Prova på-laboration i Common Lisp

Johan Sjöholm
johsj@ida.liu.se
Institutionen för datavetenskap, Linköpings universitet
2009-07-28

1. Introduktion till funktionella språk

Programmeringsspråket Lisp konstruerades redan 1959 vilket gör det till ett av de äldsta programmeringsspråken och är, tillsammans med Fortran, ett av de äldsta som fortfarande används på någorlunda stor skala. Lisp var redan från början tänkt att användas för forskning inom artificiell intelligens och det är också här det har rönt störst framgång. Lisp står för LISt Processing language och är speciellt bra på att hantera symboler och, som namnet antyder, för att hantera listor. Lisp har utvecklats åt flera håll och gett upphov till många dialekter med olika tillämpningsområden. Några av de mer kända är Emacs Lisp (elisp), konfigurations- och scriptspråket för Emacs, AutoLisp, motsvarande elisp för AutoCAD, INTERLISP, som var speciellt populärt på hårdvarumässigt specialiserade lispmaskiner på 70- och 80-talen. De två största lispdialekterna idag är Scheme, en minimalistisk Lisp-dialekt skapad av Guy Steele och Gerald Sussman på 70-talet, och Common Lisp, en större, och enligt vissa mer lättarbetad, dialekt skapad 1984 av en kommité ledd av samme Guy Steele som låg bakom Scheme. Common Lisp är den dialekt som kommer användas i den här laborationen och det är den som avses när vi hädanefter skriver Lisp.

Lisp är också, åter igen till skillnad från Ada, dynamiskt typat. Det innebär att man inte behöver bry sig om vilka typer olika variabler eller parametrar har. Det underlättar när man programmerar men kan ibland leda till konstiga buggar när variabler är av fel typ, i den här labben ska det dock bara göra saker lättare.

Funktionell programmering

Lisp är ett så kallat multiparadigmspråk, det vill säga det har stöd för många olika sätt att programmera. Dock är det speciellt lämpat för så kallad funktionell programmering varför vi använder det som exempelspråk i den här laborationen. Funktionell programmering skiljer sig från den imperativa programmeringen ni provade på i Ada-labben på ett antal punkter. I Ada bygger man upp program genom att säga åt datorn vad den ska göra med olika data lagrade i minnet. Den uppsättning data som finns lagrad i programmets olika variabler vid en viss tidpunkt kallas programmets state. Vad ett imperativt program gör är alltså att påverka (förändra) och påverkas av programmets state. I funktionell programmering undviker man state och ser snarare programmet som en serie matematiska funktioner som anropar varandra utan att påverka någon extern data. En funktion som inte beror av eller förändrar state kallas för ren funktion. Några andra viktiga begrepp inom funktionell programmering är rekursion och

högre ordningens funktioner, rekursion kommer vi prova på i den här labben men högre ordningens funktioner är mer avancerat och får vänta till en senare kurs.

Upprepning med rekursion

I Ada-labben fick ni prova att iterera, det vill säga upprepa en operation eller sekvens, genom att använda loopar. Både for- och while-looparna som användes i den labben påverkas ju dock av, och påverkar, state. For-loopen har ju en loop-variabel och while-loopen beror av ett sanningsvärdet för någon extern data. Men hur ska man då göra för att upprepa operationer utan att använda loopar?

Rekursion innebär självlikhet och kan demonstreras till exempel genom att ställa sig mellan två speglar riktade mot varandra. När man tittar i den ena spegeln ser man den andra spegeln, i vilken man ser den första spegeln, i vilken man ser den andra spegeln osv. Exemplet med speglarna kallas för oändlig rekursion, det vill säga självlikheten fortsätter i oändlighet.

Oändlig rekursion är inte så praktiskt när man programmerar (för att vara mer exakt leder det till att programmet kraschar när den så kallade anropsstacken blir full) och därför tar vi till en annan liknelse. Matrioshka-dockor, ofta kallade ryska dockor, är en form av trädockor där den största dockan innehåller den näst största och den näst största innehåller den tredje osv. Detta är en så kallad rekursiv struktur och därför lämpar sig en rekursiv process särskillt bra för att behandla den.

Man skulle kunna tänka sig en rekursiv process för att öppna en Matrioshka på följande sätt:

```
FUNKTION FÖR ATT ÖPPNA EN MATRIOSHKA
om Matrioshka kan öppnas
  Öppna matrioshka
  Ta ut inre matrioshka
  Kör samma funktion på den inre matrioshkan
Annars
  Avsluta
```

När man pratar om rekursiva funktioner menar man alltså funktioner som anropar sig själva.

Rekursion är som sagt mycket användbart när man handskas med rekursiva strukturer, exempel på rekursiva strukturer är träd (formella språk, som datorspråk, kan beskrivas som träd och därför översätter de flesta kompilatorer och interpretatorer programkod till så kallade parseträd med hjälp av rekursiva funktioner) och länkade listor. Ett annat användningsområde för rekursion är inom matematiken och det ska vi titta lite närmare på i övningarna längre fram.

2. Exempel på Lisp-program

Det enklaste Lisp-programmet man kan tänka sig är kanske följande:

```
; Lägg ihop 1 och 2
(+ 1 2)
```

Alla rader som inleds med ; är kommentarer. Detta är ett föga imponerande program som helt enkelt lägger ihop 1 och 2 och returnerar värdet av additionen, det vill säga 3. Trots det beskriver programmet ganska väl Lisps generella uppbyggnad. Alla funktionsanrop, även inbyggda matematiska operationer, skrivs i prefixform, det vill säga operatorn/funktionsnamnet först och sedan argumenten, och omsluts av parenteser. I större funktioner leder detta ofta till många slutparenteser och Lisp har av vissa betraktats som en backronym (en efterkonstruerad förkortning) för Lots of Irritating Superflous Parenthesis (Massor av irriterande överflödiga parenteser).

Ovanstående uttryck är dock inte särskilt intressant annat än som ett exempel på Lisps minimala syntax. Alla funktioner, även matematiska operationer, skrivs i prefix-notation och alla uttryck omges av parenteser, utöver detta finns det finns det mycket få syntaktiska konstruktioner i Common Lisp och inga av dem kommer användas i den här labben. En detalj som kan vara intressant att veta är att denna syntax ger Lisp-program samma struktur som listor vilket kan vara användbart inom funktionell programmering.

Ni ska nu prova att skapa en funktion som gör samma sak som ovanstående program fast med vilka två värden som helst:

```
; Lägg ihop två heltal
(defun addera-två-tal (a b)
  (+ a b))
```

Ni anropar sedan er funktion genom att skriva:

```
; Anropa funktionen
(addera-två-tal 1 2)
```

Inte heller detta program är särskilt intressant i verkligheten men demonstrerar ytterligare några detaljer i lisp. För att skapa en funktion skriver man `defun` följt av namnet på funktionen man vill skapa, följt av parametrar inom parentes och efter det kommer själva funktionskroppen där man beskriver vad programmet faktiskt ska göra. För att anropa funktionen skriver man, inom parentes, funktionsnamnet följt av två argument, i det här fallet 1 och 2.

Funktioner och villkor

Dags att göra något lite roligare. Fakulteten av ett tal N består av produkten av alla tal mellan 1 och N och betecknas N!. 1! är således 1, 2! är 2, 3! är 6, 4! är 24

och så vidare. Fakulteten kan räknas ut på flera sätt, i Ada hade man antagligen använt sig av en for-loop som itererade över alla tal 1 till N och multiplicerade ihop dem. I Lisp gör man på ett annat sätt.

Fakulteten för ett tal N blir om vi tänker efter samma sak som N multiplicerat med fakulteten för N-1 för alla N som är större än 1. Detta gäller generellt och brukar ofta användas som en definition av fakultetfunktionen, det vill säga fakultetfunktionen har en rekursiv definition. Att definitionen av fakultetfunktionen är rekursiv gör det extra lätt att konstruera en rekursiv Lisp-funktion som gör samma sak. Här kommer ett exempel på hur fakultetfunktionen kan skrivas i Lisp:

```
; Räkna ut fakulteten för ett givet heltal
(defun fakultet (n)
 (if (= 1 n)
   1
  (* n (fakultet (- n 1)))))
```

Här börjar det bli intressant och i det här exemplet kan man inte bara se exempel på rekursion utan man kan också se hur villkorssatsen `if` fungerar. Villkorssatsen `if` fungerar precis som en funktion med tre argument.

Den första parametern, `(= 1 n)` i det här fallet, är ett logiskt uttryck eller en funktion som returnerar ett sanningsvärde (i Lisp och de flesta andra programmeringsspråk kan man utan problem betrakta det som samma sak). Här använder man jämförelseoperatorn = som jämför om två tal är lika och returnerar t, sant, om de är lika eller nil, falskt, om de inte är lika. Sanningsvärdet nil är i Lisp lika med tomma listan, se nedan.

Den andra parametern, talet 1 i det här fallet, är vad som ska returneras från if-satsen om den första parametern är sann. Detta kan vara antingen ett värde eller en funktion som returnerar ett värde.

Den tredje parametern, `(* n (fakultet (- n 1))`i det här fallet, är vad som ska returneras om den första parametern är falsk. Precis som den andra parametern kan detta vara antingen ett värde eller en funktion som returnerar ett värde.

I slutet av funktionen kan man också se varför Lisp fått backronymen Lots of Irritating Superflous Parenthesis. Dock har de flesta textredigerare, till exempel Emacs, verktyg som minimerar problemet med slutparenteser.

Listor

Lisp är som sagt speciellt bra lämpat för att hantera listor. Ovan nämnde vi att Lisp-funktioner har samma struktur som listor, detta innebär att Lisp alltid betraktar listor som funktionsanrop, det vill säga om man använder listan `(a b`

`c)` kommer Lisp tro att man anropar funktionen `a` med argumenten `b` och `c`. För att undvika detta använder man quotning vilket innebär att man sätter en apostrof före listan. Alltså, om man inte vill att en lista ska köras som en funktion skriver man `'(a b c)`.

Ni kommer främst använda två funktioner för att hantera listor nedan, den ena är `first` och den andra är `rest`. Funktionen `first` returnerar det första elementet i listan, och funktionen `rest` returnerar hela listan utom första elementet. Anropas `rest` på en lista med endast ett element eller en tom lista returneras tomma listan.
Tomma listan är som namnet antyder en lista som inte innehåller några element, den skrivs oftast '() och är i Lisp lika med `nil`, det vill säga falskt.

Funktionerna `first` och `rest` fungerar som följande:

```
CL-USER(1): (first '(1 2 3 4))
1
CL-USER(2): (rest '(1 2 3 4))
(2 3 4)
CL-USER(3): (rest '(1))
nil
CL-USER(4): (rest '())
nil
```

3. Att använda Lisp

Lisp är ett *interpreterat* språk, det innebär att man till skillnad från i Ada inte behöver kompilera koden för att kunna köra programmet. På IDA finns ett Lisp-system som innebär att man kan skriva, interpretera och köra kod inifrån Emacs.

För att få tillgång till detta system öppnar man en terminal och skriver:

```
module add prog/allegro
```

För att sedan starta emacs med allegro-systemet kör man kommandot

```
emacs-allegro-cl
```

Man kommer nu få upp ett emacsfönster med två större buffrar, en övre och en undre. I den undre buffern kan man skriva Lisp-kod direkt, man kan också köra den den kod man skrivit i den övre buffern. Den undre buffern sägs ofta köra en ”read-eval-print-loop”, det vill säga den läser input, evaluerar den, och skriver ut resultatet på skärmen, allt i en så kallad prompt på samma sätt som i terminalfönstret. Det ser ut ungefär såhär:

```
CL-USER(1): (+ 2 4)
6
CL-USER(2): (/ 21 3)
7
```

I den övre buffern kan man skriva större program och flera funktioner vilka sedan kan evalueras genom ett kommando och anropas i den nedre buffern. Ni kommer prova båda dessa sätten att skriva kod nedan.

För att få fram tidigare körda kommandon i Lisp-tolken (den nedre buffern) kan man använda kommandot **C-c C-p** (Ctrl-knappen och c-knappen samtidigt, sedan Ctrl-knappen och p-knappen), det kan vara praktiskt när man experimenterar och testar samma funktion med flera olika input.

4. Övningar

I följande övningar är tanken att ni själva ska konstruera och testa enkla funktioner i Lisp. Till en början kommer ni bara använda Lisp-prompten men i slutet kommer ni skriva aningen mer avancerade funktioner i den övre buffern. Övningar markerade med asterisk är bonusuppgifter man kan göra om man hinner färdigt med de huvudsakliga uppgifterna.

Ibland händer det att man gör något fel och då kan man få upp en meny med ”Restart actions”. Dessa kan på en högre nivå användas till avancerad felsökning men under den här labben rekomenderas att ni bara väljer alternativet ”Return to Top Level ” och ser över er kod. Om ni kör fast och inte lyckas lösa problemet, kontakta handledare.

Övning 1 – Infix till prefix

Innan man kan tänkas göra några mer avancerade program i Lisp måste man vänja sig vid prefixnotation, det vill säga att sätta funktionsanrop, till exempel matematiska operatorer, framför operanderna och inte mellan dem. Till exempel är infixuttrycket (1 + 1) lika med prefixuttrycket (+ 1 1) och uttrycket (1 + 1 + 1) lika med (+ 1 1 1). Byter man operator som i uttrycket (1 + 2 – 3) skriver man (– (+ 1 2) 3), parenteserna fungerar precis som vanligt inom matematiken och markerar prioritetsordning, det vill säga de innersta parenteserna räknas ut först och de yttersta sist. Använd Lisp-prompten för att räkna ut följande uttryck:

44 + 13 – 27
44 + (13 – 27)
1 – 2 + 3 * 4 / 5
(1 – 2 + 3) * 4 / 5
(1 – 2 + 3) * (4 / 5)

Övning 2 – Funktioner

Titta på exempelfunktionen nedan som tar två tal och adderar dem:

```
; Lägg ihop två heltal
(defun addera-två-tal (a b)
  (+ a b))
```

Gör om den så att den tar tre argument, a, b och c, adderar a och b och subtraherar c.

Övning 3 – Rekursion

Titta på funktionen nedan:

```
; Räkna ut fakulteten för ett givet heltal
(defun fakultet (n)
 (if (= 1 n)
   1
  (* n (fakultet (- n 1)))))
```

Gör om funktionen så att den tar en lista som argument istället för ett tal och rekurserar över den listan, med hjälp av `first`- och `rest`-funktionerna som beskrevs ovan, och multiplicera ihop talen. Döp också om funktionen till något lämpligare än fakultet, till exempel `multiplicera-lista`.

Funktionen ska fungera som följande:

```
CL-USER(1): (multiplicera-lista '(2 4 8))
64
CL-USER(2): (multiplicera-lista '(1 5 0 2 4))
0
```

För att göra detta lite lättare rekomenderas att ni använder övre buffern och skriver in koden. Först bör ni skapa en fil att arbeta i, för att göra det sätter ni markören i övre buffern trycker **C-x C-f**, sedan får ni skriva in filnamn, det ska i det här fallet vara *prova.cl*. När ni skrivit färdigt funktionen kan ni ladda in funktionerna från buffern till Lisp-systemet i den nedre buffern genom att trycka **C-c C-b**. När det är gjort kan funktionerna från övre buffern anropas i den nedre. Om koden i den övre buffern ändras måste funktionerna laddas in igen. Glöm inte att spara med jämna mellanrum (**C-x C-s**).

Övning 4 – Fibonacci-serien

Fibonacci-serien är en talserie där varje tal, untantaget de två första, består av summan av de två föregående talen i serien. Ni ska nu implementera en rekursiv funktion, `fibonacci`, som räknar ut ett visst tal i Fibonacci-serien.

Funktionsanropet `(fibonacci 0)` ska ge det ”nollte” talet i Fibonacci-serien, 0, och anropet `(fibonacci 1)` ska ge talet 1. Exempelkörning:

```
CL-USER(1): (fibonacci 1)
1
CL-USER(2): (fibonacci 4)
3
CL-USER(3): (fibonacci 8)
21
```

*Övning 5 – Matematik

Några andra smarta matematiska funktioner som finns inbyggda i Lisp är `truncate`, `round` och `mod`. `Truncate` tar ett tal och om det är ett decimaltal kapar den av decimaldelen, det vill säga den avrundar neråt. `Round` är en mer traditionell avrundningsfunktion som avrundar som vanligt, det vill säga uppåt om decimaldelen är 0,5 eller större. `Mod` är en funktion som returnerar resten vid en heltalsdivision, det vill säga om man kör `(mod 13 5)` kommer man få resultatet 3, eftersom det närmsta heltalet som mindre än eller lika med 13 och delbart med 5 är 10, och då har man 3 (13 – 10) kvar i rest.

Använd dessa funktioner för att implementera funktionerna ental och tiotal. Funktionen ental ska ta ett godtyckligt positivt heltal och returnera entalssiffran medan funktionen tiotal ska göra samma sak med tiotalssiffran. Exempel:

```
CL-USER(1): (ental 4)
4
CL-USER(2): (ental 528)
8
CL-USER(2): (tiotal 7315)
1
CL-USER(2): (tiotal 5)
0
```

*Övning 6 – Sekventiell bearbetning

Ni ska nu prova att gå igenom en lista, element för element, och för varje element göra någon typ av operation, detta kallas sekventiell bearbetning och är ett mycket viktigt område inom programmering. I Lisp gör man detta med hjälp av rekursion.

Ni ska nu skapa funktionen `bara-tal`, som tar en lista med godtyckligt antal element och tar bort alla element som inte är tal. Funktionen ska fungera såhär:

```
CL-USER(1): (bara-tal '(1 a b 2 3 c))
(1 2 3)
```

Funktioner som kommer behövas i den här uppgiften är vilkorssatsen `cond`, listfunktionen `cons` och testfunktionen `numberp`. Dessa funktioner är beskrivna nedan.

Förrutom `if` finns också en annan vilkorssats i Lisp och det är `cond`. Funktionen `cond` kan man använda när man har mer komplicerade villkor eftersom man kan ha hur många villkor som helst. Här är ett exempel som returnerar kvadraten av ett positivt tal `n` utan att göra den onödiga beräkningen `(* 1 1)` och returnerar 0 om n inte är ett positivt tal större än eller lika med 1:

```
(cond
  ((= 1 n) 1)
  ((> 1 n) (* n n))
  (t 0))
```

Funktionen cons är en av de viktigaste funktionerna i Lisp och är kopplad till den grundläggande datastrukturen cons-cell i Lisp. För er som programmerat tidigare är en cons-cell en struktur som består av två pekare, dessa celler bygger upp Lisps viktigaste datastruktur, listan. Hur detta fungerar på en teknisk nivå ska vi inte gå in på utan vi kommer bara bara beskriva snabbt hur cons-funktionen fungerar. Funktionen cons används här för att skapa listor på följande sätt.

```
CL-USER(1): (cons 1 '())
(1)
CL-USER(2): (cons 1 '(2))
(1 2)
CL-USER(3): (cons 1 (cons 2 '()))
(1 2)
```

Testfunken numberp används för att testa om ett objekt är ett tal medan endp används för att testa om en lista är tom. Exempel:

```
CL-USER(1): (numberp 1)
t
CL-USER(2): (numberp 'a)
nil
CL-USER(3): (endp '())
t
CL-USER(4): (endp '(1 2 3))
nil
```

Tips: I funktionen kommer det finnas tre fall som ni behöver ta hänsyn till, när listan är tom, när första elementet är ett tal och när första elementet är något annat.

*Övning 7

Listor kan i Lisp innehålla vilken data som helst, även andra listor. Gör om uppgift 6 så att den tar hänsyn till att ett element i en lista kan vara en lista och rensar bort alla element som inte är tal eller listor även ur denna. Funktionen ska alltså fungera på följande sätt:

```
CL-USER(1): (bara-tal2 '(1 a b 2 3 c))
(1 2 3)
CL-USER(2): (bara-tal2 '(1 a b (d 4) 2 3 c))
(1 (4) 2 3)
CL-USER(3): (bara-tal2 '(1 a b 2 (d) 3 c))
(1 2 nil 3)
```

Att resultatet vid det tredje anropet ser ut som det gör är för att listan (d), när man tar bort alla element som inte är tal eller listor, ju är tomma listan '() vilken är lika med nil.

För att göra den här uppgiften kommer man behöva använda funktionen listp, vilken fungerar som numberp och endp men undersöker om ett objekt är en lista.

Tips: Glöm inte att även tomma listan är en lista.

5. Referenser

Framtida kurser

Programspråket Lisp används som exempelspråk i flera kurser på IDA. Bland annat i kursen Data- och Programstrukturer, där dialekten Scheme används, kursen Artificiell Intelligens för C- och D-programmen använder Common Lisp och kursen AI och Lisp använder också, som namnet antyder, Lisp.

Litteratur

I den rekommenderade referensboken *Computer Science: An Overview* av J. Glenn Brookshear tas funktionell programmering och Lisp knappt upp men det finns ett kort avsnitt i avsnitt 6.1.

Du kan läsa mer om Lisp i boken *Programmering i Lisp* av Anders Haraldsson. Boken används som kurslitteratur i kursen Funktionell Programmerin och Lisp.

Du kan också läsa om Lisp i *Concepts of Programming Languages* av Robert W. Sebesta. Det är en bok som tar upp många olika programspråk och sätt att programmera.

6. Frågor att fundera över

För dig som är nybörjare på programmering

Programmering är en process som består av flera faser. Först måste man förstå vad det är för problem som man ska lösa. Därefter ska man försöka designa en lösning. Denna lösning ska *implementeras*, det vill säga man ska skriva själva programmet. Sist men inte minst måste det färdiga programmet testas. Oftast går dock inte processen så här rakt och tydligt. Många gånger kan man tjäna på att experimentera lite, utan att ha förstått själva problemet. En del hävdar till och med att det är först när man har utformat lösningen som man verkligen har förstått problemet. För att bli en bra programmerare krävs lång träning – mycket längre än vad ens en högskoleutbildning kan ge. Man måste lära känna de olika byggstenarna som finns i programspråket, men också lära sig när och hur man ska använda dem.

Hur gjorde du när du löste övningarna i den här laborationen? Kan du känna igen de steg som beskrivs ovan? Känner du att du har förstått åtminstone lite grann av vilka byggstenar som finns i Lisp? Kändes det svårt att försöka formulera lösningar på problem i ett formellt programmeringsspråk?

För dig som programmerat en del förut

Liknar Lisp något annat språk som du har erfarenhet av? Vad känns svårare eller lättare att göra i Lisp, enligt din bedömning?