CL-USER(1): (räkna 'ett '(ett noll noll ett ett noll))
3
CL-USER(2): (filtrera '(ett 2 tre 4 (5 6 fyra fem) 4711 pelle))
(2 4 4711)
CL-USER(3): (udda '(1 2 3 4 5 17 42 4711))
(1 3 5 17 4711)
CL-USER(4): (summera '(ett fyra sju två))
14
CL-USER(5): (positiva '(5 8 -13 -2 4711 -42))
(5 8 13 2 4711 42)
CL-USER(6): (kvadratsumma 4)
84
```

Samtliga funktioner ska vara rekursiva. Det finns inbyggda funktioner i Lisp som direkt löser flera av problemen ovan, men i denna uppgift ska du givetvis inte använda dem.

Redovisa all programkod. Dessutom ska du för varje deluppgift bifoga körexempel med så många testfall att du kan garantera att alla alternativ i dina funktioner testas.

Ange för var och en av deluppgifterna om din funktion uppvisar en rekursiv eller iterativ processlösning. Motivera också varför det är så.

Uppgift 1B - Personnummer

Skriv en funktion `kontrollera-pnr` som tar ett personnummer i form av en lista med tio siffror på formen (*å å m m d d x x x k*) och kontrollerar att kontrollsiffran, den sista siffran, är korrekt.

Dela upp uppgiften i flera mindre funktioner som du kan testköra för sig. Funktionerna måste vara rekursiva om de utför någon form av repetition.

Metoden för att räkna ut kontrollsiffran i personnummer är enligt följande. Multiplicera alla siffror utom den sista med omväxlande 2 och 1. Summera de enskilda siffrorna i produkterna (dvs om produkten är $2 \cdot 5 = 10$ så summerar vi 1 och 0). Kontrollsiffran är skillnaden mellan nästa högre (eller samma) tiotal och summan av siffrorna. Exempel på manuell uträkning:

```
Personnummer:  7 2 0 1 2 3 1 2 3
Viktsiffror:   2 1 2 1 2 1 2 1 2
-------------------------------------
Produkt:      14 2 0 1 4 3 2 2 6
Siffersumma: 1+4+2+0+1+4+3+2+2+6 = 25

Närmast högre tiotal: 30
Kontrollsiffra: 30-25 = 5
```

Exempel på användning av funktionen:

```
CL-USER(1): (kontrollera-pnr '(7 2 0 1 2 3 1 2 3 5))
T
```

Testa din funktion med olika personnummer, särskilt personnummer där kontrollsiffran är noll. Fungerar det även då?

Redovisa all programkod. Bifoga körexempel som visar att funktionen kan identifiera både korrekta och felaktiga personnummer. Testa särskilt med personnummer som har noll som kontrollsiffra.

Hur ska man redovisa?
Läs igenom de allmänna anvisningarna i början av labbhäftet!