DA4003: Programmeringsparadigm

Anders Mörtberg

22 februari 2024

Innehåll

Kapitel 1

Introduktion och lite imperativ programmering i C

1.1 Programmeringsparadigm

I denna kurs ska vi titta närmare på fem paradigm och fem programmeringsspråk som tillhör dessa paradigm:

- Imperativ programmering (C)
- Objektorienterad programmering (Java)
- Webb- och händelsestyrd programmering (JavaScript)
- Funktionell programmering (Haskell)
- Logikprogrammering (Prolog)

Utöver detta kommer vi även diskutera grundläggande och centrala programmeringsbegrepp och koncept (typer, syntax, parsning, grammatiker, semantik, beräkningsmodeller, kompilering, länkning, tolkning, minneshantering…).

Definition av "paradigm" (Nationalencyklopedin): *benämning på det mönster som styr vetenskapligt tänkande.*

Programmeringsparadigm = de underliggande mönstren som styr hur man programmerar.

Ett programmeringsparadigm är mer generellt än ett specifikt programmeringsspråk. Förr i tiden var språk ofta knutna till ett specifikt paradigm (exempel: Basic, C, LISP…), men idag tillhör många moderna språk olika paradigm (exempel: Python är imperativt, objektorienterat och fuktionellt). Många språk gör det lätt att programmera på ett specifikt sätt (exempel: JavaScript är lämpat för webbprogrammering). Många språk är även under konstant utveckling och olika primitiver från andra paradigm läggs till (exempel: Java 8 har fått "lambda-uttryck" vilket är typiskt för funktionell programmering). Genom att lära sig nya språk lär man sig nya sätt att programmera och att lösa problem:

> *A language that doesn't affect the way you think about programming, is not worth knowing* –
> Alan Perlis (första vinnaren av Turingpriset, "Nobellpriset i datalogi").

Olika paradigm och programmeringsspråk är bra på att lösa olika typer av problem. Därför är det bra att kunna många olika paradigm och språk så att man kan välja det som är bäst lämpat för att lösa ett

specifikt problem. Många problem kan även lösas på olika sätt i olika paradigm. Exempel: man kan implementera quicksort både funktionellt och imperativt i Python (så genom rekursion eller genom loopar). Dessa implementationer kommer lösa samma problem (sortera en lista) och använda samma algoritm (quicksort), men lösningarna (dvs Python koden) kommer se väldigt olika ut då de är skrivna i två olika paradigm.

Man kan även översätta mellan lösningar skrivna i ett paradigm till ett annat. Ett exempel på när detta kan hända är vid *kompilering*. En kompilator är ett program som översätter från ett språk till ett annat. Detta görs ofta med ett flertal optimeringar så att den resulterande koden går snabbare att köra. Detta görs även ofta från ett "högnivåspråk" till ett "lågnivåspråk". Ett högnivåspråk är ett språk som är i princip oberoende av vilken typ av dator som koden ska köras på, medans ett lågnivåspråk är bundet till en specifik hårdvara (exempel: x86 assembler är ett språk som ursprungligen utvecklades för Intels x86 processorer).

Till varje paradigm följer en abstrakt modell för hur en beräkningsmaskin/dator fungerar, dvs det finns olika "beräkningsmodeller" som kan associeras till olika programmeringsparadigm. Exempel på hur man kan tänka att en dator utför beräkningar:

- En massa transistorer och elektriska impulser
- Minne och instruktioner som exekveras av en processor (von Neumann modellen)
- Substitutionsmodellen
- Evaluator av matematiska funktioner
- Ett nätverk av enheter som kommunicerar genom meddelanden
- En teorembevisare
- …

Det har även uppstått ett antal "paradigmskift" inom programmering. Det kanske mest kända exemplet på detta var efter Dijkstras artikel "Go To Statement Considered Harmful" vilken gjorde att programkonstruktionen "Go To" försvann från i princip alla nya språk som utvecklades. Under de senaste årtioendena har det även skett ett paradigmskifte där idéer och konstruktioner från funktionell programmering (e.g. LISP, Haskell, OCaml) har lagts till många av de mest populära språken (exempelvis C++ och Java har lagt till lambda-uttryck).

1.1.1 Vanliga programkonstruktioner

Det finns väldigt många programmeringsspråk (https://en.wikipedia.org/wiki/List_of_programming_languages) och från dessa kan man utläsa ett antal olika paradigm (https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages#Paradigm_summaries). För att se exempel på hur "Hello World!" kan skrivas i de flesta programmeringsspråken se http://helloworldcollection.de/.

Det finns dock ett antal primitiva egenskaper som delas av många programmeringsspåk och paradigm. Innan vi börjar titta på specifika språk och paradigm ska vi gå igenom ett antal av dessa gemensamma egenskaper som finns till olika grad i de språk vi ska studera i kursen.

1.1.1.1 Typer

I många programmeringsspråk finns det typer som grovt beskriver vad ett program handlar om. Vi skriver `v : T` för ett värde/program/term `v` som har typen `T`. Ett programmeringsspråk skulle kunna ha följande exempel på typer:

```
0 : int
1 : int
```

```
-2 : int
1 : nat
'c' : char
"hello" : string
0.5 : float
(1/2) : rational
True : bool
False : bool
[1,2,3] : list nat
```

Många språk låter även programmeraren definiera nya typer (exempel: i Python och Java kan man skapa en ny typ genom `class`, medans i Haskell skriver man `data`).

Vissa språk har dock inga typer och då kallas de "otypade". Bland typade språk (dvs språk med typer) finns det även skillnader i hur typade de är, ibland kan man skilja på hur uttrycksfulla typerna är i olika språk eller när kompilator/tolk testar att alla definitioner i programmen har rätt typ. Ett "statiskt typat" språk är ett språk där kompilatorn vet typen på alla variabler och kollar så att man inte applicerar t.ex. en funktion som vill ha en `int` på en sträng. Detta gör det möjligt att hitta många triviala buggar redan innan man kör programmet. Andra språk är "dynamiskt typade" vilket betyder att typerna endast är associerade till värden vid körning, på så sätt behöver programmeraren ge mindre information när man skriver program vilket gör det snabbare att koda. Men triviala buggar (som att man råkar addera ett tal till en sträng) upptäcks bara när man kör programmet, antingen genom att det kraschar eller att man får oväntad utdata. Exempelvis C, Java och Haskell är statiskt typade, medans Python, PHP och JavaScript är dynamiskt typade.

1.1.1.2 Namnge saker

En annan väldigt viktig egenskap som finns i de flesta (om inte alla?) högnivåspråk är förmågan att namnge "saker". Exempelvis om man vill beräkna arean av en cirkel med radie `2` så kan man skriva följande i Python-tolken:

```
>>> 2 * 2 * 3.14
12.56
```

Detta är dock varken speciellt informativ eller välskriven kod då man inte på ett enkelt sätt kan läsa vad den gör eller återanvända värdet för pi. Om vi istället ger konstanten `3.14` namnet `pi` kan vi skriva:

```
>>> pi = 3.14
>>> 2 * 2 * pi
12.56
```

Vi kan på detta sätt återanvända konstanten `pi` och det är lättare att gissa sig till vad koden gör.

Ett väldigt viktig egenskap hos namngivna konstanter eller variabler är om de är muterbara eller omuterbara, dvs. om deras värde kan ändras utan att man ger dom ett nytt namn. I Python är t.ex. listor muterbara:

```
>>> xs = [1, 2, 3]
>>> xs.pop()
3
>>> xs
[1, 2]
```

Man säger här att pop har en *sidoeffekt* som består i att ta bort det sista elementet i `xs`. I språk som Haskell är listor (och i princip allt annat) omuterbara och man kan inte göra några bestående ändringar utan att namnge resultatet.

```
Prelude> xs = [1,2,3]
Prelude> ys = init xs
Prelude> ys
[1,2]
```

1.1.1.3 Parametrisering och funktioner

En annan väldigt viktig egenskap som i princip alla programmeringsspråk delar är förmågan att skriva funktioner. Detta betyder att vi kan skriva uttryck som tar in ett antal abstrakta parametrar och sen ger ett resultat som beror på parametrarnas värden. På så sätt kan programmeraren återanvända kodsnuttar i olika situationer och undvika "copy-paste" programmering. Funktionsdefinitioner låter även programmeraren skriva mer abstrakt kod då variabler/parametrar är abstrakta och ej har något konkret värde förrän man anropar funktionen.

Exempelvis är det bättre att definiera en funktion för att beräkna arean istället för att skriva in koden ovan:

```
pi = 3.14

def area(radius):
    return radius * radius * pi
```

Genom att göra detta behöver vi inte skriva in uttrycket som beräknar arean varje gång vi vill beräkna den. I detta exempel har vi inte behövt specificera att `pi` är ett flyttal. I många språk, som C och Java, måste detta dock göras men då Python är dynamiskt typat behöver vi inte skriva något.

Det är lätt att lista ut vad typen på `pi` är genom att bara titta på koden, men vad är typen på funktionen `area`? I Python kan vi inte specificera detta, men vi kan föreställa oss en statiskt typad version av Python där vi skulle kunna skriva:

```
float def area(float radius):
    return radius * radius * pi
```

Detta påminner lite om C och Java. Innan `def` har vi skrivit retur-typen för `area` och innan parametern `radius` har vi skrivit dess typ. Så `area` är alltså en funktion som tar in en `float` och returnerar en `float`. Vi kommer ofta skriva detta på följande kompaktare sätt `area : float -> float`. I statiskt typade språk kommer kompilatorn/tolken testa att `area` verkligen returnerar en `float` i alla `return` uttryck och man kan inte applicera `area` på något element som inte är en `float`.

En bra grej med att ge typen på funktioner är att det dokumenterar ens kod och man kan gissa vad en funktion gör utan att se dess definition eller läsa några kommentarer. Till exempel följande funktionsnamn och typer gör det lätt att gissa vad funktionerna gör:

```
length : list nat -> nat
+ : nat -> nat -> nat
abs : int -> nat
```

1.2 Imperativ programmering

I imperativ programmering körs instruktioner i den ordning som bestäms av programmeraren och nästa instruktion som kommer köras är nästa instruktion som programmeraren skrivit. Vi kan alltså följa flödet i ett program genom att titta på vilken del som ska köras näst.

Man kan dela upp uttryck/instruktioner i två olika klasser:

- Atomiska: tilldelningar, I/O, aritmetik, logiska, ...
- Kontroll: innehåller andra uttryck och bestämmer flödet i programmet (e.g. `if`, `goto`, `for`, `while`...).

Termen används ofta i kontrast till *deklarativ* programmering där man istället fokuserar på vad programmet ska åstadkomma utan att i detalj specificera hur resultat ska uppnås (e.g. logikprogrammering).

1.2.1 von Neumann modellen

Imperativ programmering hänger ihop med en modell för hur beräkningsmaskiner/datorer fungerar som brukar kallas för "von Neumann" modellen då den uppfanns av matematikern John von Neumann 1945 (https://en.wikipedia.org/wiki/Von_Neumann_architecture). I denna modell utförs beräkningar genom att program förändrar värdet på minnesceller. Program har olika sorters operationer, bland annat tilldelningar, aritmetiska operationer, logiska operationer, hopp operationer, loopar... Detta påminner alltså om hur en vanlig dator fungerar.

En idealiserad version av von Neumann modellen är Turingmaskiner (som uppfanns av Alan Turing 1936 https://en.wikipedia.org/wiki/Turing_machine). Minnet i en Turingmaskin är ett oändligt långt 1-dimensionellt band uppdelad i lika stora delar som kan spara en bit. Beräkningar utförs sedan genom att positionen på bandet förflyttas eller att en bit skrivs på nuvarande position. Denna maskin är teoretiskt intressant då den är minst lika kraftfull som von Neumann maskinen trotts att den är så enkel. Vidare kan alla datorer eller program som byggts efter von Neumann modellen reduceras till Turingmaskiner. Ett programmeringsspråk är Turing-komplett om man kan göra lika mycket som man kan göra med en Turingmaskin i det. Alla programmeringsspråk vi ska se i den här kursen är Turing-kompletta.

1.3 Programmering i C

Programmeringsspråket C är väldigt populärt och det finns oräkneliga tutorials och böcker om det. Här kommer en lista över några tutorials som går igenom mycket mer än vad vi kommer hinna i den här kursen:

- https://www.programiz.com/c-programming
- https://www.tutorialspoint.com/cprogramming/index.htm
- https://www.cprogramming.com/tutorial/c-tutorial.html
- https://www.w3schools.in/c-tutorial/intro/

Om ni vill ha en bok om C så är det *"The C Programming Language"* av Brian Kernighan och Dennis Ritchie som gäller (https://en.wikipedia.org/wiki/The_C_Programming_Language). Den gavs ut redan 1978 och har varit väldigt inflytelserik, bland annat uppfann den "Hello, World!" programmet.

Lite fakta om C:

- C är gammalt och stabilt (standardiserades redan 1988!)
- Namnet kommer från att det är en efterföljare till språket B.
- C uppfanns under tidigt 1970-tal och har varit extremt inflytelserikt och många koncept från C återfinns i nyare språk.

- Uppfanns för att skriva operativsystemet UNIX. Även Linux är skrivet i C.
- Väldigt populärt för "systems progamming", dvs maskinnära programmering (operativsystem, drivrutiner, nätverksprogrammering…)
- Har manuell minneshantering. Bra för att skriva väldigt snabba program, men även ofta osäkert. En av de vanligaste teknikerna för att hacka program är genom så kallade *buffer overflows* (https://en.wikipedia.org/wiki/Buffer_overflow). Det är vanligt att C program är känsliga för denna typ av attacker.
- Relativt enkelt språk och väldigt imperativt.
- Finns kompilatorer till väldigt många olika typer av processorer och datorer.
- Är ett relativt lågnivåspråk (vissa säger att C är ett "mellannivåspråk"). Med "nivå" menas hur nära språket är till den underliggande dator som koden körs på. C inte på samma låga nivå som assemblyspråk, men inte heller på samma höga nivå som t.ex Python och Java.

Många erfarna programmerare tycker att alla programmerare bör lära sig C någon gång. När man skriver kod i C måste man verkligen förstå vad som händer i datorn när ett program körs, så genom att lära sig C kan man förstå varför kod skriven i andra språk inte gör vad man vill eller är långsam. På grund av detta måste man tänka på saker som register, stack, heap, minnesallokering, etc., när man programmerar i C vilket man inte alls behöver göra i språk som Python och Java. Då vi endast har två föreläsningar på oss kommer vi inte hinna gå igenom allt i C. Målet är istället att gå igenom grunderna och de viktigaste koncepten utan att gå in på alla detaljer. Allt man behöver kunna för att lösa labben kommer även gås igenom.

1.3.1 Hello, World!

Det första program som man bör skriva i ett nytt språk är "Hello, World!":

```c
#include <stdio.h>

/* Kod till Hello, World! */
int main()
{
    // printf skriver ut strängen inom dubbelfnuttar
    printf("Hello, World!\n");

    return 0;
}
```

För att köra denna kodsnutt lägg det i en fil som heter "helloworld.c" och kompilera sedan det genom:

```
$ gcc helloworld.c
```

Detta genererar en fil `a.out` som vi kan köra i terminalen:

```
$ ./a.out
Hello, World!
```

Vill vi att programmet istället ska heta `hello` skriver vi `gcc -o hello helloworld.c`

Vi kommer nu gå igenom programmet i detalj:

- `#include <stdio.h>` importerar `stdio` biblioteket vilket innehåller funktioner för input-output, som `printf` och `scanf` (för att skriva ut och läsa in). Har man inte importerat detta bibliotek kan man inte använda `printf`, så man kan inte skriva ut något.

- Kommentarer i C skrivs antingen med `//` eller inom `/*` och `*/`.
- "f" i `printf` står för "formatted" då funktionen tar in en "formatteringssträng". Mer om detta senare.
- `main` funktionen är den del av ett C program som kommer börja köras först. Då C är imperativt körs instruktionerna i `main` en efter en från toppen av funktionskroppen (dvs, koden inom "`{`" och "`}`").
- `main` funktionen måste returnera en `int` (dvs ett heltal) och detta specificeras framför `main()`. Vi returnerar `0` då returvärdet för `main` funktionen är "Exit Status" för hela programmet och i de flesta operativsystem är `0` en kod för att allt fungerade som det skulle.

Man har mycket mer frihet i hur man kan indentera sin kod i C än Python och det finns många åsikter om hur man bör indentera sin C kod. Jag försöker följa samma stil som i *"The C Programming Language"* boken: https://en.wikipedia.org/wiki/Indentation_style#K&R_style

Då man har väldigt mycket frihet i hur man skriver kod i C finns det tävlingar i att skriva mest obskyr kod, se till exempel: The International Obfuscated C Code Contest

1.3.2 Input-Output

Vi kommer nu titta på några exempel på hur input-output fungerar i C. Vi kommer bara titta på enkel utskrift och inläsning från användaren, så ingen filhantering och annan mer avancerad input-output.

1.3.2.1 Läsa in och skriva ut ett tal från användaren

```c
#include <stdio.h>

int main()
{
    int number;

    printf("Enter an integer: ");

    // läser in och sparar resultatet i number
    scanf("%d", &number);

    // skriver ut number
    printf("You entered: %d\n", number);

    return 0;
}
```

- Vi deklarerar först en variabel `number` av typ `int` (detta måste göras annars får vi ett kompileringsfel – testa själv!).
- Vi använder sedan `scanf` för att läsa in ett tal från användaren. Vi har använt `"%d"` som formateringssträng i `scanf` för att specificera att vi ska läsa in en `int` från användaren (mer om det senare). När användaren skriver in ett tal sparas det till `number`.

Obs: Vi har använt `&number` och inte bara `number` i `scanf`. Detta är då `&number` ger oss addressen i minnet där `number` är och värdet användaren skriver in sparas sedan på den addressen. Mer om detta i nästa föreläsning.

1.3.2.2 Läsa in flera tal

Vi kan även läsa in flera tal från användaren:

```c
#include <stdio.h>

int main()
{
    int number1, number2, sum;

    printf("Enter two integers: ");

    scanf("%d %d", &number1, &number2);

    // beräkna summan
    sum = number1 + number2;

    printf("%d + %d = %d\n", number1, number2, sum);

    return 0;
}
```

Obs: mellanslaget i formateringssträngen i `scanf` säger åt funktionen att ignorera alla typer av blanksteg mellan siffrorna (så nya rader, tabbar och mellanslag ignoreras).

1.3.2.3 Läsa in flyttal

Vi kan även läsa in flyttal.

```c
#include <stdio.h>

int main()
{
    float num1;

    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);

    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf\n", num2);

    return 0;
}
```

Vi använder `%f` och `%lf` för `float` respektive `double`. En `double` tar dubbelt så mycket minne som en `float` och kan därför hålla mycket större tal, mer om det senare.

1.3.2.4 Läsa in bokstäver

Vi kan även läsa in och skriva ut bokstäver:

```c
#include <stdio.h>

int main()
{
    char chr;
    printf("Enter a character: ");

    scanf("%c",&chr);

    printf("You entered: %c", chr);
    printf("ASCII value: %d", chr);

    return 0;
}
```

När en bokstav skrivs in av användaren sparas inte bokstaven själv i `chr`. Istället sparas ett heltal som representerar bokstavens värde enligt ASCII standarden (ASCII står för *American Standard Code for Information Interexchange*).

När vi skriver ut `chr` med `%c` skrivs bokstaven ut, men när vi använder `%d` skrivs dess ASCII värde ut. En av uppgifterna i labben handlar om att skriva ut ASCII värden för en massa bokstäver.

1.3.2.5 Formateringssträngar

Både `printf` och `scanf` använder sig av så kallade formateringssträngar för att läsa in och skriva ut saker. Dessa strängar använder platshållare (eng. *placeholders*) för att specificera var olika typer av argument ska vara. Här kommer en lista över vanliga platshållare:

- `%d` för `int`
- `%f` för `float`
- `%lf` för `double`
- `%c` för `char`
- `%s` för `string`

Det finns många fler, men dessa behövs inte i kursen. Wikipediasidan om formateringssträngar innehåller mycket mer information: https://en.wikipedia.org/wiki/Printf_format_string

1.3.3 Typer

I C måste alla variabler ges en typ. Vilken typ en variabel har bestämmer hur mycket minne som allokeras för den. Om vi skriver

```c
int myVar;
```

så kommer `myVar` vara en variabel av typ `int` (heltal). Storleken på en `int` är 4 byte, så när vi skriver detta kommer kompilatorn allokera 4 byte för `myVar` (dvs, $4 \cdot 8 = 32$ bitar).

Variabelnamn är bara symboliska representationer för minnesplatser. Till exempel

```c
int myVar = 42;
```

kommer skapa en variabel av `int` typ med värdet 42 på en plats i minnet som nu kallas `myVar`. För att få den riktiga minnesaddressen för en variabel skriver man `&myVar`. Detta är vad vi använt i `scanf` ovan och vi kommer prata mycket mer om det i nästa föreläsning. Nedan följer en kodsnutt där vi skriver ut värdet och addressen för `myVar` som ett hexadecimaltal (platshållaren `%p` står för "pointer", dvs pekare):

```
#include <stdio.h>

int main()
{
    int myVar = 42;

    printf("Value at myVar: %d\n",myVar);
    printf("Address of myVar: %p\n",&myVar);

    return 0;
}
```

C är ett starkt typat språk. Detta betyder att en variabels typ inte kan ändras när den väl är deklarerad. Till exempel följande ger ett felmeddelande:

```
    int number = 5;
    double number = 5.0;
```

1.3.3.1 Aritmetiska typer

Alla typer i C har en storlek. Vi kan få ut hur många byte en variabel av en specifik typ upptar med `sizeof`:

```
#include <stdio.h>

int main()
{
    int intType;
    float floatType;
    double doubleType;
    char charType;

    // sizeof beräknar storleken på en variabel
    printf("Size of int: %ld bytes\n", sizeof(intType));
    printf("Size of float: %ld bytes\n", sizeof(floatType));
    printf("Size of double: %ld bytes\n", sizeof(doubleType));
    printf("Size of char: %ld byte\n", sizeof(charType));

    return 0;
}
```

Utdata:

```
Size of int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of char: 1 byte
```

Finns även short, long, unsigned, signed versioner av dessa men de behövs inte för kursen.

Hur stor en typ är representerar hur många olika värden variabler av den kan ha. Då en `char` bara är 1 byte (8 bitar), så kan variabler av typ `char` endast ha ett av $2^8 = 256$ olika möjliga värden. ASCII standarden bestämmer hur dessa 256 olika värden ska tolkas som bokstäver.

1.3.3.2 void typen

C har även en "tom" typ som heter `void`. Den är användbar för funktioner som inte returnerar något.

1.3.3.3 bool

Det finns ingen primitiv `bool` typ i C, men man kan importera `stdbool` för att få typen `bool` med värden `true` och `false`.

1.3.3.4 Strängar

Det finns även strängar i C. Dessa skrivs inom dubbelfnuttar ". En sträng är en array av characters och vi kommer prata mer om dom i nästa föreläsning.

1.3.4 Operatorer i C

C har många välbekanta operatorer som även återfinns i Python, men det finns några som inte finns Python.

1.3.4.1 Aritmetiska operationer

Följande funktioner känner alla igen:

- `+` är addition
- `-` är subtraktion
- `*` är multiplikation
- `/` är division
- `%` är rest/modulo

C har även inkrementerings och dekrementerings operatorer. Dessa kan användas prefix och postfix vilket påverkar när värdet ökas eller minskas.

```c
#include <stdio.h>

int main()
{
    int var = 5;

    // 5 skriv ut, sen ökas var till 6
    printf("%d\n",var++);

    // här är var = 6
    printf("%d\n",var);

    // först är var = 6, sen ökas den till 7 och därefter skrivs den ut
    printf("%d",++var);
```

```
    return 0;
}
```

Detta är användbart i loopar som vi ska diskutera senare.

1.3.4.2 Tilldelningsoperatorer

Precis som i Python finns det tilldelningsoperatorer i C:

- `a += b` är `a = a + b`
- `a -= b` är `a = a - b`
- `a *= b` är `a = a * b`
- `a /= b` är `a = a / b`
- `a %= b` är `a = a % b`

1.3.4.3 Jämförelseoperatorer

I och med att det inte finns bool i C används `0`/`1` för boolska värden om man inte importerar stdbool.

- == är "likhet" (`5 == 3` evaluerar till `0`)
- > är "större än" (`5 > 3` evaluerar till `1`)
- < är "mindre än"
- `!=` är "olikhet" (`5 != 3` evaluerar till `1`)
- >= är "större än eller lika"
- <= är "mindre än eller lika"

1.3.4.4 Logiska operatorer

- && är logiskt OCH (AND)
- || är logiskt ELLER (OR)
- ! är logisk NEGERING (NOT)

1.3.4.5 Bitoperatorer

Det finns även operatorer för att manipulera bitsekvenser, men dessa kommer inte användas i kursen.

Kapitel 2

Mer imperativ programmering i C

2.1 Funktioner i C

Användaren kan skriva sina egna funktioner i C. Returtypen kommer först, sen funktionsnamnet med argument och deras typer inom paranteser och sist funktionskroppen som specificerar vad funktionen gör inom "{" och "}".

```
#include <stdio.h>

int addNumbers(int a, int b)
{
    int result;

    result = a + b;

    return result;
}

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);

    printf("sum = %d",sum);

    return 0;
}
```

Funktioner behöver inte returnera något, då används `void`.

```
#include <stdio.h>
```

```c
void foo()
{
    printf("Hej hopp!\n");
}

int main()
{
    foo();

    return 0;
}
```

Obs: en funktions typ måste deklareras innan den används. Detta kan göras antingen genom att skriva funktionen innan den används som ovan eller genom att ge funktionens typ i en header fil (med filändelse `.h`).

2.2 Villkorssatser

Precis som i Python finns det `if-else` i C:

```c
#include <stdio.h>

int main()
{
    int number;

    printf("Enter an integer: ");
    scanf("%d", &number);

    // blir true (dvs 1) om resten är 0
    if (number % 2 == 0) {
        printf("%d is an even integer.\n",number);
    } else {
        printf("%d is an odd integer.\n",number);
    }

    return 0;
}
```

Man kan även skriva kod som motsvarar Python's `elif`:

```c
#include <stdio.h>

int main()
{
    int number1, number2;

    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
```

```
    if(number1 == number2) {
        printf("Result: %d = %d\n",number1,number2);
    } else if (number1 > number2) {
        printf("Result: %d > %d\n", number1, number2);
    } else {
        printf("Result: %d < %d\n",number1, number2);
    }

    return 0;
}
```

2.3 Loopar

C har tre typer av loopar:

- `for` loopar
- `while` loopar
- `do-while` loopar

2.3.1 `for` loopar

Syntaxen för `for` loopar i C är:

```
for (startUttryck; testUttryck; uppdateringsUttryck)
{
    // loopens kropp
}
```

När C exekverar en `for` loop händer följande:

- Startuttrycket körs endast en gång
- Sedan evalueras testuttrycket, om det blir false (dvs 0) avslutas loopen
- Annars kommer loopens kropp köras och sedan körs uppdateringsuttrycket
- Sedan körs testet igen och loopen slutar eller fortsätter beroende på om testet blir false eller true.

Ett exempel som skriver ut siffror:

```
#include <stdio.h>

int main()
{
    int i;

    for (i = 1; i < 11; ++i) {
      printf("%d ", i);
    }

    return 0;
}
```

Beräkna summan av alla tal upp till ett tal som användaren skriver in:

```c
#include <stdio.h>

int main()
{
    int num, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    for (int count = 1; count <= num; ++count) {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```

2.3.2 `while` loopar

Syntaxen för `while` loopar är:

```c
while (testUttryck)
{
    // loopens kropp
}
```

När C exekverar en `while` loop händer följande:

- Testuttrycket evalueras först.
- Om det blir true kommer loopens kropp köras, sen evalueras testuttrycket igen.
- Detta görs till testuttrycket är falskt.

Exempel:

```c
#include <stdio.h>

int main()
{
    int i = 1;

    while (i <= 5) {
        printf("%d\n", i++);
    }

    return 0;
}
```

2.3.3 `do-while` loopar

En `do-while` loop är väldigt lik en `while` loop, bortsett från en viktig skillnad. Kroppen i en `do-while` loop körs en gång innan testet evalueras.

Syntaxen för do-while loopar är:

```
do
{
    // loopens kropp
}
while (testUttryck);
```

När C exekverar en do-while loop händer följande:

- Kroppen körs en gång.
- Sen evalueras testuttrycket.
- Om det blir true kommer loopens kropp köras igen och sen evalueras testuttrycket igen.
- Detta görs till testuttrycket är falskt.

Exempel:

```
#include <stdio.h>

int main()
{
    double number, sum = 0;

    do {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    } while (number != 0.0);

    printf("Sum = %.2lf\n",sum);

    return 0;
}
```

2.3.4 break och continue

Precis som i Python finns break och continue.

I följande program avslutas loopen så fort användaren skriver in ett negativt tal:

```
#include <stdio.h>

int main()
{
    int i;
    double number, sum = 0.0;

    for (i = 1; i <= 10; ++i) {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        if (number < 0.0)
            break;
```

```
        sum += number;
    }

    printf("Sum = %.2lf\n",sum);

    return 0;
}
```

I följande exempel börjas istället loopens kropp om om användaren skriver ett negativt tal:

```
#include <stdio.h>

int main()
{
    int i;
    double number, sum = 0.0;

    for (i=1; i <= 10; ++i) {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        if (number < 0.0)
            continue;

        sum += number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

2.4 Scope, lokala/globala variabler

Ett *scope* inom programmering är en del av ett program där en definierad variabel existerar och utanför scopet kan man inte komma åt variabeln. Det finns tre ställen där en variabel kan deklarareras i C:

- I en funktion eller kodblock (så även startuttryck i `for` loopar). Dessa kallas *lokala* variabler.
- Utanför alla funktioner, dvs på "toppnivå". Dessa variabler kalla *globala*.
- Som parameter till en funktion, då kallas variabeln en *formell parameter*.

Med "kodblock" menas kod som står inom `{` och `}` (samt kod efter typ en for-loop där det bara är en rad så att `{` och `}` inte behövs).

2.4.1 Lokala variabler

En variabel som deklarares i ett kodblock är lokal till det blocket. Det betyder att den inte existerar utanför blocket där den deklarerats. Exempel:

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; ++i) {
        printf("Hej!");
    }

    // Felmeddelande: i is not declared at this point
    printf("i = %d\n", i);

    return 0;
}
```

När vi kör programmet ovan kommer vi få ett felmeddelande då `i` endast existerar i kodblocket för loopen. På samma sätt är `n1` och `n2` lokala till `main` respektive `func` i:

```c
int main()
{
    int n1 = 1;
}

void func()
{
    int n2 = 2;
}
```

2.4.2 Globala variabler

Variabler som deklareras utanför alla funktioner är globala. De kan nås från alla funktioner i programmet.

```c
#include <stdio.h>

int n = 5;

void display()
{
    ++n;
    printf("n = %d", n);
}

int main()
{
    ++n;
    display();
    return 0;
}
```

2.4.3 Formella parametrar

En parameter till en funktion är som en lokal variabel i funktionen och de överskuggar globala variabler. Exempel:

```
#include <stdio.h>

int a = 20;

int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 0;

    printf ("value of a in main() = %d\n",  a);
    c = sum( a, b);
    printf ("value of c in main() = %d\n",  c);

    return 0;
}
```

2.4.4 Statiska variabler

Man kan även deklarera en variabel som statisk genom att använda nyckelordet `static`:

```
static int i;
```

Värdet på en statisk variabel består till slutet av programmet.

```
#include <stdio.h>

void display()
{
    static int c = 1;
    c += 5;
    printf("d = %d\n",c);
}

int main()
{
    display();
    display();
```

```
    return 0;
}
```

Under det första anropet till `display` sätts värdet på `c` till 1, sedan ökas det med 5 så att dess värde blir 6. Under andra anropet till `display` sätts inte värdet på `c` till 1 igen utan det har fortfarande värdet 6 så att när vi lägger till 5 får vi 11.

2.4.5 Initialisera variabler

Variabler av olika typ initialiseras automatiskt till följande värden av kompilatorn:

```
Typ        Standardvärde
-----------------------
int            0
char           '\0'
float          0
double         0
pointer        NULL
```

Det är dock bra praxis att inte förlita sig på detta och initialisera variabler till något explicit. Annars kan oväntade saker hända om man har otur.

2.5 Arrayer (eng. *arrays*)

En array är en variabel som kan innehålla multipla värden. Till exempel, om vi vill spara 5 tal skriver vi:

```
int data[5];
```

Obs: storleken på en array kan inte ändras efter att den har deklarerats. På detta sätt skiljer sig arrayer från listor. Listor kan implementeras i C som länkade listor, dvs varje element innehåller även en pekare till nästa element i listan.

Man hämtar element från en array genom att hämta ut specifika index. Till exempel första elementet i `data` är `data[0]`, andra är `data[1]`, etc. Om storleken på en array är `n` så finns sista elementet på index `n-1`, så `data[4]` är sista elementet.

Om startadressen i minnet för `data[0]` är 2120 decimal då är addressen för `data[1]` 2124 decimal, addressen för `data[2]` är 2128 decimal, och så vidare. Detta är eftersom storleken på en `int` är 4 bytes.

Om man hämtar ut ett element som ligger utanför arrayen (t.ex. `data[12]`) så kan vad som helst hända. Ibland kraschar programmet och ibland verkar det fungera som man tänkt sig. Detta beror på att man helt enkelt hämtar ut ett godtyckligt värde i minnet och vad som finns där beror på vad som hänt i andra delar av programmet. Man måste därför vara försiktig att aldrig indexera utanför en array.

Man kan initialisera en array när man skapar den:

```
int data[5] = {19, 10, 8, 17, 9};
```

Man kan även skriva bara:

```
int data[] = {19, 10, 8, 17, 9};
```

Då vi inte specificerat storleken kommer kompilatorn lista ut storleken automatiskt och sätta den till 5 då den innehåller 5 element.

Vi kan ändra värdet på element i en array genom:

```
int data[5] = {19, 10, 8, 17, 9}
data[2] = -1;
data[4] = 0;
```

2.5.1 Skriva till och läsa från arrayer

Man kan läsa in data från en användare och skriva det direkt till en position i en array genom:

```
scanf("%d", &data[2]);
```

Man kan även skriva ut element från en array genom:

```
printf("%d", data[2]);
```

Nedan följer ett program som tar 5 värden från användaren och sparar dom i en array. Dessa skrivs sedan ut med hjälp av en loop.

```
#include <stdio.h>

int main()
{
    int values[5];
    printf("Enter 5 integers: ");

    for(int i = 0; i < 5; ++i) {
        scanf("%d", &values[i]);
    }

    printf("The array contains: ");

    for(int i = 0; i < 5; ++i) {
        printf("%d\n", values[i]);
    }

    return 0;
}
```

2.5.2 Flerdimensionella arrayer

Man kan även skapa arrayer av arrayer, så kallade flerdimensionella arrayer. Exempel:

```
int x[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int x[][3] = {{1, 3, 0}, {-1, 5, 9}};

int x[2][3] = {1, 3, 0, -1, 5, 9};
```

Här är x en tvådimensionell (2d) array. Den kan hålla 6 element och man kan tänka sig x som en tabell med 2 rader och 3 kolumner.

Man kan även initialisera en tredimensionell array på liknande sätt:

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

Vi kan använda denna typ av arrayer till en massa saker. Nedan följer ett exempel på ett program där vi sparar temperaturen i två städer:

```
#include <stdio.h>

int CITY = 2;
int WEEK = 7;

int main()
{
    int temperature[CITY][WEEK];

    for (int i = 0; i < CITY; ++i) {
        for (int j = 0; j < WEEK; ++j) {
            printf("City %d, Day %d: ", i + 1, j + 1);
            scanf("%d", &temperature[i][j]);
        }
    }

    printf("\nDisplaying values: \n\n");

    for (int i = 0; i < CITY; ++i) {
        for (int j = 0; j < WEEK; ++j) {
            printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
        }
    }

    return 0;
}
```

Nedan följer ett exempel på en 3d-array som innehåller 12 värden som ges av användaren.

```
#include <stdio.h>

int main()
{
    int test[2][3][2];

    printf("Enter 12 values: \n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                scanf("%d", &test[i][j][k]);
            }
        }
    }
```

```
    printf("\nDisplaying values:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
            }
        }
    }

    return 0;
}
```

2.5.3 Arrayparametrar till funktioner

Man kan även skicka in arrayer till funktioner. Nedan följer ett program för att beräkna summan av array element.

```
#include <stdio.h>

float calculateSum(float vals[], int len)
{
    float sum = 0.0;

    for (int i = 0; i < len; ++i)
        sum += vals[i];

    return sum;
}

int main()
{
    float result, vals[] = {23.4, 55, 22.6, 3, 40.5, 18};

    int len = sizeof(vals) / sizeof(float);

    result = calculateSum(vals,len);

    printf("Result = %.2f\n", result);

    return 0;
}
```

Vi kan även skicka flerdimensionella arrayer som argument till funktioner.

```
#include <stdio.h>

void displayNumbers(int num[2][2])
{
    printf("Displaying:\n");
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
```

```
            printf("%d\n", num[i][j]);
        }
    }
}

int main()
{
    int num[2][2];
    printf("Enter 4 numbers:\n");

    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    displayNumbers(num);

    return 0;
}
```

Obs: en funktion kan inte returnera en array i C! För detta (och mycket annat) måste vi istället använda pekare.

2.6 Pekare

Vi har redan sett att om vi har en variabel `var` i vårat program så kan vi få dess address i minnet genom `&var`. Vi har använt detta i `scanf`:

```
scanf("%d", &var);
```

Pekare (eller pekarvariabler) är speciella variabler som används för att spara addresser istället för värden. Så här deklarerar vi en pekare till en `int`:

```
int* p;
```

Vi kan även deklarera pekar på följande sätt:

```
int *p1;
int * p2;
```

Obs: om vi skriver följande

```
int* p1, p2;
```

så är bara `p1` en pekare medans `p2` bara är en **normal** variabel av typ `int`.

2.6.1 Pekarvärden

Vi kan även använda `*` för att få ut värdet från en pekare. Denna operator kallas för "dereference operator". Exempel:

```
#include <stdio.h>

int main()
{
```

```
    int* pc, c;

    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);  // 22

    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 22

    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 11

    *pc = 2;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 2

    return 0;
}
```

Detta program fungerar på följande sätt:

- `int* pc, c`: en pekare pc och en normal variabel c, båda av typ `int`, skapas. Då pc ej är initialiserad till något specifikt värde innehåller den slumpmässigt nonsens (dvs, pc kan peka på något ställe i minnet eller så pekar den på address som inte finns i minnet).
- `c = 22`: Talet 22 sparas på minnesaddressen för c.
- `pc = &c`: tilldelar addressen för c till pekaren pc.
- `*pc = 2`: ändrar värdet på addressen som pc pekar på (dvs, samma address som c).

2.6.2 Vanliga misstag när man programmerar med pekare

Om man vill att en pekare pc ska peka på addressen för c då är följande fel:

```
int main()
{
    int c, *pc;

    pc = c; // fel: pc är address men c är det inte

    *pc = &c; // fel: &c är en address med *pc är inte det

}
```

Ett exempel på pekarsyntax som är lite förvirrande:

```
#include <stdio.h>

int main()
{
```

```
    int c = 5;
    int *p = &c;
    printf("%d\n", *p);

    return 0;
}
```

Anledningen att detta fungerade är att

```
int *p = &c;
```

är detsamma som

```
int *p;
p = &c;
```

I båda fallen skapar vi en pekare p (inte `*p`) och tilldelar `&c` till den.

2.6.3 Arrayer och pekare

En array är ett block av sekventiell data som man kan hämtaut genom pekare.

```
#include <stdio.h>

int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;
    ptr = &x[2];

    printf("*ptr = %d\n", *ptr);
    printf("*(ptr+1) = %d\n", *(ptr+1));
    printf("*(ptr-1) = %d\n", *(ptr-1));

    return 0;
}
```

Här är `&x[2]` addressen till det tredje elementet och det tilldelas till `ptr`. Genom att addera och subtrahera till en pekare kommer den peka på något nytt (vad detta är beror på typen av pekaren). Så `*(ptr+1)` är det fjärde elementet och `*(ptr-1)` är det andra elementet i arrayen. Detta är ett exempel på *pekararitmetik*, dvs vi ändrar var en pekare pekar med hjälp av aritmetiska operatorer. Ett till exempel:

```
#include <stdio.h>

int main()
{
    int i, x[6], sum = 0;
    printf("Enter 6 numbers: ");

    for(i = 0; i < 6; ++i) {
        // Samma som scanf("%d", &x[i]);
        scanf("%d", x+i);
        // Samma som sum += x[i]
```

```
        sum += *(x+i);
    }

    printf("Sum = %d\n", sum);

    return 0;
}
```

I de flesta fall är arrayvariabler konverterade till pekare. Det är därför vi kan använda pekare för att manipulera dem. En array är dock inte detsamma som en pekare (en array har t.ex. en specificerad storlek och kan indexeras med `[i]`).

2.6.4 Pekarargument till funktioner

Vi kan ge pekare som parametrar till funktioner.

```
#include <stdio.h>

void addOne(int* ptr) {
    (*ptr)++;
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);
    printf("%d\n", *p);
    return 0;
}
```

Värdet som `p` pekar på är `10` till en början. Pekaren skickas sedan som argument till `addOne` vilken öker värdet på elementet sparat i `ptr` med 1. Då både `ptr` och `p` är pekare med samma address blir `*p` i `main` också 11.

Funktioner kan även returnera pekare. Men man bör inte returnera en pekare till lokala variabler!

2.6.5 `malloc` och `sizeof`

När man programmerar med pekare och arrayer vill man ofta allokera en sekvens bytes i minnet. För detta kan man använda funktionen `malloc` som allokerar ett givet antal bytes i minnet och returnerar addressen till början av det allokerade minnesutrymmet. För att göra detta behöver man ofta veta storleken som en typ av element tar i bytes, för detta kan man använda funktionen `sizeof`. Detta förklaras bäst med ett exempel. Koden nedan innehåller en funktion `inc` som tar in en array `arr` av längd `len` och returnerar en pekare till en minnessekvens med alla element i `arr` plus ett.

```
#include <stdio.h>
#include <stdlib.h>

int* inc(int arr[],int len) {

  // allokera lämplig mängd minne för det som ska returneras
```

```
  int* new_array = malloc(len*sizeof(int));

  for (int i = 0;i < len;i++) {
    *(new_array+i) = 1+arr[i];
  }

  return new_array;
}
```

Funktionen kan anropas på följande sätt:

```
int main() {

  int data[5] = {1,2,3,4,5};
  int l = sizeof(data) / sizeof(int); // Blir 5
  int* output = inc(data,l);

  for (int i = 0;i < 5;i++) {
    printf("%d\n",*(output+i));
  }
}
```

Notera hur längden på arrayen beräknas med hjälp av `sizeof`. På detta sätt kan man dynamiskt allokera rätt minnesmängd.

När man allokerar minne på detta sätt måste man vara försiktig så att inte minnet på datorn fylls upp. Detta är kanske ett mindre problem idag då datorer har så mycket minne, men traditionellt har detta varit ett stort problem. C program med så kallade "minnesläckor" är program som bara allokerar mer och mer minne när programmet körs vilket till slut gör datorn seg. För att undvika detta används funktionen `free` vilken tar in en pekare och frigör minne som "mallocats" när man är klar med det. Detta är något man bör ha koll om man programmerar C och en anledning till att högnivåspråk där man inte behöver tänka på denna typ av lågnivå detaljer utvecklades

2.7 Strängar

En sträng är en array av bokstäver avslutad med ett "nulltecken" `\0`.

```
char c[] = "Hej!";
```

När kompilatorn ser en sträng inom dubbelfnuttar (dvs, ") kommer den lägga till `\0` i slutet av strängen. Man kan initialisera en sträng på många andra sätt:

```
char c[50] = "abcd";
char c[] = {'a', 'b', 'c', 'd', '\0'};
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

Här är ett exempel på något man inte bör göra:

```
char c[2] = "abcde";
```

Vi försöker lägga 6 bokstäver (den sista är `\0`) till en sträng av längd 2. Detta är ingen bra idé då vi kommer skriva över minnesaddresserna efter `c[1]` och vad som helst kan sedan hända.

Man kan läsa in en sträng med `scanf`. Den läser då ett antal bokstäver tills den kommer till ett blanksteg (mellanslag, tab, radbrytning).

```
#include <stdio.h>

int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.\n", name);

    return 0;
}
```

På samma sätt som arrayer kan strängar konverteras till pekare. Detta betyder att vi kan använda pekare för att manipulera strängar.

```
#include <stdio.h>

int main(void)
{
    char name[] = "Anders";
    printf("%c\n", *name);
    printf("%c\n", *(name+1));
    printf("%c\n", *(name+4));

    char *namePtr;
    namePtr = name;
    printf("%c\n", *namePtr);
    printf("%c\n", *(namePtr+1));
    printf("%c\n", *(namePtr+3));

    return 0;
}
```

2.8 goto

goto-uttryck kan användas för att hoppa från en position i programmet till en specificerad “label”.

```
# include <stdio.h>

int main()
{
    int i;
    double number, average, sum=0.0;

    for(i=1; i<=5; ++i) {
        printf("%d. Enter a number: ", i);
        scanf("%lf",&number);
```

```
        if(number < 0.0)
            goto jump;
        sum += number;
    }
    jump:
    average=sum/(i-1);
    printf("Sum = %.2f\n", sum);
    printf("Average = %.2f", average);
    return 0;
}
```

Kod med `goto` kan var både buggig och svårläst. Intressant läsning är Edgar Dijkstras "*Go To Statement Considered Harmful*": https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf.

Mer om detta finns att läsa i "*Go To Statement Considered Harmful: A Retrospective*" av David R. Tribble: http://david.tribble.com/text/goto.html.

Bjarne Stroustrup, skaparen av C++, har även sagt att *"The fact that 'goto' can do anything is exactly why we don't use it."*.

Det är dock bra att veta om `goto` då det är en programkonstruktion som används flitigt i de flesta assemblerspråk. Dock bör man undvika att använda det i högnivåspråk.

2.8.1 Addera två matriser med hjälp av goto

Nedan följer ett program där två matriser adderas:

```
#include <stdio.h>

int main()
{
    int a[2][2] = { {1 , 2} , { 3, 4 } };
    int b[2][2] = { {6 , -2} , { 7, 4 } };

    int result[2][2];

    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            result[i][j] = a[i][j] + b[i][j];

    printf("Summan:\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            printf("%d ", result[i][j]);
            if (j == 1)
                printf("\n");
            }

    return 0;
}
```

Vi kan skriva om programmet så att det använder `goto` istället:

```c
#include <stdio.h>

int main()
{
    int a[2][2] = { {1 , 2} , { 3, 4 } };
    int b[2][2] = { {6 , -2} , { 7, 4 } };

    int result[2][2];

    int i = 0;
 outer:
    if (i < 2) {
      int j=0;
    inner:
      if (j<2) {
        result[i][j] = a[i][j] + b[i][j];
        ++j;
        goto inner;
      } else {
        goto endinner;
      }
    endinner:
      ++i;
      goto outer;
    } else {
      goto endouter;
    }
 endouter:

    printf("Summan:\n");
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            printf("%d ", result[i][j]);
            if (j == 1)
                printf("\n");
            }

    return 0;
}
```

Vilket program är mest lättläst?

2.9 Imperativ programmering, sammanfattning

Nu är vi klara med imperativ programmering i den här kursen. Som vi har sett är imperativ programmering ett paradigm där man ger instruktioner/uttryck som exekveras i ordning och vilka modifierar ett programs tillstånd. Imperativ programmering är nära sammankopplat med hur datorn fungerar och bygger på von Neumann's beräkningsmodell där en dator är en processor som gör beräkningar vilka modifierar register och minnen (dvs på det sätt som en vanlig dator fungerar).

Vi har lärt oss språket C vilket är både väldigt imperativt och maskinnära. När man programmerar i C måste man ha förståelse för hur datorn fungerar under skalet, t.ex. hur minneshantering fungerar. Detta är både en styrka och svaghet då man å ena sidan kan skriva väldigt snabb kod med å andra sidan kan introducera väldigt många typer av buggar. För att överkomma detta introducerades objektorienterade språk som C++, Java, Perl, Python, PHP, Ruby… Dessa språk är i grunden imperativa, men de är längre från maskinen och man har mindre kontroll över hur minneshanteringen fungerar (dvs, de är "högnivåspråk"). Ett viktigt koncept för just högnivåspråk är "garbage collection" (typ "skräpuppsamling") vilket betyder att minne som allokerats av programmet men inte används längre frias så att det kan användas igen. I C händer inte detta automatiskt och det är lätt att skriva program med "minnesläckor", dvs där mer och mer minne allokeras men aldrig frias så att man till slut använder upp allt minne.

2.10 Övningar

- Skriv om exemplet med städerna och temperaturen så att den använder strängar för stadsnamnen istället för `int`.
- Skriv om koden för matrisaddition så att den använder pekare istället för arrayer.
- Skriv om koden för matrisaddition så att den inte innhåller några loopar alls utan bara `goto`.
- Skriv om koden för matrisaddition så att additionen sker i en separat funktion. Kom ihåg att man inte kan returnera en array i C, så den måste antingen returnera en pekare eller ta in `result` arrayen som ett tredje argument där resultatet sparas.

Kapitel 3

Objektorienterad programmering i Java

3.1 Objektorienterad programmering

Objektorientering är ett programmeringsparadigm där man samlar datastrukturer med de funktioner som kan tillämpas på dem i klasser.

Fördelar:

- Uppmuntrar god struktur
- Bra för abstraktion
- Stark koppling till datamodellering

Detta åstadkoms genom att skapa en klass som packeterar data och funktionalitet (tänk att "klass" betyder typ "klassificering"). En klass är en ny typ och man kan skapa *instanser* av klasser (exempelvis är `[1,2,3]` en instans av klassen `list` i Python). Varje instans kan ha olika *attribut* kopplade till den för att representera dess nuvarande tillstånd (exempelvis listan ovan kan ha dess längd som attribut). Instanser kan även ha *metoder* för att modifiera dess tillstånd (ex. `xs.pop()` anropar `pop` metoden i `list` klassen).

3.2 Objektorientering med Java

Syntaxen i Java påminner till stor del om den från C, men det finns många viktiga skillnader i hur språken fungerar. Bland annat:

- Minneshantering – Java har "garbage collection" och man behöver inte hantera minne själv.
- Inga pekare i Java.
- Java är plattformsoberoende (mer om det senare).
- Java är objektorienterat.

3.2.1 Hello, World!

Nedan följer en kodsnutt för "Hello, World!":

```java
// Hello, World! in Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

För att köra programmet måste det läggas i en fil som heter "HelloWorld.java" och sedan kompileras och köras med:

```
> javac HelloWorld.java
> java HelloWorld
Hello, World!
```

Ovan använder vi `javac` för att generera en `.class` fil vilken vi sedan kör med `java`. I "Hello, World!" exemplet har vi använt följande koncept från Java:

- Kommentarer skrivs efter `//` eller inom `/*` och `*/` (så samma som i C).
- Java är väldigt objektorienterat, så i princip all kod man skriver måste tillhöra en klass. Här skapar vi en klass `HelloWorld` som är publik (mer om det senare). Den här klassen måste vara samma som filnamnet.
- På nästa rad ges typen för `main` metoden. Precis som i C är det denna kod som kommer köras när man kör programmet. Denna metod är publik och statisk (mer om det senare) och returnerar en `void` (dvs ingenting returneras). Den tar en array av strängar `args`, detta är argumenten som man ger till programmet när man kör det.
- `println` metoden från `System.out` anropas. Detta skriver ut en sträng och lägger till en ny rad i slutet.

När vi kör `javac` på det här programmet genereras inte assemblykod för din processor utan "Java Bytecode". Detta är ett språk som körs i Javas abstrakta maskin—Java Virtual Machine (JVM). Denna "maskin" kan köras på en mängd olika processorer och Java är alltså plattformsoberoende, dvs så länge man kan köra JVM så kan man köra Javakod. Man behöver alltså inte ha speciella kompilatorer för olika processorer när man kör Javakod.

3.2.2 Att programmera i Java

Många grundläggande saker som typer, aritmetiska operatorer, tilldelningsoperatorer, kodblock inom `{` och `}`, `if`-satser, `for` och `while` loopar fungerar på liknande sätt i Java som C:

```java
class Loop {
    public static void main(String[] args) {
        int i;
        for (i = 0; i <= 10; ++i) {
            if (i % 2 == 0) {
                System.out.println("Even " + i);
            } else {
                System.out.println("Odd " + i);
            }
        }
    }
}
```

För att läsa in indata från en användare kan vi använda en Scannner (för dokumentation se https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html).

```
import java.util.Scanner;

class Input {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);
    }
}
```

Java har även arrayer, precis som C.

3.2.3 Objektorientering i Java

Som vi sett ovan är Java objektorienterat och i princip all kod tillhör en klass. Nedan följer ett exempel med en metod för att beräkna kvadraten av ett tal. När koden ligger i samma klass på detta sätt är `getSquare` som en vanlig funktion i C.

```
public class SquareMethod {

    public static int getSquare(int x) {
        return x * x;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            // method call
            result = getSquare(i)
            System.out.println("Square of " + i + " is : " + result);
        }
    }
}
```

Låt oss gå igenom `public static int getSquare(int x)` ord för ord:

- `public` betyder att metoden är synlig överallt i programmet.
- `static` betyder att man kan använda metoden utan att skapa en instans av klassen (på så sätt är det som en vanlig funktion).
- `int` är returtypen.
- `getSquare` är namnet på metoden.
- `int x` är argumentet till metoden.

3.3 Kurs i Java

Låt oss skriva en klass för att representera kurser i Java.

```
import java.util.List;
import java.util.ArrayList;

public class Course {

    public List<String> participants;
    public String code;
    public String name;
    public int year;

    public Course(String code, String name, int year) {
        participants = new ArrayList<>();
        this.code = code;
        this.name = name;
        this.year = year;
    }

    public int numberOfParticipants() {
        return participants.size();
    }

    public void addParticipant(String name) {
        participants.add(name);
    }

    public String toString() {
        return ("<Course: " + code + " " + name + " (" + year + ")>");
    }

    public static void main(String[] args) {
        Course mm2001 = new Course("MM2001", "Matematik", 2020);

        System.out.println(mm2001.numberOfParticipants());

        mm2001.addParticipant("Anders");

        System.out.println(mm2001.numberOfParticipants());

        Course da4003 = new Course("DA4003", "ProgP", 2021);

        System.out.println(da4003.numberOfParticipants());
        System.out.println(da4003.year);
        System.out.println(da4003);
        System.out.println(mm2001);
    }
}
```

I det här exemplet ser vi en hel del nya koncept från Java. Låt oss kommentera koden från början till slut:

- Först importerar vi List interfacet och en implementation med hjälp av arrayer.
- Sedan skapar vi en klass `Course` med fyra publika attribut.
- Sedan har vi konstruktorn för klassen. Den har samma namn som klassen och ingen returtyp. Konstruktorn tar tre argument.
- Vi använder `new` för att skapa ett nytt objekt av ArrayList typen.
- Vi använder sedan `this` för att modifiera attributen i klassen (detta är som `self` i Python). Vi behöver ingen `this` för `participants` då den ej är ett argument till konstruktorn.
- `toString()` är Javas svar på Pythons `__str__`.
- I `main` konstruerar vi sedan instanser av `Course` klassen med `new` och modifierar dem med metoderna för klassen. Notera att när vi skriver ut `da4003` och `mm2001` så anropas deras `toString()` metoder automatiskt.

3.3.1 Privata och publika metoder/attribut

Metoder och attribut kan vara privata i Java. Följande exempel fungerar:

```
class MyClass {
    public String x = "hej!";
}

public class MyTest {
    public static void main(String[] args) {
        MyClass foo = new MyClass();
        System.out.println(foo.x);
    }
}
```

Men om vi ändrar till

```
class MyClass {
    private String x = "hej!";
}

public class MyTest {
    public static void main(String[] args) {
        MyClass foo = new MyClass();
        System.out.println(foo.x);
    }
}
```

får vi ett felmeddelande. På så sätt kan vi gömma attribut och metoder i klasser i Java. Detta leder till abstraktion och är god praxis. I Java brukar man ha “getters” och “setters” för att hämta och ändra värdet på privata variabler i klasser:

```
class MyClass {
    private String x = "hej!";
```

```
    public String getX() {
        return x;
    }

    public void setX(String x) {
        this.x = x;
    }
}

public class MyTest {
    public static void main(String[] args) {
        MyClass foo = new MyClass();
        System.out.println(foo.getX());
        foo.setX("bar!");
        System.out.println(foo.getX());
    }
}
```

3.3.2 Instans och klassvariabler i Java

Java skiljer även även på instans och klassvariabler:

```
public class Dog {

    public String name;
    public static String kind = "canine";

    public Dog(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Dog d = new Dog("Fido");
        Dog e = new Dog("Buddy");

        System.out.println(d.kind);
        System.out.println(e.kind);
        System.out.println(d.name);
        System.out.println(e.name);
        d.kind = "K9";
        System.out.println(d.kind);
        System.out.println(e.kind);
    }
}
```

I exemplet är `name` en instansattribut då den är kopplad till varje instans, medans `kind` är ett klassattribut vilken delas av alla instanser. Detta ses genom att när vi ändrar värden på `d.kind` så ändras även `e.kind`. Genom att använda `static` beter dessa sig alltså dessa som statiska variabler i C.

3.3.3 Kurs med deltagare i Java

Låt oss nu lägga till deltagare till kursexemplet:

```
import java.util.List;
import java.util.ArrayList;

class Participant {
    private String name;
    private String email;

    public Participant(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String toString() {
        return ("<" + name + ", " + email + ">");
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
}

class Course {

    public List<Participant> participants;
    public String code;
    public String name;

    public Course(String code, String name) {
        participants = new ArrayList<>();
        this.code = code;
        this.name = name;
    }

    public int numberOfParticipants() {
        return participants.size();
    }

    public void addParticipant(Participant d) {
        participants.add(d);
    }

    public List<String> nameList() {
```

```
        List<String> nameList = new ArrayList<String>();

        int n = participants.size();

        for (int i = 0; i < n; i++) {
            nameList.add(participants.get(i).getName());
        }

        return nameList;
    }

    public String toString() {
        return ("<Course: " + code + " " + name + ">");
    }
}

public class f3 {
    public static void main(String[] args) {
        Course da4003 = new Course("DA4003", "ProgP");

        Participant d = new Participant("Anders", "anders.mortberg@math.su.se");

        da4003.addParticipant(d);

        da4003.addParticipant(new Participant("Foo Bar", "foo@bar.com"));

        System.out.println(da4003.nameList());
    }
}
```

Genom att göra `Participant` till en egen klass är det nu lätt att ändra på den utan att annan kod behöver ändras. Exempelvis kan man lätt lägga in en attribut som indikerar om en `Participant` är lärare eller inte. Detta kan göas på ett sånt sätt att de metoder vi redan har är oförändrade, vilket betyder att kod som bara använder dessa metoder inte behöver ändras. På detta sätt leder objektorientering till robust kod där ändringar kan göras utan att hela programmet behöver skrivas om.

3.4 Ordlista objektorienterad programmering

- Klass: en typ innehållande attribut och metoder
- Metod: en funktion i en klass
- Instansiering: ett objekt av en specifik klass skapas
- Objekt: en instans av en klass
- Konstruktor: metod för att instansiera ett objekt av klassen
- Klassattribut: delas av alla instanser av klassen
- Instansattribut: tillhör en specifik instans av klassen

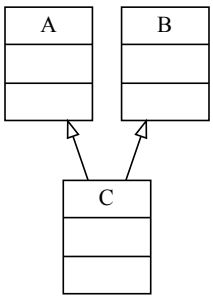
3.5 Arv i objektorienterad programmering

Ett väldigt viktigt koncept inom objektorienterad programmering är *arv*. Det handlar om att man kan skriva klasser som ärver attribut och metoder från en "föräldraklass" eller "superklass". Hur detta är implementerat och vilken typ av arv som tillåts skiljer sig mellan olika språk.

Det finns fem typer av arv:

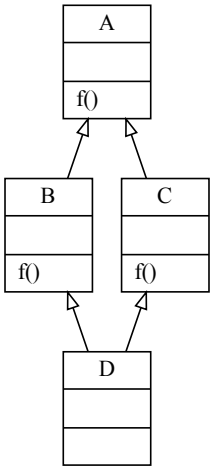
- Enkelt arv (eng. *single inheritance*): klass `B` ärver endast från klass `A`
- Mångnivå arv (eng. *multilevel inheritance*): klass `B` ärver från `A` och klass `C` ärver sen från klass `B`
- Hierarkiskt arv (eng. *hierarcical inheritance*): klass `A` är superklass till `B`, `C`, `D`...
- Multipelt arv (eng. *multiple inheritance*): klass `C` ärver från *både* `A` och `B` (men de ärver inte från varandra)
- Hybrid arv (eng. *hybrid inheritance*): en mix av två eller fler typer av arv

Man kan illustrera ovanstående former av arv genom att rita klassdiagram. Exempelvis kan multipelt arv ritas som i Figur 3.1.



Figur 3.1: Klassdiagram för `A`, `B` och `C`

Ett problem med multipelt arv är att det leder till så kallade "diamantproblem". Detta illustreras i Figur 3.2. Här finns fyra klasser och både `A`, `B` och `C` har samma metod `f()`. När man instansierar `D` kommer alla dessa metoder ärvas och frågan är då vilket `f()` som anropas från `D`? Denna typ av problem leder lätt till rörig kod och mystiska buggar där en annan metod än den man tror anropas. På grund av detta stödjer Java inte multipelt arv, men det gör Python. Vilken metod som anropas i Python beror på i vilken ordning man skriver att `D` ärver från `B` och `C`.



Figur 3.2: Klassdiagram för `A`, `B`, `C` och `D`

En bra tumregel för hur man kan designa komplexa program med många komplexa klasser är Liskov's substitutionsprincip: allt som gäller för en klass ska gälla för dess subklasser.

Detta är nära relaterat till konceptet "subtyper" (eng. *subtyping*) vilket används i språk med kraftfulla typsystem som ser till att man inte bryter mot Liskov's princip. Många språk är dock väldigt flexibla och man kan lätt bryta mot principen, vilket i sin tur lätt leder till rörig kod. Det finns många böcker om hur man designar bra klasser och hierarkier av klasser, dvs hur man skriver bra objektorienterad kod.

3.6 Arv i Java

Arv i Java fungerar till stor del som i Python, men det finns några skillnader. Bland annat stöds inte multipelt arv i Java. Sen har Java en del egenheter som styr hur man kan definiera och instansiera klasser som inte Python har. Dessa gör det lite mer komplicerat att skriva kod i Java, men de tvingar en även att tänka lite mer när man konstruerar klasser. Bland annat finns så kallade "abstrakta" klasser vilka man inte kan instansiera, detta kan verka lite konstigt om man kommer från Python men vi ska se ett användningsområde nedan.

Innan vi går in på arv i Java ska vi prata lite om varför det finns två versioner av t.ex. heltal och sanningsvärden i Java. När ni jobbar med labb 2 kan ni stöta på både `int` och `Integer`, `boolean` och `Boolean`, osv. Anledningen är att Javas parametriserade typer (som listor och uppslagstabeller vilka fungerar för olika sorters typer av element) bara tar objekt som ärver från Javas `Object` klass. Detta gör

inte primitiva typer som `int` och `boolean`, så vi måste istället använda `Integer` och `Boolean` vilka helt enkelt är klasser som innehåller ett heltal eller ett sanningsvärde. Java konverterar ofta automatiskt en `int` till en `Integer` (detta kallas ibland för "autoboxing") så att man inte behöver tänka på detta, men när man skriver typsignaturer måste man tänka på att skriva t.ex `List<Integer>` och inte `List<int>`.

3.6.1 Geometriska figurer i Java

För att skriva en superklass för geometriska figurer skriver vi:

```
class Shape {
    private String color;
    private boolean filled;

    public Shape(String color, boolean filled) {
        this.color = color;
        this.filled = filled;
    }

    public String get_color() {
        return color;
    }

    public void set_color(String color) {
        this.color = color;
    }

    public boolean get_filled() {
        return filled;
    }

    public void set_filled(boolean filled) {
        this.filled = filled;
    }
}
```

Här har vi två privata variabler för färgen och om figuren är ifylld eller inte. Dessa kan hämtas och ändras med "getters" och "setters".

För att skapa en klass för rektanglar vilken ärver från Shape skriver vi:

```
class Rectangle extends Shape {

    private double length, height;

    public Rectangle(String color,boolean filled,double length, double height) {
        super(color,filled);
        this.length = length;
        this.height = height;
    }

    public double get_length() {
```

```
        return length;
    }

    public void set_length(double length) {
        this.length = length;
    }

    public double get_height() {
        return height;
    }

    public void set_height(double height) {
        this.height = height;
    }

    public double get_area() {
        return length * height;
    }

    public double get_circumference() {
        return 2 * (length + height);
    }
}
```

Precis som i Python använder vi `super` för att anropa konstruktorn i superklassen. Vi kan även använda `super` för att komma åt metoder från superklassen.

En cirkel kan definieras på liknande sätt:

```
class Circle extends Shape {

    private double radius;

    public Circle(String color,boolean filled,double radius) {
        super(color,filled);
        this.radius = radius;
    }

    public double get_radius() {
        return radius;
    }

    public void set_radius(double radius) {
        this.radius = radius;
    }

    public double get_area() {
        return 3.14 * radius * radius;
    }
```

```
    public double get_circumference() {
        return 2 * 3.14 * radius;
    }
}
```

Vi kan även skapa typen av fyrkanter vilken ärver från `Rectangle` (och då även från `Shape`):

```
class Square extends Rectangle {

    public Square(String color,boolean filled,double side) {
        super(color,filled,side,side);
    }

    public double get_side() {
        return this.get_length();
    }

    public void set_side(double side) {
        this.set_length(side);
        this.set_height(side);
    }
}
```

För att slutligen kunna köra lite tester skapar vi en klass som heter samma som filen (`f3.java`) med en `main` metod i:

```
public class f3 {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle("black",false,10,3);
        System.out.println(rect.get_area());
        System.out.println(rect.get_circumference());
        System.out.println(rect.get_color());
        System.out.println(rect.get_filled());
        rect.set_filled(true);
        System.out.println(rect.get_filled());
        rect.set_color("orange");
        System.out.println(rect.get_color());

        Circle circ = new Circle("black",false,2);
        System.out.println(circ.get_area());
        System.out.println(circ.get_circumference());
        System.out.println(circ.get_color());
        System.out.println(circ.get_filled());

        Square square = new Square("black",false,5);
        System.out.println(square.get_area());
        System.out.println(square.get_side());
        System.out.println(square.get_color());
    }
}
```

3.6.2 Binära träd och abstrakta klasser i Java

Vi kan implementera en klass `BinTree` för binära träd i Java på följande sätt:

```
class BinTree {
}

class Branch extends BinTree {

    private Integer node;
    private BinTree left;
    private BinTree right;

    public Branch(Integer node, BinTree t1, BinTree t2) {
        this.node = node;
        left = t1;
        right = t2;
    }

    public String toString() {
        return "Branch(" + node + "," + left + "," + right + ")";
    }
}

class Leaf extends BinTree {

    private Integer val;

    public Leaf(Integer val) {
        this.val = val;
    }

    public String toString() {
        return "Leaf(" + val + ")";
    }
}

public class f3 {
    public static void main(String[] args) {
    BinTree t =
        new Branch(-32,
                   new Leaf(2),
                   new Branch(1,
                              new Branch(23,
                                         new Leaf(4),
                                         new Leaf(-2)),
                                         new Leaf(12)));

        System.out.println(t);
    }
```

```
}
```

Observera att `BinTree` klassen är tom. Om vi skulle vilja lägga till en `member` metod vilken testar om ett element finns i trädet till `Branch` kanske vi försöker att lägga in:

```
    public boolean member(Integer x) {
        return (node == x || left.member(x) || right.member(x));
    }
```

Men gör vi det får vi ett felmeddelande då `left` och `right` kanske inte har någon metod som heter `member`! För att lösa detta måste vi specificera att alla klasser som ärver från `BinTree` har en sån metod. Vi kan göra detta genom att lägga till en *abstrakt* metod till `BinTree`, dvs en metod som inte har en implementation i `BinTree` klassen men vilken alla klasser som ärver `BinTree` måste implementera. När vi gör detta måste vi även göra `BinTree` till en abstrakt klass, dvs en klass som man inte kan instansiera. Anledningen att vi inte ska kunna skapa en instans av `BinTree` är att vi inte ska kunna få en instans som innehåller en abstrakt metod (dvs en funktion utan någon kropp). För att göra både `BinTree` och `member` abstrakta skriver vi:

```
abstract class BinTree {
    abstract boolean member(Integer x);
}
```

Att `BinTree` inte går att instansiera är inget problem då vi aldrig vill skapa en instans av just `BinTree` utan vi kommer istället alltid skapa instanser av `Branch` eller `Leaf`. I Python finns det inga abstrakta klasser eller metoder vilket betyder att om vi glömmer bort att implementera `member` metoden i något klass som ärver `BinTree` så kommer vår kod kanske krascha när vi kör den. Java är alltså mycket säkrare än Python när det kommer till vilka metoder man kan anropa och vi kan gardera oss från att anropa metoder som inte finns.

Av samma anledning som att vi måste göra `member` till en abstrakt metod i `BinTree` så måste vi även göra metoden `linearize` vilken konverterar ett träd till en lista abstrakt. Nedan har ni komplett kod för exemplet med både `member` och `linearize`.

```java
import java.util.List;
import java.util.ArrayList;

abstract class BinTree {
    abstract boolean member(Integer x);
    abstract List<Integer> linearize();
}

class Branch extends BinTree {

    private Integer node;
    private BinTree left;
    private BinTree right;

    public Branch(Integer node, BinTree t1, BinTree t2) {
        this.node = node;
        this.left = t1;
        this.right = t2;
    }

    public String toString() {
        return "Branch(" + node + "," + left + "," + right + ")";
    }

    public boolean member(Integer x) {
        // Calling member here would not work unless it was abstract above
        return (node == x || left.member(x) || right.member(x));
    }

    public List<Integer> linearize() {
        List<Integer> res = new ArrayList<>();
        res.addAll(left.linearize());
        res.add(node);
        res.addAll(right.linearize());
        return res;
    }
}

class Leaf extends BinTree {

    private Integer val;

    public Leaf(Integer val) {
        this.val = val;
    }

    public String toString() {
        return "Leaf(" + val + ")";
    }
```

```
    public boolean member(Integer x) {
        return x == val;
    }

    public List<Integer> linearize() {
        List<Integer> res = new ArrayList<>();
        res.add(val);
        return res;
    }
}

public class f3 {
    public static void main(String[] args) {
        BinTree t =
          new Branch(-32,
                     new Leaf(2),
                     new Branch(1,
                                new Branch(23,
                                           new Leaf(4),
                                           new Leaf(-2)),
                                           new Leaf(12)));


        System.out.println(t);
        System.out.println(t.member(-2));
        System.out.println(t.member(-20));
        System.out.println(t.linearize());
    }
}
```

Notera att dessa träd egentligen inte har något med `Integer` att göra utan kan användas för alla typer med en likhetsoperator (vilket krävs för att `member` ska fungera). Man kan generalisera koden med hjälp av "generics" i Java vilket gör att vi får en typ `BinTree<A>` där `A` är en "typvariabel" (dvs vilken typ som helst). Detta kallas för *polymorfism* och vi kommer prata mer om det när vi kommer in på funktionell programmering och Haskell.

3.7 Övningar

- Lägg till attributen `teacher` av typ `boolean` till klassen `Participant`. Skriv ut lärarens namn på något speciellt sätt så att man från `participants` listan ser vem det är.
- Lägg till fler geometriska figurer som ärver `Shape` (t.ex. kuber).
- Lägg till en metod för att beräkna hur många element ett binärt träd innehåller.

Kapitel 4

Händelsestyrd och webbprogrammering i JavaScript 1

4.1 Händelsestyrd och webbprogrammering

Vi ska nu prata lite om webbprogrammering i JavaScript. Inom webb/internetprogrammering använder man sig ofta av en modell där man har en server och ett antal klienter som kommunicerar med servern. Exempelvis är en chat ett program som bygger på denna modell, man skickar meddelanden till servern vilka sedan distribueras till alla användare/klienter. Typiskt för denna typ av program är att man använder *sockets* för att hantera datakommunikationen. I tidigare versioner av kursen har internetprogrammeringsdelen handlat om socketprogrammering. Detta kräver dock en hel del förkunskaper om nätverk (TCP, UDP, portar, IP nummer, etc) vilket så klart är bra att kunna, men det har inte direkt något med programmeringsparadigm att göra.

På grund av detta kommer vi istället fokusera på webbprogrammering i bred mening vilket sker på en högre nivå och där man inte behöver veta en massa detaljer om nätverk och protokoll. En anledning att fokusera på webbprogrammering är att det hänger ihop med paradigmet *händelsestyrd programmering* (eng. *event driven programming*). Inom händelsestyrd programmering styrs programflödet av olika händelser, exempelvis genom knapptryckningar, mustryckningar, meddelanden från andra program, trådar eller skickade över nätverk, etc. Detta paradigm är en viktig del av GUI (*graphical user interface*) programmering och webbsidor (om man trycker på "Log in" på gmail så körs en kodsnutt som kollar att man skrivit in rätt lösenord).

4.2 JavaScript

För att lära oss om händelsestyrd och webbprogrammering ska vi använda JavaScript (JS). Tillsammans med HTML och CSS utgör detta språk nyckeltekniken bakom webben (WWW - World Wide Web). I princip alla hemsidor använder HTML för att beskriva upplägget, CSS för designen och JS för att göra sidan interaktiv. Jag kommer anta att alla har grundläggande kunskaper i HTML och vet hur man skriver väldigt enkla hemsidor. Om man vill eller behöver fräscha upp sina kunskaper kan man läsa på https://www.w3schools.com/html/. På samma sida finns det även en bra och väldigt detaljerad tutorial för JavaScript: https://www.w3schools.com/js/. Man behöver dock inte kunna något om CSS då det här kursmomentet inte handlar om webbdesign.

JS är ett högnivå- och multiparadigmspråk. Syntaxen påminner om C och Java, men det är dynamiskt typat som Python. Det stödjer även både imperativ och funktionell programmering och är objektorienterat. JS är alltså en mix av många av de paradigm och idéer som vi ska se i kursen.

Obs: det finns många likheter mellan JavaScript och Java utöver namnet, men språken är ändå väldigt olika i sin design och har utvecklats som helt olika projekt. Namnet valdes av företaget Netscape när de uppfann språket mest för att Java var det nya coola språket på 1990-talet och de hoppades väl på att JS skulle bli en succé om de tog ett liknande namn. Den som är intresserad av mer historia kan läsa på https://en.wikipedia.org/wiki/JavaScript#History.

JavaScript kördes traditionellt endast i webbläsare, men nu för tiden kan man köra JavaScript utanför webbläsaren. Ett sätt att göra detta är med hjälp av Node.js. I den här kursen kommer vi dock endast fokusera på webbprogrammering och all JS kod kommer köras i webbläsaren. Alla större webbläsare (Firefox, IE, Chrome…) stödjer JavaScript, så ingen bör ha några problem med installation.

4.2.1 Hello World!

Som vanligt börjar vi med att skriva "Hello World!". För att göra detta skriver vi lite HTML för en väldigt enkel hemsida och lägger in JavaScript koden i `<script>` taggar:

```
<html>
<head>
  <title>Hello World!</title>
</head>
<body>

  <script>
    document.write('Hello World!');
  </script>

</body>
</html>
```

För att köra koden sparar vi programmet i en `.html` fil och öppnar med en webbläsare. I exemplet skriver vi "Hello World!" till sidan med hjälp av `document.write`. Vi hade även kunnat gjort det genom `<p>Hello World!</p>` i HTML.

Vi kan även använda `alert` vilket skapar en ruta istället:

```
<html>
<head>
  <title>Hello World!</title>
</head>
<body>

  <script>
    alert('Hello World!');
  </script>

</body>
</html>
```

Obs: kommentarer i JS skrivs med `//` eller `/*` följt av `*/`. Men i HTML skrivs dom inom `<!--` följt av `-->`:

```
<html>
<body>

  <!-- HTML kommentar -->

  <script>
    // Kort JS kommentar

    /* Lång
       JS
       kommentar
    */
  </script>

</body>
</html>
```

4.2.2 Hello World! med knapp

Vi kan även lägga JavaScript kod i headern av HTML filen:

```
<html>
  <head>
    <title>Another example</title>

    <script>
      function myFunction() {
          document.getElementById("test").innerHTML = "World!";
      }
    </script>
  </head>
  <body>

    <p id="test">Hello</p>
    <button type="button" onclick="myFunction()">Press me!</button>

  </body>
</html>
```

Här ser vi ett exempel på en funktion `myFunction` som hämtar elementet med `id=test"` i HTML dokumentet och ändrar dess värde till "World!". Vi ser även att vi kan lägga till knappar vilka anropar funktioner när man klickar på dom. Detta är ett exempel på händelsestyrd programmering: programflödet påverkas av vad användaren gör.

JavaScript kod kan även läggas i filer. Om vi har en fil `myScript.js` med:

```
function myFunction() {
    document.getElementById("test").innerHTML = "World!";
}
```

så kan vi ladda den i vår HTML sida genom att lägga in `<script src=myScript.js"></script>` (i antingen <head> eller <body>). Externa script är bra då samma kod kan delas mellan flera sidor och man uppnår separation mellan HTML och JavaScript.

Vi kan även ha flera knappar som anropar olika funktioner:

```
<html>
  <head>
    <title>Hello World!</title>

    <script>
      function hide() {
          document.getElementById("test").style.display = "none";
      }

      function show() {
          document.getElementById("test").style.display = "block";
      }
    </script>
  </head>
  <body>

    <button type="button" onclick="show()">Show me!</button>
    <button type="button" onclick="hide()">Hide me!</button>

    <p id="test">Hello World!</p>

  </body>
</html>
```

Här har vi två knappar vilka antingen visar texten eller döljer den genom att ändra dess `style.display` attribut.

4.3 Variabler

Vi kan definiera variabler med `var`:

```
<html>
  <head>
    <script>
      function show() {
          var name = "Anders";
          document.getElementById("test").innerHTML = "Hello " + name;
      }
    </script>
  </head>
  <body>

    <button type="button" onclick="show()">Show me!</button>
```

```
    <p id="test"></p>

  </body>
</html>
```

Vi ser även att vi kan konkatenera strängar med hjälp av +. Strängar kan även skrivas inom enkelfnuttar (`'hej'`) precis som i Python.

JavaScript stödjer de vanliga räknesätten

```
<html>
  <head>
    <script>
      function show() {
          var x = 1 + 1;
          document.getElementById("test").innerHTML = "1 + 1 = " + x;
      }
    </script>
  </head>
  <body>

    <button type="button" onclick="show()">Show me!</button>

    <p id="test"></p>

  </body>
</html>
```

Obs: vi adderar talet `x` till strängen `"1 + 1 = "`. JavaScript konverterar alltså automatiskt talet `x` till en sträng för att göra adderingen.

Notera att vi inte behöver ge `x` eller `name` någon typ. Detta är ännu ett sätt som JavaScript påminner om Python.

JavaScript stödjer som sagt de vanliga räknesätten. Det finns även tilldelningsoperatorer, jämförelseoperatorer, logiska operatorer och bitoperatorer (man kan läsa om alla dessa på https://www.w3schools.com/js/):

```
<html>
  <head>
    <script>
      function show() {
          var x = 1 + 1;
          x += 5;
          x--;
          document.getElementById("test1").innerHTML = "x = " + x;
          document.getElementById("test2").innerHTML = (x == 1) || (x < 3);
          document.getElementById("test3").innerHTML = (x == x);
      }
    </script>
  </head>
  <body>
```

```
    <button type="button" onclick="show()">Show me!</button>

    <p id="test1"></p>
    <p id="test2"></p>
    <p id="test3"></p>
  </body>
</html>
```

Som vi ser är `true`/`false` de boolska värdena i JavaScript.

Då JavaScript automatiskt konverterar värden mellan olika typer kan konstiga saker ibland hända:

```
<html>
  <head>
    <script>
      function show() {
          document.getElementById("test1").innerHTML = "x = " + 1 + 1;
          document.getElementById("test2").innerHTML = 1 + 1 + " = x";
      }
    </script>
  </head>
  <body>

    <button type="button" onclick="show()">Show me!</button>

    <p id="test1"></p>
    <p id="test2"></p>

  </body>
</html>
```

I den första rader konverteras `1` till en sträng innan additionen. I den andra raden hanteras dom som tal innan de konverteras till strängar. Detta kan ge väldigt förvånande resultat.

Variabler i JavaScript är dynamiskt typade. Detta betyder att en variabel kan ha olika typer i olika delar av programmet. Detta skiljer sig från t.ex. C och Java.

```
<html>
  <head>
    <script>
      function show() {
          var x;
          document.getElementById("test1").innerHTML = "x = " + x;
          x = 5;
          document.getElementById("test2").innerHTML = "x = " + x;
          x = "hej";
          document.getElementById("test3").innerHTML = "x = " + x;
      }
    </script>
  </head>
  <body>
```

```
    <button type="button" onclick="show()">Show me!</button>

    <p id="test1"></p>
    <p id="test2"></p>
    <p id="test3"></p>

  </body>
</html>
```

Obs: som vi ser när vi klickar på knappen är värdet på en oinitierad variabel är `undefined`.

4.3.1 typeof

Man kan använda `typeof` för att få fram vilken typ ett värde har:

```
<html>
  <body>
    <script>
      document.write(typeof "Anders" + "<br>");
      document.write(typeof 3.14 + "<br>");
      document.write(typeof true + "<br>");
      document.write(typeof false + "<br>");
      document.write(typeof x + "<br>");
    </script>
  </body>
</html>
```

Obs: då `x` är `undefined` är även dess typ det.

4.4 Funktioner

Som vi sett ovan kan vi skriva funktioner med `function`. Notera att vi inte behöver skriva några typer för varken returvärde eller argument. Här kommer en funktion som konverterar från fahrenheit till celsius:

```
<html>
  <head>
    <script>
      function toCelsius(f) {
          return (5 / 9) * (f - 32);
      }
    </script>
  </head>
  <body>

    <script>
      document.write('32F is ' + toCelsius(32) + 'C');
    </script>
```

```
  </body>
</html>
```

Funktioner är som vilka värden som helst i JavaScript och om man inte har några paranteser efter ett funktionsanrop får man ut ett funktionsvärde:

```
<html>
  <head>
    <script>
      function toCelsius(f) {
          return (5 / 9) * (f - 32);
      }
    </script>
  </head>
  <body>

    <script>
      document.write('32F is ' + toCelsius + 'C');
    </script>

  </body>
</html>
```

Detta är typiskt för funktionella språk och vi kommer titta på det närmare i nästa paradigm.

4.5 Input-output

Vi kan enkelt läsa in indata från användaren och anropa vår konverteringsfunktion:

```
<html>
  <head>
    <script>
      function toCelsius(f) {
          return (5 / 9) * (f - 32);
      }

      function showTemp() {
          var x = document.getElementById("temp").value;
          document.getElementById("tempc").innerHTML =
              "Temperature (C): " + toCelsius(x);
      }
    </script>
  </head>
  <body>

    Temperature (F): <input type="text" id="temp" value="Write a number">

    <button onclick="showTemp()">Convert to C</button>

    <p id="tempc"></p>
```

```
  </body>
</html>
```

4.6 Loopar

JavaScript är även väldigt imperativt och om man vill göra något många gånger gör man det enklast med en loop. Det finns flera olika sorters loopar i JS.

4.6.1 For loopar

Vi har samma sorts `for` loopar som i C och Java:

```
<html>
  <body>

    <p id="test"></p>

    <script>
      var text = "";
      var i;

      for (i = 0; i < 5; i++) {
          text += "The number is " + i + "<br>";
      }

      document.getElementById("test").innerHTML = text;
    </script>

  </body>
</html>
```

Obs: kodsnutten `<p id=test"></p>` måste komma innan JS koden annars finns det inget element med `id=test"` att uppdatera.

Vi kan även lägga till följande kodsnutt för att skriva `Hello World!` 10 gånger:

```
<script>
for (var i = 0; i < 10; ++i) {
    document.write(i + ": Hello World!<br>");
}
</script>
```

4.6.2 For-of loopar

Det finns även loopar som påminner om dem i Python och vilka låter oss loopa över någon itererbar typ. Detta inkluderar strängar:

```
<html>
  <body>
    <script>
```

```
      var txt = 'Hej hopp!';
      var c;

      for (c of txt) {
          document.write(c + "<br>");
      }
    </script>
  </body>
</html>
```

Nästa vecka ska vi titta på andra itererbara typer, bland annat arrayer.

4.6.3 While loopar

Det finns även `while` loopar:

```
<html>
  <body>

    <p id="test"></p>

    <script>
      var text = "";
      var i = 0;

      while (i < 10) {
          text += "The number is " + i + "<br>";
          i++;
      }

      document.getElementById("test").innerHTML = text;
    </script>

  </body>
</html>
```

Precis som i C och Java finns det även `do-while` loopar.

4.7 if-else

Som i C/Java/Python finns det `if-else` och `else if` som kan användas för att anpassa kontrollflödet i programmet. Nedan följer ett litet program som printar ut en hälsning baserad på vad klockan är:

```
<html>
  <body>

    <p>Tryck på knappen för att få en tidsbaserad hälsning:</p>

    <button onclick="showGreeting()">Klicka här!</button>

    <p id="test"></p>
```

```
    <script>
      function showGreeting() {
          var greeting;
          var time = new Date();

          if (time.getHours() < 10) {
              greeting = "God morgon";
          } else if (time.getHours() < 20) {
              greeting = "God dag";
          } else {
              greeting = "God kväll";
          }

          document.getElementById("test").innerHTML = greeting;
      }
    </script>

  </body>
</html>
```

Vi använder `new` för att skapa ett `Date` objekt och hämtar ut `hours` attributen. JS är objektorienterat och man kan läsa mer om `Date` objekt på https://www.w3schools.com/js/js_dates.asp.

4.7.1 FizzBuzz

Ett vanligt test på programmeringsintervjuer är att skriva "FizzBuzz" programmet. Det ska fungera så att man skriver ut alla tal mellan `1` och `100`, men för alla tal som är delbara med `3` skrivs istället `Fizz`, för alla tal som är delbara med `5` skrivs `Buzz` och för alla tal som är delbara med både `3` och `5` skrivs `FizzBuzz`. Nedan följer en lösning på detta problem i JavaScript:

```
for (var i=1; i <= 100; i++) {
    if (i % 15 == 0)
        document.write("FizzBuzz<br>");
    else if (i % 3 == 0)
        document.write("Fizz<br>");
    else if (i % 5 == 0)
        document.write("Buzz<br>");
    else
        document.write(i + "<br>");
}
```

4.8 Övningar

- Skriv en funktion som konverterar celsius till fahrenheit och gör det möjligt för användaren att välja vilken konvertering som ska köras.
- Implementera FizzBuzz i de andra språk som vi lärt oss i kursen (görs med fördel även för Haskell och Prolog).

Kapitel 5

Händelsestyrd och webbprogrammering i JavaScript 2

5.1 Arrayer

JavaScript har arrayer/listor vilka vi kan indexera med siffror.

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];
      document.write(arr[1]);
    </script>
  </body>
</html>
```

Obs: vi behöver inte specificera storleken på arrayen och den kan innehålla element av olika typer.

Vi kan även skriva ut hela arrayen:

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];

      document.write(arr);
    </script>
  </body>
</html>
```

Obs: för itererade arrayer är utskriften kanske inte vad man förväntar sig:

```
<html>
  <body>
    <script>
      var arr = [[1, "hej", true],[2,3]];
```

```
      document.write(arr);
    </script>
  </body>
</html>
```

Det är därför första deluppgiften på uppgift 2 är att skriva ut nästlade arrayer så att man kan se nästlingen.

Värden i arrayer skrivs över på följande sätt:

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];
      arr[0] = 5;
      document.write(arr[0]);
    </script>
  </body>
</html>
```

Varning: om man indexerar utanför arrayens storlek så får man `undefined` värden mitt i:

```
<html>
  <body>
    <p id="test"></p>

    <script>
      var arr = [1, "hej", true];
      arr[4] = 2;
      document.getElementById("test").innerHTML =
          "" + arr[2] + " " + arr[3] + " " + arr[4] + " " + arr[5];
    </script>
  </body>
</html>
```

Vi använder `length` attributen för att få fram längden på en array:

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];
      document.write(arr.length);
    </script>
  </body>
</html>
```

Metoden `concat` används för att slå ihop arrayer:

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];
      document.write(arr.concat([2,3,4]));
```

```
    </script>
  </body>
</html>
```

Detta funkar även för flera arrayer:

```
<html>
  <body>
    <script>
      var arr = [1, "hej", true];
      document.write(arr.concat([2,3,4],["du",false]));
    </script>
  </body>
</html>
```

Vi kan även iterera över en array med `for`:

```
<html>
  <body>
    <script>
      var arr = [1, 2, 3];

      for (var i = 0; i < arr.length; i++) {
          document.write(arr[i] + "<br >");
      }
    </script>
  </body>
</html>
```

Det finns även `for-of` loopar som beter sig som Pythons `for`-loopar:

```
<html>
  <body>
    <script>
      var arr = [1, 2, 3];

      for (x of arr) {
          document.write(x + "<br >");
      }
    </script>
  </body>
</html>
```

5.2 Händelser/events

Det finns massor med olika händelser (events) som man kan använda i sina JS program, se exempelvis: https://www.w3schools.com/js/js_events.asp

Vi ska idag titta på hur man kan koppla ihop klickningar på olika delar av en hemsida med JS funktioner. Detta är en del av vad ni ska göra på sista uppgiften i labben.

På följande sätt kan vi koppla ihop olika funktioner till HTML element:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function fun1() {
          alert("You clicked the first square!");
      }
      function fun2() {
          alert("You clicked the second square!");
      }
      function fun3() {
          alert("You clicked the third square!");
      }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td onclick="fun1()" id="c0"></td>
        <td onclick="fun2()" id="c1"></td>
        <td onclick="fun3()" id="c2"></td>
      </tr>
    </table>
  </body>
</html>
```

Obs: Jag använder `style.css` filen från labben så att rutorna har samma design som på labben.

Vi kan även skicka en cell till vår funktion:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function fun(cell) {
          alert(cell.id);
      }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td onclick="fun(this)" id="c0"></td>
        <td onclick="fun(this)" id="c1"></td>
        <td onclick="fun(this)" id="c2"></td>
      </tr>
    </table>
  </body>
</html>
```

På så sätt kan vi ändra cellens attribut i funktionen:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function fun(cell) {
          cell.style.backgroundColor = "#b6d9ee";
      }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td onclick="fun(this)" id="c0"></td>
        <td onclick="fun(this)" id="c1"></td>
        <td onclick="fun(this)" id="c2"></td>
      </tr>
    </table>
  </body>
</html>
```

Vi kan även använda `prompt` för att fråga användaren vad som ska stå i rutan vi klickat på:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function fun(cell) {
          var x = prompt("Write something!");
          cell.innerHTML = x;
          cell.style.backgroundColor = "#b6d9ee";
      }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td onclick="fun(this)" id="c0"></td>
        <td onclick="fun(this)" id="c1"></td>
        <td onclick="fun(this)" id="c2"></td>
      </tr>
    </table>
  </body>
</html>
```

Vi kan göra en knapp oklickbar genom att ändra dess onclick attribut till `null`:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function fun(cell) {
```

```
            var x = prompt("Write something!");
            cell.innerHTML = x;
            cell.style.backgroundColor = "#b6d9ee";
            cell.onclick = null;
        }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td onclick="fun(this)" id="c0"></td>
        <td onclick="fun(this)" id="c1"></td>
        <td onclick="fun(this)" id="c2"></td>
      </tr>
    </table>
  </body>
</html>
```

Vi kan även använda en annan funktion för att koppla en cell till någon funktion. Detta är vad ni ska göra på labben i `init` funktionen.

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script>
      function init(arr) {
          for(var i = 0; i < 3; i++) {
              var cell = document.getElementById("c" + i);

              cell.innerHTML = arr[i];
              cell.onclick = klick;
          }
      }

      function klick() {
          var x = prompt("Write something!");
          var cell = this;
          cell.innerHTML = x;
          cell.style.backgroundColor = "#b6d9ee";
          cell.onclick = null;
      }
    </script>
  </head>
  <body>
    <table align="center">
      <tr>
        <td id="c0"></td>
        <td id="c1"></td>
        <td id="c2"></td>
      </tr>
```

```
    </table>
    <script>
      init([1,2,3]);
    </script>
  </body>
</html>
```

5.3 Tre-i-rad spel

Med detta kan man till exempel göra spel i webbläsare. Här kommer komplett kod för ett tre-i-rad spel vilket gås igenom på föreläsning och påminner om det ni ska göra på labben.

Först samma CSS fil som till labben vilken bör sparas i `style.css`:

```
table {
    border-spacing: 0;
    border: 4px solid #192841;
    border-collapse: collapse;
}

td {
    border: 2px dotted #192841;
    padding: 0;
    width: 60px;
    height: 60px;
    text-align: center;
    font-size: 2.0em;
}

td:hover {
    background-color:#f2e5ff;
}

#c0x2, #c0x5, #c1x2, #c1x5, #c2x5,
#c3x2,#c3x5, #c4x2, #c4x5, #c5x2, #c5x5,
#c6x2, #c6x5, #c7x2, #c7x5, #c8x2, #c8x5 {
    border-right: 3px solid #192841;
}

#c2x0, #c2x1, #c2x3, #c2x4, #c2x6, #c2x7,
#c5x0, #c5x1, #c5x3, #c5x4, #c5x6, #c5x7 {
    border-bottom: 3px solid #192841;
}

#c2x2, #c2x5, #c2x8, #c5x2, #c5x5, #c5x8 {
    border-bottom: 3px solid #192841;
    border-right: 3px solid #192841;
}
```

Sen en HTML fil för att rita upp spelet i webbläsaren:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet" type="text/css">
    <script type="text/javascript" src="tictactoe.js"></script>
  </head>
  <body>
    <table align="center">
      <tr><td id='c0x0'></td><td id='c0x1'></td><td id='c0x2'></td></tr>
      <tr><td id='c1x0'></td><td id='c1x1'></td><td id='c1x2'></td></tr>
      <tr><td id='c2x0'></td><td id='c2x1'></td><td id='c2x2'></td></tr>
    </table>
    <script>
      init();

      // Tester
      var foo = emptyGrid();

      for (x of foo) {
          for (y of x) {
              document.write("x");
          }
          document.write("<br>");
      }

      document.write(isFilled(filled) + "<br>");

      document.write(rader(filled) + "<br>");

      document.write(isWon(filled) + "<br>");

      document.write(isWon(won) + "<br>");

      document.write(getGrid() + "<br>");
    </script>
  </body>
</html>
```

Slutligen en JavaScript fil för själva spelet vilket bör sparas i `tictactoe.js`:

```
function emptyGrid() {
    return [["","",""],["","",""],["","",""]];
}

var filled = [["O","X","X"],
              ["X","X","O"],
              ["O","O","X"]];


var won = [["X","X","X"],
           ["X","O","O"],
```

```
          ["O","O","X"]];

function isFilled(grid) {
    var b = true;

    for (row of grid)
        for (x of row)
            b &&= (x == "O" || x == "X");

    return b;
}

function rader(grid) {

    // 3 rader
    var rows = grid;

    // 3 kolumner
    var cols = emptyGrid();

    for (var i = 0; i < 3; i++)
        for (var j = 0; j < 3; j++)
            cols[i][j] = grid[j][i];

    // 2 diagonaler
    var diags = [[grid[0][0],grid[1][1],grid[2][2]],
                 [grid[2][0],grid[1][1],grid[0][2]]];

    // returnera alla dessa
    return rows.concat(cols,diags);
}

function isWon(grid) {
    var rs = rader(grid);

    for (r of rs)
        if ((r[0] == r[1]) && (r[1] == r[2])
            && (r[0] == "X" || r[0] == "O"))
            return true;

    return false;
}

function init() {
    for (var i = 0; i < 3; i++)
        for (var j = 0; j < 3; j++) {
            var cell = document.getElementById("c" + i + "x" + j);
            cell.onclick = klick;
        }
```

```
}

function getGrid() {
    var out = emptyGrid();

    for (var i = 0; i < 3; i++)
        for (var j = 0; j < 3; j++) {
            var cell = document.getElementById("c" + i + "x" + j);
            out[i][j] = cell.innerHTML;
        }

    return out;
}

// true is "X" and false is "O"
var player = true;

function klick() {
    var cell = this;

    if (player) {
        cell.innerHTML = "X";
        player = !player;
    } else {
        cell.innerHTML = "O";
        player = !player;
    }

    var g = getGrid();

    if (isWon(g)) {
        if (player)
            alert("O won!");
        else
            alert("X won!");
    }

    if (not (isWon(g)) && isFilled(g)) {
        alert("No one won!");
    }
}
```

Lägger man detta i en mapp och öppnar HTML filen så får man upp ett tre-i-rad spel som kan spelas.

5.4 Sammanfattning

Detta är slutet på delen om händelsestyrd och webbprogrammering. Vi har sett hur multiparadigmspråket JavaScript kan användas för att skriva interaktiva webbsidor där vad som händer bestäms av vad användaren klickar på och skriver in. Precis som många andra scriptspråk är JavaScript dynamiskt och

svagt typat. Detta betyder att variabler kan byta typ under körningen av ett program och att värden kan byta typ om det behövs. Detta gör det lätt att slänga ihop ett script som gör något, men det är väldigt svårt att försäkra sig om att det är fritt från buggar. Det är även svårare att debugga denna typ av program jämfört med C/Java program då buggar som i C/Java hade hittats under kompilering hittas först när man kör programmet. Trots detta är JavaScript ett väldigt populärt språk för att skriva interaktiva hemsidor, men det finns varianter av JavaScript med starkare typsystem som Microsofts TypeScript eller Facebooks Flow.

Andra former av händelsestyrd programmering kan vara program som styrs av sensorer ("om temperaturen är över X grader gör Y") eller program som styrs av meddelanden (exempelvis meddelandeappar på mobiltelefoner). De flesta, om inte alla, större språk stödjer olika former av händelsestyrd programmering. Detta hänger ofta ihop med GUI programmering och andra koncept som är nödvändiga för att stödja användarinteraktion.

Man kan säga att HTML, CSS och JavaScript är grundpelarna inom webbprogrammering, men många andra språk är viktiga för moderna webbsidor. Det finns bland annat många språk som designats för att köras på en server, t.ex. PHP och ASP. Många webbsidor har även databaser vilka hanteras med hjälp av specifika databasspråk som SQL. Denna typ av språk är "domänspecifika" (*eng. domain-specific language*), dvs de är designade för en specifik tillämpning. Detta står i kontrast till de språk vi sett hittils i kursen vilka har generella tillämpningsområden (*eng. general-purpose language*). Det finns massor med domänspecifika språk, bland annat Bash (Unix shellscripting), LaTeX (avancerat typsättningsspråk), Excel (kalkylbladsspråk), Matlab/Mathematica/Maple (språk för datoralgebra), språk för att designa dataspel, osv…

Nästa paradigm vi ska titta på är *funktionell* programmering. För detta ska vi använda språket Haskell. Detta skiljer sig markant från vad vi sett än så länge i kursen och kan vara ganska svårt att komma igång med första gången man ser det. Bland annat finns det inga for/while-loopar, utan all iteration görs genom rekursion. Haskell är även statiskt typat och har ett mycket kraftfullare typsystem än de språk vi sett hittils.

Kapitel 6

Funktionell programmering i Haskell 1

6.1 Funktionell programmering

Vi är nu klara med imperativ, objektorienterad och webb/händelsestyrd programmering. Dessa är relativt lika paradigm då man kan se objektorienterad programmering som en utökning av imperativ programmering. I den här och de följande 3 föreläsningarna ska vi titta på ett paradigm som är väldigt annorlunda mot vad vi sett hittils i kursen. Detta paradigm heter *funktionell programmering* och bygger på att man programmerar genom att anropa, sätta ihop och modifiera funktioner.

Med en "funktion" menar vi funktioner som i matematiken, dvs någonting som tar element av en viss typ/mängd och returnerar element av en annan eller samma typ/mängd. Så givet ett indata returnerar alltid en funktion ett utdata. Detta skiljer sig från funktioner i till exempel C, Java och Python vilka kan göra lite vad som helst och sedan returnera `void`. En sån funktion i C, Java och Python kan t.ex. skriva ut saker eller läsa från en fil och göra olika saker beroende på vad vi läst in. En ren matematisk funktion å andra sidan kan inte göra såna saker. Man säger att en funktion i programmering är "ren" (eng. *pure*) om den inte har några sidoeffekter (så ingen input-output och dylikt) och om den alltid givet ett indata ger samma utdata (så vad funktionen gör är inte beroende på några externa omständigheter).

Detta betyder att man i funktionell programmering ofta skiljer på rena funktioner som beter sig som matematiska funktioner och de funktioner som kan ha sidoeffekter. Med sidoeffekter menar vi alltså sånt som input-output, att modifiera globala variabler, och hantera state och muterbar data.

Den beräkningsmodell som hänger ihop med funktionell programmering är logikern Alonzo Churchs λ-kalkyl från 1930-talet. Det är ett väldigt simpelt språk där allt är en funktion och man kan skriva godtyckligt komplexa program genom att slå ihop funktioner (dvs genom funktionskomposition). Så i λ-kalkylen är t.ex. även heltal och sanningsvärden representerade som funktioner. Rent beräkningsmässigt är denna modell lika kraftfull som Turingmaskiner, men moderna funktionella språk är mer uttrycksfulla så att heltal och sanningsvärden inte behöver kodas som funktioner utan som typer med olika värden (på samma sätt som i andra moderna språk vi har sett i kursen).

Det finns många funktionella språk, några av de mer kända är Lisp, Scheme, Haskell, OCaml, F#… Många språk som Python och Java har även de funktionella inslag, t.ex stödjer både Python och Java så kallade lambda-uttryck (dvs så kallade anonyma funktioner från λ-kalkylen). I den här kursen ska vi använda

oss av Haskell då det är väldigt funktionellt och lämpar sig väl för att introducera många koncept som är specifika för funktionell programmering. De koncept från funktionell programmering som vi ska titta närmare på är:

- Rekursion
- Mönstermatchning
- Algebraiska datatyper
- Typinferens
- Lathet
- Högre ordningens funktioner
- Lambda uttryck
- Polymorfism och typvariabler

6.2 Funktionell programmering i Haskell

Haskell är som sagt ett väldigt funktionellt språk. En egenhet med Haskell är att det har ett väldigt rikt typsystem som låter en specificera vilken typ av sidoeffekter en funktion har. Man kan även på ett enkelt sätt skriva väldigt generella funktioner som fungerar för många olika typer. En annan egenhet är att det är ett "lat" språk, detta är i princip unikt för Haskell och det betyder att ett uttryck eller en funktion inte anropas om man inte behöver det. På så sätt kan man enkelt representera oändliga värden, t.ex. listor med oändligt många element. I de kommande föreläsningarna ska vi lära oss om detta.

För att köra Haskellkod bör ni installera GHC (Glasgow Haskell Compiler). Detta kan man göra genom att ladda ner och installera från https://www.haskell.org/ghc/ eller genom att installera Haskell platform från https://www.haskell.org/platform/.

Haskell har även en bra hemsida med en massa information: https://www.haskell.org/

En bra och kort bok om Haskellprogrammering som ni kan läsa helt gratis på Internet är Learn You a Haskell for Great Good!. Det finns även en massa andra tutorials och material online.

Idag ska vi gå igenom grunderna i Haskell:

- Grundläggande typer och operatorer
- If-satser och guards
- Input-output

6.2.1 Hello, World!

Nedan kommer koden för Hello, World!

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Om man lägger detta i en fil som heter `HelloWorld.hs` så kan vi sedan kompilera den med hjälp av `ghc` genom att skriva följande rader i terminalen:

```
$ ghc --make HelloWorld.hs
[1 of 1] Compiling Main             ( HelloWorld.hs, HelloWorld.o )
Linking HelloWorld ...
$ ./HelloWorld
Hello, World!
```

Då vi kallat funktionen `main` kommer den anropas när vi kör det kompilerade programmet. Vi kan även ladda programmet i GHC tolken `ghci` och anropa `main` funktionen:

```
$ ghci HelloWorld.hs
GHCi, version 8.8.2: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( HelloWorld.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello, World!
```

Låt oss nu gå igenom `HelloWorld.hs` rad för rad:

- Rad 1: Här specificerar vi typen hos `main`, dvs vi skriver dess *typsignatur*. Typen kommer alltid efter `::` och i detta fall är den `IO ()`. Detta betyder att det är en funktion som kan göra input-output (`IO`) och vilken inte returnerar något (detta specificeras genom `()`). Man bör alltid skriva funktioners typ då det underlättar för den som läser koden och med felsökning, men detta är inte ett måste och GHC kan i princip alltid räkna ut vad typen är om man utelämnar den.
- Rad 2: Här kommer själva funktionskroppen/definitionen för `main`. I detta fall anropas funktionen `putStrLn` ("put-string-line") vilken skriver ut dess första argument och lägger till en nyrad (`'\n'`) i slutet. Observera att vi inte behöver skriva någon parentes runt argumenten till `putStrLn`. Funktionsanrop skrivs alltså med mellanslag, så `f x` istället för `f(x)`.

6.2.2 Haskell tolken `ghci`

Då vi inte ska skriva några större program i Haskell i den här kursen räcker det med att använda `ghci`. Om vi startar `ghci` från terminalen så kan vi köra en massa olika kommandon och använda den som en (avancerad) miniräknare:

```
$ ghci
GHCi, version 8.8.2: https://www.haskell.org/ghc/  :? for help
Prelude> 2 + 15
17
Prelude> 49 * 100
4900
Prelude> 1892 - 1472
420
Prelude> 5 / 2
2.5
Prelude> True && False
False
Prelude> True && True
True
Prelude> False || True
True
Prelude> not False
True
Prelude> not (True && True)
False
Prelude> 5 == 5
True
Prelude> 1 == 0
```

```
False
Prelude> 2 > 1
True
Prelude> 1 < 2
False
Prelude> 5 /= 5
False
Prelude> 5 /= 4
True
Prelude> "hello" == "hello"
True
```

Att det står "Prelude" i ghci prompten här betyder att vi bara har laddat Haskells Prelude bibliotek. Obs: beroende på er installation kan det hända det står något annat (t.ex. `ghci`) i prompten samt att versionsnumret är annorlunda. Det spelar ingen roll för kursen utan allt vi gör ska funka på de senaste versionerna av GHC.

Ni kan läsa om vad det har för olika typer och funktioner i https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html. Om ni har installerat en annan version av `ghci` än jag har så kan det stå andra saker i prompten, det spelar troligtvis ingen roll när ni programmerar men om ni får problem kan ni be handledaren om hjälp.

Vad händer om vi försöker jämföra `5` och `True`?

```
Prelude> 5 == True
<interactive>:1:1: error:
    • No instance for (Num Bool) arising from the literal ‘5’
    • In the first argument of ‘(==)’, namely ‘5’
      In the expression: 5 == True
      In an equation for ‘it’: it = 5 == True
```

Vi får ett kryptiskt felmeddelande som i princip säger att `True` inte är ett numeriskt värde och att vi inte kan jämföra det med 5. Till skillnad från t.ex. Python och JavaScript är Haskell statiskt typcheckat och allt man skriver måste ha korrekt typ. Därför får vi även ett felmeddelande när vi skriver:

```
Prelude> 2 * "hej"
<interactive>:2:1: error:
    • No instance for (Num [Char]) arising from a use of ‘*’
    • In the expression: 2 * "hej"
      In an equation for ‘it’: it = 2 * "hej"
```

Till skillnad från Python kan vi alltså inte multiplicera heltal och strängar.

6.3 Grundläggande typer

Haskell har ett antal grundläggande typer:

- `Bool`: sanningsvärden (kan vara True eller False)
- `Int`: heltal (av begränsad storlek)
- `Integer`: heltal (obegränsad storlek)
- `Double` och `Float`: flyttal av begränsad storlek
- `Char`: bokstäver (skrivs inom enkelfnuttar)

- `String`: strängar (skriv inom dubbelfnuttar, representeras som listor av `Char`)
- `()`: unit (ungefär som void, har bara ett värde som även det heter `()`)

Obs: typer i Haskell skrivs alltid med stor första bokstav.

Vi kan även skapa mer komplicerade typer med hjälp av:

- `(A,B)`: tupler (ex: `(Int,Bool)` är typen för par av heltal och sanningsvärden)
- `[A]`: listor (ex: `[Bool]` är typen av listor med sanningsvärden)
- `A -> B`: funktioner (ex: `Int -> Bool` är typen för funktioner från heltal till sanningsvärden):

Vi kan använda `ghci` för att få fram typen på värden. Detta kallas för *typinferens*.

```
Prelude> :t True
True :: Bool
Prelude> :t "hej"
"hej" :: [Char]
Prelude> :t 'h'
'h' :: Char
Prelude> :t [("hej",True),("du",False)]
[("hej",True),("du",False)] :: [([Char], Bool)]
Prelude> :t 0
0 :: Num p => p
Prelude> :t 2.2
2.2 :: Fractional p => p
```

De första 4 exemplen ovan är inte så konstiga, bortsett från att `String` skrivs ut som `[Char]`. Anledningen till detta är att strängar representeras som listor av bokstäver i Haskell.

De två sista exemplen är dock konstigare. Typen

```
0 :: Num p => p
```

säger att `0` är ett element av någon typ `p` som är en "instans" av typklassen `Num` (typklassinstanser skrivs alltid till vänster om `=>` i början av typsignaturen). Detta betyder i princip att `0` har en typ `p` som är numerisk på något sätt. På samma sätt säger `ghci` åt oss att `2.2` har någon "Fractional" typ, dvs en typ av fraktioner av något slag. Anledningen till detta är att `ghci` alltid försöker lista ut den mest generella typen som är möjlig för något vi skriver så att koden kan fungera på så många ställen som möjligt. Då det finns många olika typer som har ett `0` värde (t.ex. `Int`, `Integer`, `Double`, ...) så drar `ghci` inga förhastade slutsatser om vad detta kan vara för typ av nolla. Vi kan säga åt `ghci` att nollan vi skrivit ska tolkas som en `Int` genom:

```
Prelude> :t (0 :: Int)
(0 :: Int) :: Int
```

Vi kan även fråga `ghci` om typen för olika funktioner:

```
Prelude> :t not
not :: Bool -> Bool
Prelude> :t (&&)
(&&) :: Bool -> Bool -> Bool
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Här ser vi att `not` är en funktion som tar en `Bool` och ger oss en `Bool`. `&&` å andra sidan tar in två `Bool` och ger oss en `Bool` tillbaka. Observera att för att be om typen för en operator som `&&` måste vi skriva den inom paranteser. Slutligen ser vi att `+` tar in två element av någon numerisk typ `a` och ger oss ett `a` tillbaka. Detta verkar vettigt då vi kan använda `+` för många olika numeriska typer.

6.4 Grundläggande operatorer

Haskell stödjer många av de vanliga räknesätten:

- `(+)`: addition
- `(*)`: multiplikation
- `(-)`: subtraktion
- `(/)`: division
- `mod`: modulo
- `div`: heltalsdivision

Logiska funktioner på `Bool`:

- `(&&)`: logiskt *och*
- `(||)`: logiskt *eller*
- `not`: logisk *negation*

Jämförelseoperatorer:

- `(==)`: likhet
- `(/=)`: olikhet
- `(>=)`: större än eller lika
- `(<=)`: mindre än eller lika
- `(>)`: större än
- `(<)`: mindre än

De första två av jämförelseoperatorerna hänger ihop med typklassen `Eq` och de 4 sista med `Ord`. Så om ni ser felmeddelanden som handlar om `Eq` eller `Ord` så har det med dessa operatorer att göra.

Två andra väldigt användbara funktioner är `show` och `read`. Den första av dessa konverterar ett element till en sträng (så `show 42` returnerar strängen `"42"`). Funktionen `read` går åt andra hållet och konverterar en sträng till den typ man anger om möjligt (så `read "42" :: Int` kommer bli en `Int`). Dessa funktioner hänger ihop med typklasserna `Show` och `Read`. Vi kommer prata mer om typklasser senare, men redan nu ser vi att de låter oss göra liknande saker som `__str__` i Python och `toString()` i Java (dvs lägga till samma funktionalitet till olika typer som vi definierar).

6.4.1 Att skriva Haskellprogram

Nu ska vi äntligen skriva några enklare Haskellprogram. I princip allt i Haskell är funktioner eller konstanter (dvs funktioner utan argument). Vi kan definiera en konstant för `pi` och sedan en funktion som använder `pi` för att beräkna omkretsen av en cirkel.

```haskell
pi :: Double
pi = 3.14

circumference :: Double -> Double
circumference r = 2 * pi * r
```

Om vi laddar detta program i `ghci` kan vi göra följande:

```
> circumference 4.0
25.132742
```

Funktioner kan även ta flera argument:

```
addThreeNumbers :: Int -> Int -> Int -> Int
addThreeNumbers x y z = x + y + z
```

och vi kan använda funktionen på följande sätt:

```
> addThreeNumbers 1 2 3
6
```

Observera att vi applicerar funktioner genom att skriva mellanslag. Detta hänger ihop med ett koncept som heter *partiell applicering*. Om vi bara ger `addThreeNumbers` ett argument får vi en ny funktion som förväntar sig två fler argument och om vi ger den två argument får vi en funktion som förväntar sig ett till argument:

```
> :t addThreeNumbers 1
addThreeNumbers 1 :: Int -> Int -> Int
> :t addThreeNumbers 1 2
addThreeNumbers 1 2 :: Int -> Int
> :t addThreeNumbers 1 2 3
addThreeNumbers 1 2 3 :: Int
```

På så sätt kan vi skapa nya funktioner från funktioner som tar flera argument och vi ser alltså att `addThreeNumbers` har följande typ:

```
addThreeNumbers :: Int -> (Int -> (Int -> Int))
addThreeNumbers x y z = x + y + z
```

Det vill säga, det är en funktion som tar in en `Int` och ger oss en ny funktion som tar in en `Int` och ger oss en ny funktion som tar in en `Int` och ger oss en `Int`. Puh! Att funktioner som tar flera argument är som funktioner som returnerar funktioner hänger ihop med konceptet högre ordningens funktioner vilket vi ska prata om senare i kursen.

Vi kan dock inte skriva ut en partiellt applicerad funktion:

```
> addThreeNumbers 1
<interactive>:17:1: error:
    • No instance for (Show (Int -> Int -> Int))
        arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```

Det felmeddelandet säger är att vi inte har någon instans av `Show` klassen för funktioner `Int -> Int -> Int`, så Haskell kan inte anropa `show` funktionen på `addThreeNumbers 1`.

6.5 If-satser och guards

Precis som de andra språken vi sett stödjer Haskell if-satser. Syntaxen är lite annorlunda, men de fungerar på samma sätt:

```haskell
isEven :: Int -> Bool
isEven x = if mod x 2 == 0 then True else False

-- Kortare:
isEven' :: Int -> Bool
isEven' x = mod x 2 == 0
```

Vi kan även definiera vår egen `max` funktion:

```haskell
myMax :: Integer -> Integer -> Integer
myMax x y = if x > y
               then x
               else y
```

Haskell har inga `else-if` satser, istället kan man använda "guards". Om vi skriver `myMax` på följande sätt

```haskell
myMax :: Integer -> Integer -> Integer
myMax x y | x > y = x
          | otherwise = y
```

så kommer Haskell först testa om `x > y`, är det sant returneras `x`, annars returneras `y`. Vi kan ha flera guards på raken vilket ger oss samma funktionalitet som `else-if`:

```haskell
compareToString :: Integer -> Integer -> String
compareToString x y | x > y = "GT"
                    | x == y = "EQ"
                    | otherwise = "LT"
```

Detta är användbart så att man inte behöver skriva nästlade `if-then-else` satser vilket snabbt blir oläsligt.

6.6 Input-output

För att skriva program med IO i Haskell måste vi ge vår funktion typen `IO ()`:

```haskell
main :: IO ()
main = do
    putStrLn "Enter two numbers:"
    x <- getLine
    y <- getLine
    putStrLn ("The sum: " ++ show (read x + read y))
```

När vi skriver IO kod så kan vi använda do-notation vilket gör koden i do-blocken imperativ. Raderna exekveras alltså från toppen till botten inom do-blocket. Då Haskell är indenteringskänsligt precis som Python är det viktigt att koden är indenterad på rätt sätt. Detta kan vara svårt att lära sig i början och det kan vara svårt att hitta eventuella fel. Vad är fel med nedanstående kod?

```haskell
main :: IO ()
main = do
    putStrLn "Enter two numbers:"
    x <- getLine
    y <- getLine
   putStrLn ("The sum: " ++ show (read x + read y))
```

Det är bra programmeringsstil att dela upp sitt program i rena funktioner som gör beräkningar och bara returnerar något och de funktioner som har sidoeffekter:

```
isEven :: Int -> Bool
isEven x = mod x 2 == 0

main :: IO ()
main = do
    putStrLn "Enter a number:"
    x <- getLine
    if isEven (read x)
       then putStrLn "even"
       else putStrLn "odd"
```

På detta sätt är det lättare att felsöka och testa sin kod. En ren funktion har ofta en väldigt tydlig specifikation, medans funktioner med sidoeffekter ofta kan göra en massa saker som är svåra att förutse. På detta sätt kan man lättare skriva robust och modulär kod. Detta är inte unikt till funktionell programmering utan man kan så klart dela upp sina imperativa eller objektorienterade program i rena funktioner och funktioner med sidoeffekter. Det unika med Haskell är dock att typsystemet tvingar programmeraren att programmera på detta sätt medans i C, Java, Python och många andra språk kan man göra lite som man vill.

Kapitel 7

Funktionell programmering i Haskell 2

7.1 Mönstermatchning

Vi kan även skriva Haskellfunktioner genom falluppdelning, så om argumentet har ett visst värde gör programmet en speciell sak. Nedan ser vi hur vi kan skriva `not` genom att mönstermatchning.

```
myNot :: Bool -> Bool
myNot True = False
myNot False = True
```

Vi kan även mönstermatcha på flera argument:

```
myAnd :: Bool -> Bool -> Bool
myAnd True True = True
myAnd True False = False
myAnd False True = False
myAnd False False = False
```

Då alla tre sista fallen ovan har samma högersida kan vi använda ett "wildcard pattern" genom att skriva _ för argumentet. Detta betyder att fallet matchar oberoende vad värdet på argumentet är.

```
myAnd :: Bool -> Bool -> Bool
myAnd True True = True
myAnd _ _ = False
```

Vi kan även mönstermatcha på heltal:

```
safeDiv :: Integer -> Integer -> Integer
safeDiv _ 0 = 0
safeDiv x y = div x y
```

Man kan även skriva funktioner som mönstermatchar på tupler på följande sätt:

```
myAndPair :: (Bool,Bool) -> Bool
myAndPair (True,True) = True
```

```
myAndPair _ = False
```

Vi kan även skriva våra egna projektionsfunktioner från tupler:

```
fstInt :: (Int,Int) -> Int
fstInt (x,_) = x

sndInt :: (Int,Int) -> Int
sndInt (_,y) = y
```

På liknande sätt kan vi mönstermatcha på i princip alla typer i Haskell. Detta är väldigt användbart och leder till lättläst kod. Vi kommer prata mer om det när vi kommer in på algebraiska datatyper.

7.2 Rekursion

I Haskell finns det inga loopar som i imperativa språk. Istället måste man använda rekursion om man vill göra något flera gånger. Detta betyder att man skriver funktioner som anropar sig själva. Det finns många exempel på detta i matematik, bland annat inom talsekvenser. För den som är van vid imperativ programmering kan det vara svårt att se hur man kan lösa ett problem genom rekursion och det kan ta ett tag att vänja sig vid att programmera på detta sätt. Många problem har dock väldigt eleganta lösningar genom rekursion.

7.2.1 Fakultet

Standardexemplet på en rekursiv funktion är fakultet:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Detta kan beskrivas rekursivt, dvs i termer av sig själv, genom:

$$\begin{aligned} 5! &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \\ &= 5 * 4 * 3 * 2 * 1 * 0! \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 120 \end{aligned}$$

Detta är en rekursiv beräkning med *basfall* $0! = 1$. En matematiker skriver detta som:

$$n! = \begin{cases} 1 & \text{, om n är 0} \\ n * (n-1)! & \text{, annars} \end{cases}$$

I Haskell skriver vi

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Fallet när `n` är `0` kallas *basfall*. Det är här som rekursionen stannar och börjar nystas upp. Det andra fallet kallas *rekursivt steg* och går ut på att göra ett "enklare" anrop, där problemet i någon mening har gjorts mindre.

Obs: vad händer om vi anropar `factorial` med ett negativt tal?

Tester:

```
> factorial 0
1
> factorial 1
1
> factorial 2
2
> factorial 3
6
> factorial 4
24
> factorial 5
120
```

7.2.2 Fibonacci numren

Varje kurs som tar upp rekursion måste ha med Fibonacci-talen som exempel.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Det n:te Fibonacci-talet, $F(n)$, är summan av de två tidigare Fibonacci-talen. Därför kan vi skriva

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

Det är ju tydligt en rekursiv funktion och vi kan skriva den direkt i Haskell med hjälp av mönstermatchning och rekursion:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Som ni ser påminner Haskellkod ofta väldigt mycket om matematiska definitioner som den ovan. Detta har gjort Haskell ett populärt språk bland programmerare som gillar matematik.

Tester:

```
> fib 0
0
> fib 1
1
> fib 2
1
> fib 3
2
```

```
> fib 4
3
> fib 5
5
```

7.2.3 Största gemensamma delare

Euklides algoritm används för att beräkna största gemensamma delare för två tal `a` och `b`. Den har en enkel rekursiv beskrivning:

- Antag att `a >= b`,
- låt `r` vara resten av heltalsdivision av `a` och `b` (dvs `r = mod(a,b)`),
- om `r == 0`, då är `SGD(a,b) = b`,
- annars använder vi att `SGD(a,b) = SGD(b,r)`.

Vi kan översätta detta direkt till Haskell:

```
sgd :: Integer -> Integer -> Integer
sgd a b = if b > a
             then error "sgd"
             else if mod a b == 0
                     then b
                     else sgd b (mod a b)
```

Tester:

```
> sgd 25 15
5
> sgd 18 7
1
> sgd 18 27
*** Exception: sgd
CallStack (from HasCallStack):
  error, called at temp.hs:13:19 in main:Main
> sgd 27 18
9
```

Vi kan skriva `sgd` elegantare med hjälp av guards och en lokal definition genom `let-in`:

```
sgd :: Integer -> Integer -> Integer
sgd a b | b > a = error "sgd"
        | otherwise = let r = mod a b
                      in if r == 0
                            then b
                            else sgd b r
```

Med `let-in` kan vi alltså skapa lokala definitioner i funktioner:

```
foo :: Int
foo = let x = 3
          y = 39
      in x + y
```

Detta gör ofta kod lättare att läsa då man inte behöver repetera samma kodsnutt flera gånger.

7.2.4 Loopar genom rekursion

Man kan simulera `for` och `while` loopar med hjälp av rekursion. Låt oss säga att vi vill översätta följande for-loop från C som summerar alla tal upp till `num`:

```
int sum = 0;

for (int count = 1; count <= num; ++count) {
  sum += count;
}
```

då kan vi skriva:

```
mySumRec :: Integer -> Integer -> Integer -> Integer
mySumRec count num sum =
   if count <= num
      then mySumRec (count + 1) num (sum + count)
      else sum

mySum :: Integer -> Integer
mySum num = mySumRec 1 num 0
```

Tester:

```
> mySum 0
0
> mySum 1
1
> mySum 2
3
> mySum 3
6
> mySum 4
10
> mySum 5
15
```

Ett till exempel där vi sparar summan av tal som vi läst in från användaren i ett argument till funktionen `loop` och avslutar loopen när användaren skriver in en nolla (detta exempel skrevs med en `do-while` loop i C):

```
main :: IO ()
main = do
  putStrLn "Write a number:"
  x <- getLine
  loop (read x) 0

loop :: Integer -> Integer -> IO ()
loop 0 sum = putStrLn ("The sum of the numbers is " ++ show sum)
loop n sum = do
  putStrLn "Write a number:"
  x <- getLine
  loop (read x) (n + sum)
```

Testkörning:

```
> main
Write a number:
1
Write a number:
2
Write a number:
3
Write a number:
0
The sum of the numbers is 6
```

Man kan även använda `let` när man skriver `IO` funktioner. Detta används för att binda resultatet av rena beräkningar, dvs från anrop till funktioner som inte har typ `IO`. Testa följande:

```
main :: IO ()
main = do
  putStrLn "Write a number:"
  x <- getLine
  let s = mySum (read x)
  putStrLn (show s)
```

Testa att byta `let` mot `<-` och vice versa.

7.3 Listor

Typen för listor skrivs `[a]` för en godtycklig typ `a` (så `[Bool]` är listor med sanningsvärden). En lista är antingen tom (`[]`) eller ett element `x` och resten av listan `xs` (`x : xs`).

```
> []
[]
> 1 : 2 : []
[1,2]
> 1 : [2,3,4]
[1,2,3,4]
```

Listor kan ha element av godtyckliga typer (så länge alla element har samma typ):

```
> :t []
[] :: [a]
> :t (:)
(:) :: a -> [a] -> [a]
> :t (:) 1
(:) 1 :: Num a => [a] -> [a]
> :t (:) True
(:) True :: [Bool] -> [Bool]
> [1,2,True]
<interactive>:52:2: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the expression: 1
```

```
    In the expression: [1, 2, True]
    In an equation for ‘it’: it = [1, 2, True]
```

Då listor kan ha alla sorters typer i Haskell är de *parametriskt polymorfa* (eng. *parametric polymorphism*). Mer om detta senare i kursen.

Listor kan även ha listor som element (som i sig kan ha listor som element):

```
> [[1,2,3],[4],[]]
[[1,2,3],[4],[]]
> [[[]],[[1,2],[4,5]]]
[[[]],[[1,2],[4,5]]]
```

Ett smidigt sätt att skapa listor med tal är att använda `..`:

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
> [1,3..10]
[1,3,5,7,9]
> [1,5..10]
[1,5,9]
```

För att hämta ut ett specifikt index ur en lista använder man `!!` och för att slå ihop två listor använder man `++`.

```
> [1,2] ++ [3,4]
[1,2,3,4]
> [1,2] ++ []
[1,2]
> [1,2] !! 0
1
> [1,2] !! 1
2
> [1,2] !! 2
*** Exception: Prelude.!!: index too large
```

7.3.1 Grundläggande funktioner

För att skriva funktioner på listor kan vi använda mönstermatchning:

```
headIntList :: [Int] -> Int
headIntList [] = error "Can't take the head of an empty list"
headIntList (x:xs) = x

myNull :: [Int] -> Bool
myNull [] = True
myNull _ = False
```

Tester:

```
> headIntList [1,2,3]
1
> headIntList [17,1,2,3]
17
```

```
> headIntList []
*** Exception: Can't take the head of an empty list
CallStack (from HasCallStack):
  error, called at temp.hs:57:18 in main:Main
> myNull []
True
> myNull [1]
False
```

Vi kan iterera över listor med hjälp av rekursion:

```
myLength :: [Double] -> Integer
myLength [] = 0
myLength (x:xs) = 1 + myLength xs

myTake :: Integer -> [Double] -> [Double]
myTake 0 xs = []
myTake _ [] = []
myTake n (x:xs) = x : myTake (n-1) xs

myDrop :: Integer -> [Double] -> [Double]
myDrop 0 xs = xs
myDrop _ [] = []
myDrop n (x:xs) = myDrop (n-1) xs
```

Tester:

```
> myLength []
0
> myLength [4,2,3]
3
> myTake 2 [1..]
[1.0,2.0]
> myDrop 2 [1..10]
[3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0]
```

Den generalla versionen av dessa funktioner finns i Data.List biblioteket (utan my och med liten första bokstav): https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html

7.3.2 Sortering

Vi kan lätt sortera en lista med värden på följande sätt:

```
getSmallerEq :: Int -> [Int] -> [Int]
getSmallerEq _ [] = []
getSmallerEq n (x:xs) | x <= n = x : getSmallerEq n xs
                      | otherwise = getSmallerEq n xs

getGreater :: Int -> [Int] -> [Int]
getGreater _ [] = []
getGreater n (x:xs) | x > n = x : getGreater n xs
                    | otherwise = getGreater n xs
```

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) =
  let smallerSorted = sort (getSmallerEq x xs)
      biggerSorted = sort (getGreater x xs)
  in smallerSorted ++ [x] ++ biggerSorted
```

Test:

```
> sort [2,3,1,4]
[1,2,3,4]
```

7.3.3 Pascals triangel

Ett klassiskt exempel på en rekursiv definition är Pascals triangel:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

osv…

Vi kan implementera en funktion som beräknar den n:e raden genom:

```
addLists :: [Int] -> [Int] -> [Int]
addLists [] _ = []
addLists _ [] = []
addLists (x:xs) (y:ys) = (x + y) : addLists xs ys

pascal :: Int -> [Int]
pascal 0 = [1]
pascal n = addLists ([0] ++ pascal (n-1)) (pascal (n-1) ++ [0])
```

Detta resulterar i:

```
> pascal 0
[1]
> pascal 1
[1,1]
> pascal 2
[1,2,1]
> pascal 3
[1,3,3,1]
> pascal 4
[1,4,6,4,1]
> pascal 5
[1,5,10,10,5,1]
```

7.3.4 Rekursion på listor med tupler

Vi kan slå ihop två listor till en lista med par:

```
myZip :: [Int] -> [Int] -> [(Int,Int)]
myZip [] _ = []
myZip _ [] = []
myZip (x:xs) (y:ys) = (x,y) : myZip xs ys
```

Vi kan även göra detta åt andra hållet:

```
myUnzip :: [(Int,Int)] -> ([Int],[Int])
myUnzip [] = ([],[])
myUnzip ((x,y):xys) =
  let (xs,ys) = myUnzip xys
  in (x:xs,y:ys)
```

Tester:

```
> myZip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
> myZip [1,2,3] [4,5,6,8]
[(1,4),(2,5),(3,6)]
Main> myUnzip [(1,3),(4,1),(4,2)]
([1,4,4],[3,1,2])
```

7.4 Lathet och oändliga listor

Haskell är ett lat språk. Detta betyder att kod bara körs om resultatet behövs.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)

main :: IO ()
main = do
  let xy = (factorial 100000000,42)
  putStrLn (show (snd xy))
```

Vad händer om vi kör `putStrLn (show (fst xy))` istället?

7.4.1 Oändliga listor

Med hjälp av lathet kan vi skapa oändliga listor. Vi kan så klart aldrig skriva ut en hel oändlig lista, men vi kan hämta ut hur många element som helst. Genom att skriva `[x..]` får vi en oändlig lista med tal från `x`.

```
> take 4 [1..]
[1,2,3,4]
> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

7.4.2 Primtal

Vi kan skapa en lista med alla primtal genom:

```
isPrimeRec :: Integer -> Integer -> Bool
isPrimeRec x y | x == y = True
               | mod x y == 0 = False
               | otherwise = isPrimeRec x (y+1)

isPrime :: Integer -> Bool
isPrime x = isPrimeRec x 2

primesRec :: [Integer] -> [Integer]
primesRec [] = []
primesRec (x:xs) | isPrime x = x : primesRec xs
                 | otherwise = primesRec xs

primes :: [Integer]
primes = primesRec [2..]
```

Och vi kan hämta ut så många primtal vi vill ur listan:

```
> take 4 primes
[2,3,5,7]
> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Det är nu relativt lätt att skriva en funktion som primtalsfaktoriserar:

```
primeFactors :: Integer -> [Integer]
primeFactors x = primeFactorsRec x primes
  where
    primeFactorsRec 1 _ = []
    primeFactorsRec x (p:ps)
      | p > x = []
      | mod x p == 0 = p : primeFactorsRec (div x p) (p:ps)
      | otherwise = primeFactorsRec x ps
```

Den här koden är inte alls optimerad, men den fungerar ändå för relativt stora tal:

```
> primeFactors 125
[5,5,5]
> primeFactors 1250
[2,5,5,5,5]
> primeFactors 12501
[3,3,3,463]
> primeFactors 12201
[3,7,7,83]
> primeFactors 72201
[3,41,587]
```

7.5 Övningar

- Skriv `myOr :: Bool -> Bool -> Bool` som implementerar logiskt *eller*.
- Skriv `myXor :: Bool -> Bool -> Bool` som implementerar logiskt *exklusivt eller* (dvs den blir bara sann om max ett argument är sant).
- Skriv `fst3Int :: (Int,Int,Int) -> Int` som tar in en trippel och returnerar första elementet.
- Skriv `addPair :: (Int,Int) -> Int` som tar in ett par av tal och returnerar deras summa.

Kapitel 8

Funktionell programmering i Haskell 3

8.1 Algebraiska datatyper

Precis som i Java och Python kan vi lägga till våra egna datatyper i Haskell. Dessa görs inte genom klasser som i objektorienterad programmering, utan genom så kallade *algebraiska* datatyper.

8.1.1 Sanningsvärden

Vi kan definiera våra egna sanningsvärden genom:

```
data Bool = True | False

not :: Bool -> Bool
not True = False
not False = True
```

Då `Bool` redan finns i Haskell i `Prelude` biblioteket måste vi börja vår fil med följande rad för att exemplet ska fungera

```
import Prelude hiding (Bool(..))
```

8.1.2 Färger

Vi kan definiera en typ för några färger genom:

```
data Color = Red | Green | Blue

colorToString :: Color -> String
colorToString Red   = "red"
colorToString Green = "green"
colorToString Blue  = "blue"

sameColor :: Color -> Color -> Bool
```

```
sameColor Red Red     = True
sameColor Green Green = True
sameColor Blue Blue   = True
sameColor _ _         = False
```

8.1.3 Heltal

Vi kan definiera våra egna heltalsvärden genom:

```
data Nat = Zero | Succ Nat
  deriving (Show)
```

Detta är ett exempel på en *rekursiv* datatyp. Ett heltal är antingen `Zero` eller `Succ` applicerat på ett annat heltal. Så exempelvis 3 skrivs genom:

```
three :: Nat
three = Succ (Succ (Succ Zero))
```

Vi använder `deriving (Show)` för att generera en `Show` instans för `Nat`, dvs vi får en standarddefinition av `show` som funkar på `Nat`. Detta gör att vi kan skriva ut `Nat` värden som strängar.

När vi definierar datatyper som ovan får vi konstruktorer med följande typ:

```
> :t Zero
Zero :: Nat
> :t Succ
Succ :: Nat -> Nat
```

Vi kan skriva enkla funktioner på heltal genom mönstermatchning och rekursion:

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)

natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Succ n) = 1 + natToInt n

intToNat :: Int -> Nat
intToNat n | n < 0 = error "intToNat"
intToNat 0 = Zero
intToNat n = Succ (intToNat (pred n))
```

Tester:

```
> add three three
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
> natToInt three
3
> intToNat 2
Succ (Succ Zero)
```

Det kan vara bra att "handexekvera funktioner för att förstå hur de beter sig:

```
add three three =
```

```
add (Succ (Succ (Succ Zero))) three =
Succ (add (Succ (Succ Zero)) three) =
Succ (Succ (add (Succ Zero) three)) =
Succ (Succ (Succ (add Zero three))) =
Succ (Succ (Succ three)) =
Succ (Succ (Succ (Succ (Succ Zero)))))


natToInt three =
natToInt (Succ (Succ (Succ Zero))) =
1 + natToInt (Succ (Succ Zero)) =
1 + 1 + natToInt (Succ Zero) =
1 + 1 + 1 + natToInt Zero =
1 + 1 + 1 + 0 =
3
```

8.1.4 Listor med tal

Vi kan definiera våra egna listor med tal genom:

```
data IntList = Nil | Cons Int IntList
```

Funktioner på dessa skrivs också med mönstermatchning:

```
head :: IntList -> Int
head Nil = error "head"
head (Cons x _) = x

tail :: IntList -> IntList
tail Nil = error "tail"
tail (Cons _ xs) = xs

length :: IntList -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs

(++) :: IntList -> IntList -> IntList
Nil ++ ys = ys
(Cons x xs) ++ ys = Cons x (xs ++ ys)
```

Obs: för att detta exempel ska fungera måste vi gömma `(++)` och `length` från `Prelude`:

```
import Prelude hiding ((++),length)
```

8.1.5 Binära träd med tal

Vi kan definiera binära träd med tal genom:

```
data BinTree = Leaf Int | Branch Int BinTree BinTree

binTreeToString :: BinTree -> String
binTreeToString (Leaf x) = "Leaf " ++ show x
```

```
binTreeToString (Branch x t1 t2) =
  "Branch " ++ show x ++
    " (" ++ binTreeToString t1 ++ ") (" ++ binTreeToString t2 ++ ")"

member :: Int -> BinTree -> Bool
member x (Leaf y) = x == y
member x (Branch y t1 t2) = x == y || member x t1 || member x t2

linearize :: BinTree -> [Int]
linearize (Leaf x) = [x]
linearize (Branch x t1 t2) = linearize t1 ++ [x] ++ linearize t2

tree1 :: BinTree
tree1 = Leaf 42

tree2 :: BinTree
tree2 = Branch 1 (Leaf 2) (Branch 7 (Leaf (-1)) (Leaf 0))

mytree :: BinTree
mytree = Branch 1 (Leaf 5) (Branch (-2) (Leaf 9) (Leaf 3))
```

Vi kan rita upp `mytree` som:

```
      1
     / \
    5  -2
       / \
      9   3
```

Hur ser `tree1` och `tree2` ut om man ritar dem?

Tester:

```
> binTreeToString tree1
"Leaf 42"
> binTreeToString tree2
"Branch 1 (Leaf 2) (Branch 7 (Leaf -1) (Leaf 0))"
> member 2 tree2
True
> member 8 tree2
False
> linearize tree1
[42]
> linearize tree2
[2,1,-1,7,0]
```

Jämför längden på den här koden med motsvarande kod i Java. Vilken är lättast att läsa? En styrka med Haskell är att man ofta kan lösa problem med väldigt lite kod.

8.2 Polymorfism

Ett värde är polymorft om det finns fler än en typ som det kan ha. Exempelvis såg vi senast att listtypen `[a]` fungerar för element av vilken typ `a` som helst. Så `[]` kan vara en tom lista av tal, en tom lista av sanningsvärden, en tom lista av listor, osv. Många språk tillåter användaren att specificera polymorfa typer (t.ex Java har "Generics" som låter en göra detta). I Haskell är detta en central del och Haskellprogrammerare strävar ofta efter att skriva så generella typer och program som möjligt.

8.2.1 Polymorfa listor

Vi kan skriva om listexemplet på följande sätt för att få våra egna listor med element av godtycklig typ `a` och inte bara `Int`:

```
data List a = Nil | Cons a (List a)
```

Argumentet `a` är en typvariabel. Vi kan byta ut den mot vilket typ vi vill. Så t.ex `List Int`, `List Bool`, `List (List (List Double))`...

```
> :t Nil
Nil :: List a
> :t Cons
Cons :: Int -> List a -> List a
```

Vi skriver funkioner på `List` på samma sätt som på `IntList`:

```
head :: List a -> Int
head Nil = error "head"
head (Cons x _) = x

tail :: List a -> List a
tail Nil = error "tail"
tail (Cons _ xs) = xs

length :: List a -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs

(++) :: List a -> List a -> List a
Nil ++ ys = ys
(Cons x xs) ++ ys = Cons x (xs ++ ys)
```

Obs: exakt samma kod som för `IntList` fungerar för `List`, trots att `List` är mycket mer generell!

8.2.2 Polymorfa träd

Vi kan även göra träden mer generella genom:

```
data BinTree a = Leaf a | Branch a (BinTree a) (BinTree a)
```

Exakt samma kod som ovan fungerar nästan, men vi måste anta att `a` är en instans av `Show` i `binTreeToString` och att `a` är en instans av `Eq` i `member`:

```
binTreeToString :: Show a => BinTree a -> String
binTreeToString (Leaf x) = "Leaf " ++ show x
```

```
binTreeToString (Branch x t1 t2) =
  "Branch " ++ show x ++
    " (" ++ binTreeToString t1 ++ ") (" ++ binTreeToString t2 ++ ")"

member :: Eq a => a -> BinTree a -> Bool
member x (Leaf y) = x == y
member x (Branch y t1 t2) = x == y || member x t1 || member x t2

linearize :: BinTree a -> [a]
linearize (Leaf x) = [x]
linearize (Branch x t1 t2) = linearize t1 ++ [x] ++ linearize t2

tree1 :: BinTree Int
tree1 = Leaf 42

tree2 :: BinTree Int
tree2 = Branch 1 (Leaf 2) (Branch 7 (Leaf (-1)) (Leaf 0))

tree3 :: BinTree Double
tree3 = Branch 1.12 (Leaf 24.11) (Branch 123.7 (Leaf (-1.0001)) (Leaf 0.1231))
```

Observera att vi i `binTreeToString` måste anta att `a` har en `Show` instans så att vi kan använda `show`. Samma gäller för `member` där `a` måste ha en `Eq` instans så att vi kan jämföra dess element. Jämför detta med hur vi var tvungna att implementera abstrakta klasser/metoder i Java om vi ville kunna anta att någon metod fanns tillfänglig i en klass.

Tester:

```
> binTreeToString tree3
"Branch 1.12 (Leaf 24.11) (Branch 123.7 (Leaf -1.0001) (Leaf 0.1231))"
> linearize tree3
[24.11,1.12,-1.0001,123.7,0.1231]
> linearize tree2
[2,1,-1,7,0]
```

8.2.3 Maybe

En användbar Haskelltyp är `Maybe`. Den används när man inte vill kasta ett `error`, men vill att programmet inte ska göra något givet fel indata.

```
import Prelude hiding (Maybe(..))

data Maybe a = Nothing | Just a
  deriving (Show)

safeDiv :: Double -> Double -> Maybe Double
safeDiv x 0 = Nothing
safeDiv x y = Just (x / y)
```

8.2.4 IO

Även input-output är polymorft i Haskell. Här är det returtypen för en funktion av typ `IO a` som är `a`:

```
myfun :: IO Int
myfun = do putStrLn "hej"
           return 3

main :: IO ()
main = do
  myfun
  x <- myfun
  print x
```

8.2.5 Polymorfa funktioner

Det finns många polymorfa funktioner definierade i `Prelude`:

```
> :t id
id :: a -> a
> :t const
const :: a -> b -> a
> :t fst
fst :: (a, b) -> a
> :t snd
snd :: (a, b) -> b
> :t (,)
(,) :: a -> b -> (a, b)
```

Kan du gissa vad de gör bara från typsignaturen?

8.3 Typklasser

Än så länge har vi sett parametrisk polymorfism (eng. *parametric polymorphism*): typer och funktioner som beror på typvariabler.

Det finns även ad hoc polymorfism (eng. *ad hoc polymorphism*) där ett värde kan ändras beroende på hur det används. T.ex. `(==)`, `(+)`, `show`, `read`… betyder alla olika saker för olika typer. Detta leder till "overloading", dvs `(==)` operatorn går att använda för många olika typer.

Vi har sett hur parametrisk polymorfism uppnås i Haskell, nu ska vi titta på hur ad hoc polymorfism fungerar genom *typklasser*.

8.3.1 Show

Vi har sett att vi kan skriva show för olika typer. Vi kan även definiera vår egen `Show` instans:

```
data Color = Red | Green | Blue

colorToString :: Color -> String
colorToString Red = "red"
colorToString Green = "green"
```

```
colorToString Blue = "blue"

instance Show Color where
  show x = colorToString x
```

Samma för BinTree:

```
data BinTree a = Leaf a | Branch a (BinTree a) (BinTree a)

binTreeToString :: Show a => BinTree a -> String
binTreeToString (Leaf x) = "Leaf " ++ show x
binTreeToString (Branch x t1 t2) =
  "Branch " ++ show x ++
    " (" ++ binTreeToString t1 ++ ") (" ++ binTreeToString t2 ++ ")"

instance Show a => Show (BinTree a) where
  show = binTreeToString
```

8.3.2 Eq

Vi kan även instansiera Eq typklassen så att vi kan använda == och /= för `Color`:

```
sameColor :: Color -> Color -> Bool
sameColor Red Red = True
sameColor Green Green = True
sameColor Blue Blue = True
sameColor _ _ = False

instance Eq Color where
  (==) = sameColor
```

Tester:

```
> Red == Green
False
> Red == Red
True
> Red /= Green
True
```

8.3.3 Deriving

Vi kan även få Haskell att automatisk generera typklassinstanser åt oss:

```
data Color = Red | Green | Blue
  deriving (Show,Eq)
```

Tester:

```
> Red == Green
False
> Red == Red
True
```

```
> show Red
"Red"
```

För BinTree:

```
data BinTree a = Leaf a | Branch a (BinTree a) (BinTree a)
  deriving (Show,Eq)
```

Tester:

```
> tree1
Leaf 42
> tree2
Branch 1 (Leaf 2) (Branch 7 (Leaf (-1)) (Leaf 0))
> tree1 == tree2
False
> tree1 == tree1
True
```

Obs: den kod som genereras av `deriving` är inte alltid den vi vill ha. T.ex. om vi representerar rationella tal `x / y` som ett par `(x,y)` så kommer den genererade `Eq` instansen jämföra paren element för element. Istället vill vi ju använda att `x / y == z / w` omm `x * w == z * y`.

8.4 Övningar

- Implementera rationella tal `x / y` så som förklaras ovan. Definiera deras `Eq` instans så att rätt jämförelse används.

Kapitel 9

Funktionell programmering i Haskell 4

9.1 Högre ordningens funktioner

När vi har pratat variabler och värden har vi hittills framförallt pratat om heltal, flyttal, strängar, listor, med mera. Det finns goda skäl att lägga *funktioner* till uppräkningen av värden att arbeta med, vilket handlar om möjligheten att skapa nya funktioner när man behöver dem, skicka funktioner som argument, returnera funktioner, och kunna tillämpa operatorer på funktioner. När ett programmeringsspråk har de möjligheterna säger man att funktioner behandlas som *first class citizens* (eller *first class objects* eller *first class functions*). Detta är en av grundegenskaperna i funktionella språk och något som vi redan sett en del av.

En funktion som tar in en annan funktion eller returnerar en funktion är en så kallad "högre ordningens funktion" (eng. *higher order function*). Då funktioner är som vilka andra värden som helst i Haskell kan man även lätt skriva egna högre ordningens funktioner.

9.1.1 `map` för listor

Poängen med högre ordningens funktioner är att de är ett sätt att generalisera funktionalitet. Genom att identifiera vanliga mönster för beräkningar och låta dessa implementeras med högre ordningens funktioner kan man förenkla, korta ner, och i många fall snabba upp sin kod. Några exempel på beräkningar man kan behöva:

1. Ta en lista med tal och returnera en lista med dess kvadrater.
2. Ta en lista med icketomma listor och returnera en lista med det första elementet från varje lista.
3. Ta en lista med strängar och räkna ut deras längder.

Dessa tre beräkningar har gemensamt att man ska utföra något på varje element i en lista. I imperativa språk hade man skrivit en loop för varje beräkning, men i språk med stöd för högre ordningens funktioner kan man istället använda funktionen `map`:

```
square :: Int -> Int
square x = x ^ 2
```

```
squares :: [Int] -> [Int]
squares xs = map square xs

heads :: [[Int]] -> [Int]
heads xs = map head xs

lengths :: [[a]] -> [Int]
lengths xs = map length xs
```

Tester

```
> squares [1,2,3]
[1,4,9]
> heads [[1,2,3],[4],[9..100]]
[1,4,9]
> lengths [[1,2,3],[4],[9..100]]
[3,1,92]
```

Så här är `map` definierad:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Som ni ser kan man hantera funktioner precis som andra värden och definitionen av `map` är inte mer komplicerad än andra funktioner vi redan sett i kursen.

9.1.1.1 Inte bara för listor

Vi kan även definiera funktioner som motsvarar `map` för andra typer än listor:

```
data BinTree a = Leaf a | Branch a (BinTree a) (BinTree a)
  deriving (Show)

mapBinTree :: (a -> b) -> BinTree a -> BinTree b
mapBinTree f (Leaf x) = Leaf (f x)
mapBinTree f (Branch x t1 t2) = Branch (f x) (mapBinTree f t1) (mapBinTree f t2)

t1 :: BinTree Int
t1 = Branch 1 (Leaf 2) (Leaf 3)
```

Test:

```
> mapBinTree square t1
Branch 1 (Leaf 4) (Leaf 9)
```

Det finns så klart en typklass för detta. Den heter `Functor` vilket hänger ihop med ett koncept från *kategoriteori* (en gren av matematiken som handlar om väldigt generell abstrakt algebra).

9.1.2 `filter`

Ett annat vanligt behov när man programmerar är att välja ut element ur en lista och till det finns funktionen `filter`. Den tar ett predikat och filtrerar ut de element som uppfyller predikatet.

```
isEven :: Int -> Bool
isEven x = mod x 2 == 0

isOdd :: Int -> Bool
isOdd x = not (isEven x)

evens :: [Int] -> [Int]
evens xs = filter isEven xs

odds :: [Int] -> [Int]
odds xs = filter isOdd xs

isEmpty :: [a] -> Bool
isEmpty [] = True
isEmpty _  = False

nonEmpty :: [a] -> Bool
nonEmpty [] = False
nonEmpty _  = True

nonEmpties :: [[a]] -> [[a]]
nonEmpties xs = filter nonEmpty xs
```

Tester:

```
> evens [1..10]
[2,4,6,8,10]
> odds [1..10]
[1,3,5,7,9]
> nonEmpties [[1..10],[],[],[2],[3,4],[]]
[[1,2,3,4,5,6,7,8,9,10],[2],[3,4]]
```

Så här kan man implementera `filter`:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

9.1.3 Partiell applikation igen

Vi såg i första Haskellföreläsningen att vi kan skriva

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

och få en funktion `Int -> Int` genom att ge den två argument:

```
> :t addThree 1 2
addThree 1 2 :: Int -> Int
```

vi kan därför även skriva

```
> map (addThree 1 2) [1,2,3]
[4,5,6]
```

På samma sätt kan vi skriva:

```
> map (2 *) [1,2,3]
[2,4,6]
> map (1 +) [1,2,3]
[2,3,4]
> filter (<3) [1..10]
[1,2]
```

9.1.4 Zipping: `zip`/`unzip`/`zipWith`

Några väldigt användbara funktioner är `zip` och `unzip`. Dessa funktioner låter oss skapa och dela upp en lista av par.

```
> :t zip
zip :: [a] -> [b] -> [(a, b)]
> :t unzip
unzip :: [(a, b)] -> ([a], [b])
> zip [1..10] [10..1]
[]
> zip [1..10] [2..11]
[(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,10),(10,11)]
> zip [1..10] [True,False,True,True]
[(1,True),(2,False),(3,True),(4,True)]
> unzip [(1,-1),(2,1),(6,9)]
([1,2,6],[-1,1,9])
```

Det finns även en väldigt användbar högre ordningens funktion som låter oss istället slå ihop elementen som vi zippar med hjälp av en funktion.

```
> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
> zipWith (+) [1..10] [1..10]
[2,4,6,8,10,12,14,16,18,20]
> zipWith (*) [1..10] [1..10]
[1,4,9,16,25,36,49,64,81,100]
> zipWith (,) [1..10] [1..10]
[(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10)]
```

Följande variant av `zipWith` kan nog hjälpa lite på labben (uppgiften om `matrixToList`):

```
> zipWith (:) [1,2,3] [[1,2],[3,4],[5,6]]
[[1,1,2],[2,3,4],[3,5,6]]
> :t zipWith (:)
zipWith (:) :: [a] -> [[a]] -> [[a]]
```

Med hjälp av `zipWith` kan vi skriva definitionen av Fibonaccitalen på en rad:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Här ser vi hur kraftfullt lathet, rekursion och högre ordningens funktioner är tillsammans. Vi genererar en oändlig lista `fibs` med Fibonaccital som börjar med `0` och `1` och sen adderas varje tal till det föregående i listan.

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

9.1.5 Currying: `curry`/`uncurry`

Ett viktigt koncept i funktionell programmering är *currying*. Det har inget med mat att göra utan det har fått namnet från logikern som gett Haskell sitt namn: https://en.wikipedia.org/wiki/Haskell_Curry

Det går ut på att vi alltid kan konvertera en funktion som tar två individuella argument till en funktion som bara tar ett par.

```
> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Detta kan användas på följande sätt:

```
> curry fst 2 3
2
> uncurry (*) (3,4)
12
```

En viktig konsekvens av currying är att vi kan se en funktion som tar `n` argument som en funktion från en `n`-tupel. Detta har teoretiska fördelar då man kan se alla funktioner som en funktion som tar in en sak och returnerar en sak.

9.1.6 Folds: `foldr`

Vi har sett att högre ordningens funktioner är väldigt användbara för att iterera över data och på så sätt förenkla kod och få väldigt kompakta funktioner. En annan väldigt vanlig operation är att på något sätt slå ihop alla element i en datastruktur, t.ex. att ta summan eller produkten av alla element i en lista.

Följande kodsnuttar beräknar summan och produkten av elementen i två listor:

```
sumRec :: [Int] -> Int
sumRec [] = 0
sumRec (x:xs) = x + sumRec xs

prodRec :: [Int] -> Int
prodRec [] = 1
prodRec (x:xs) = x * prodRec xs
```

Tester

```
> sumRec [1,2,3,4]
10
> prodRec [1,2,3,4]
24
```

Den enda skillnaden mellan dessa två funktioner är att `sumRec` använder `+` och `0`, och `prodRec` använder `*` och 1. Vi kan generalisera dessa till en funktion som tar in en funktion `f` och ett element `e`:

```
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr _ e [] = e
myFoldr f e (x:xs) = f x (myFoldr f e xs)

mySum :: [Int] -> Int
mySum xs = myFoldr (+) 0 xs

myProd :: [Int] -> Int
myProd xs = myFoldr (*) 1 xs
```

Tester:

```
> mySum [1,2,3,4]
10
> myProd [1,2,3,4]
24
```

Vi kan nu lätt skriva en funktion som använder exponentiering som operation:

```
myExps :: [Integer] -> Integer
myExps xs = myFoldr (^) 1 xs
```

Test

```
> myExps [2,3,4]
2417851639229258349412352
```

Obs: detta beräknar `2^(3^4) = 2^81`.

Vi kan även skriva en funktion för att slå ihop listor av listor:

```
myConcat :: [[a]] -> [a]
myConcat xs = myFoldr (++) [] xs
```

Test

```
> myConcat [[1,2,3],[],[4],[5,6]]
[1,2,3,4,5,6]
```

Högre ordningens funktioner är alltså mycket smidiga för att skriva väldigt generella funktioner och minimera duplikation av kod!

Det finns en `foldr` funktion i Haskell med följande typ:

```
> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Som vanligt är den mer generell än min `myFoldr` funktion ovan då den fungerar för alla typer `t` som är `Foldable`.

I Python finns det en funktion som fungerar exakt som `foldr`, men den heter `reduce`. Den ligger i biblioteket `functools` och importeras på följande sätt:

```
import functools

functools.reduce(lambda s, x: s+x, [1,2,3], 0)
```

Anledningen att den heter `reduce` är att det är som den andra delen i den kända MapReduce idiomet för att behandla stora data: https://en.wikipedia.org/wiki/MapReduce

9.1.7 Funktionskomposition (`.`)

En högre ordningens funktion som vi alla är bekanta med från matematik är funktionskomposition. Givet funktioner `f` och g kan vi skapa deras sammanslagning genom `f` ∘ g. Detta är en ny funktion som fungerar på följande sätt: (`f` ∘ g)(x) = `f`(g(x)). Detta finns definierat i Haskell och heter helt enkelt bara punkt (`.`):

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
> ((+1) . (*2)) 3
7
> (negate . (*2)) 3
-6
```

Detta är väldigt användbart i kombination med andra högre ordningens funktioner:

```
> map (negate . (*2)) [1,2,3]
[-2,-4,-6]
> filter (not . isEven) [1..10]
[1,3,5,7,9]
```

Detta är ännu ett sätt där vi på ett enkelt sätt kan skriva väldigt matematisk kod på ett enkelt sätt i Haskell.

9.1.8 Funktionsapplikation (`$`)

En till högre ordningens funktion som vi kan definiera är funktionsapplikation:

```
> :t ($)
($) :: (a -> b) -> a -> b
```

Det kan tyckas som om detta är en helt oanvändbar funktion då vi redan kan applicera funktioner genom att bara skriva mellanslag. Men det finns faktiskt massor med användningsområden. Det främsta är att vi kan använda det för att undvika att skriva en massa paranteser

```
> negate (succ (length [1,2,3]))
-4
> negate $ succ $ length [1,2,3]
-4
> (negate . succ . length) $ [1,2,3]
-4
```

Vi kan även använda (`$`) för att applicera en lista av funktioner på en lista av indata:

```
> :t zipWith ($)
zipWith ($) :: [a -> c] -> [a] -> [c]
```

```
> zipWith ($) [(+ 1),(* 2),(^ 3),negate] [1..10]
[2,4,27,-4]
```

9.1.9 Iterera funktioner

En sista användbar högre ordningens funktion är iterering. Givet en funktion `f` och ett startvärde `x` så kan genererar den en oändlig lista `[x,f(x),f(f(x)),f(f(f(x)))...]`.

```
> :t iterate
iterate :: (a -> a) -> a -> [a]
> take 10 (iterate (*2) 2)
[2,4,8,16,32,64,128,256,512,1024]
> iterate (*2) 1 !! 4
16
```

På detta sätt kan vi göra många loopar på ett enkelt sätt. Definiera kroppen som en funktion och använd `iterate` för att generera en oändlig lista med fler och fler funktionsanrop.

9.2 Anonyma funktioner och λ-uttryck

Begreppet *anonyma funktioner* med hjälp av *λ-uttryck* finns i många språk idag. Man säger att funktionerna är *anonyma* eftersom man inte ger dem ett namn, utan bara skapar dem. Tag kodsnutten

```
> map (\x -> x * 2) [1,2,3]
[2,4,6]
```

Den innehåller ett λ-uttryck `\x -> x * 2` vilket är som en funktion som tar ett argument `x` och sedan multiplicerar det med 2. Man har dock inte gett funktionen något namn. Dessa uttryck är väldigt smidiga för att skapa tillfälliga små funktioner att använda tillsammans med högre ordningens funktioner. Fler exempel:

```
> zipWith (\x y -> x + y ^ x) [1,2] [3,4]
[4,18]
> foldr (\x y -> x + y) 0 [1,2,3]
6
> foldr (\x y -> x + y ^ x) 0 [1,2,3]
12
> foldr (\x y -> x * y) 1 [1,2,3]
6
> foldr (\x y -> x * y) 0 [1,2,3]
0
```

Anledningen att dessa heter λ-uttryck är från den så kallade λ-kalkylen. Detta är en beräkningsmodell uppfunnen av logikern Alonzo Church på 1930-talet (https://en.wikipedia.org/wiki/Lambda_calculus). I denna modell är allting uppbyggt från funktioner och variabler (så t.ex. heltal representeras även de som funktioner), och man kan på detta sätt representera alla beräkningar som kan göras med en Turingmaskin genom att bara använda λ-uttryck. λ-kalkylen utgör grunden för det funktionella paradigmet och varianter av den utgör beräkningsmodellen som används för att förklara hur funktionella språk fungera. Detta skiljer sig alltså från imperativa språk där modellen istället var som en vanlig dator med minne och processor. På detta sätt är funktionella språk mer matematiska än imperativa språk. En intressant konsekvens av detta är att de låter oss enklare arbeta med just funktionskomposition.

9.3 Listomfattningar

Listomfattning (eng. *list comprehension*) är en teknik för att skapa listor som också är populär bland funktionella språk. Listomfattning är praktiskt för att initiera listor med värden att jobba vidare med eller för att modifiera listor. En fördel, som kanske ligger bakom listomfattningars populäritet, är att de kan vara mycket tydliga tack vare sin nära koppling till matematisk notation. Betrakta till exempel mängden av kvadraten av talen upp till 10:

```
> [ x ^ 2 | x <- [0..10] ]
[0,1,4,9,16,25,36,49,64,81,100]
```

Man kan också filtrera elementen—inte alla behöver presenteras. Till exempel alla udda heltal upp till 20 kan skrivas så här:

```
> [ x | x <- [0..20], mod x 2 == 1 ]
[1,3,5,7,9,11,13,15,17,19]
```

Sammanfattning: en listomfattning har tre delar:

- resultatdelen, med ett uttryck som ger elementen i resultatet,
- generatordelen, där `<-` används för att plocka fram värden till resultatdelen, och
- predikatdelen, som bestämmer vilka element som ska presenteras.

Mer komplicerat exempel:

```
> [ (x,y) | x <- [2..8], y <- [2..8], mod x y == 0 ]
[(2,2),(3,3),(4,2),(4,4),(5,5),(6,2),(6,3),(6,6),(7,7),(8,2),(8,4),(8,8)]
```

Detta är alltså ett mycket smidigt sätt för att även få alla kombinationer av element i två listor.

9.3.1 Quicksort

Vi kan skriva `sort` funktionen som vi tittade på i en annan föreläsning mycket kortare med hjälp av listomfattningar.

```
sort :: [Integer] -> [Integer]
sort []     = []
sort (p:xs) = sort [x | x <- xs, x < p] ++ [p] ++ sort [x | x <- xs, x >= p]
```

Test:

```
> sort [3,1,2,4,1,-2]
[-2,1,1,2,3,4]
```

9.3.2 Beräkna alla primtal

Vi kan även implementera Erathostenes såll för att generera all primtal på ett väldigt kompakt sätt genom:

```
primes :: [Integer]
primes = sieve [2..]
  where
  sieve :: [Integer] -> [Integer]
  sieve [] = []
  sieve (p:xs) = p : sieve [x | x <- xs, mod x p /= 0]
```

Test:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

9.4 Övningar

- Implementera `zip`, `unzip` och `zipWith`.
- Skriv en `foldr` funktion för `BinTree`.

Kapitel 10

Logikprogrammering i Prolog 1

10.1 Introduktion

Vi är nu klara med funktionell programmering i den här kursen. Vi ska nu ha tre lektioner med logikprogrammering i språket Prolog. Detta är det sista paradigmet i kursen och det påminner till viss del om funktionell programmering, men det är också väldigt annorlunda jämfört med de paradigm ni sett än så länge.

Idag ska vi gå igenom grunderna i logikprogrammering och Prolog:

- Bakgrund och lite om logik
- Fakta, regler och frågor
- Aritmetik i Prolog

10.2 Logikprogrammering

Paradigmet logikprogrammering skiljer sig till stor del från de paradigm vi sett tidigare i kursen. Vissa idéer delas med funktionell programmering, men det finns många stor skillnader. Logikprogrammering bygger på *deklarativ* programmering vilket kan beskrivas genom:

`"Säg vad du vill göra, inte hur du vill göra det."`

Grunderna för logikprogrammering kommer från matematisk logik och det bygger på ett delsystem till predikatlogik som heter Horn klausuler (eng. *Horn clasues*). Till skillnad från imperativa språk som C och Java skriver man inte program som en sekvens av instruktioner som körs i ordning utan man skriver istället en samling axiom och regler vilka definierar relationen mellan olika objekt. Ett program är sedan ett (konstruktivt) bevis för ett påstående som kan härledas från axiomen och reglerna. Den underliggande beräkningsmodellen för logikprogrammering är alltså *automatisk teorembevisning* och program hittas genom att söka efter bevis. Detta gör logikprogrammering till ett ganska unikt paradigm, men det är även väldigt uttrycksfullt och vi kommer se att man kan lösa många problem med väldigt lite kod precis som i funktionell programmering.

Sammanfattning av logikprogrammering:

- Deklarativ programmering med hjälp av **predikat** som beskriver relationer mellan objekt.

- Domänen beskrivs som en sammansättning av predikatlogiska formler i *Horn klausuler*.
- Frågor (eng. *queries*) besvaras genom en sök-algoritm som heter **resolution**. En viktig del av denna algoritm är **unifiering** (eng. *unification*).

Kombinationen av att använda en begränsad logik i form av Horn klausuler och resolution är vad som möjliggör för logikprogrammering att skapa snabba program.

10.2.1 Lite om matematisk logik

Logikprogrammering bygger, som namnet antyder, på logik. Mer specifikt bygger det på predikatlogik (eng. *first order logic*). Det finns många olika sätt att presentera detta system och vi kommer fokusera på varianten som används i Prolog.

Detta system har de vanliga logiska konnektiven:

- $x \wedge y$: konjunktion (läses: "`x och y`")
- $x \vee y$: disjunktion (läses: "`x eller y`")
- $x \rightarrow y$: implikation (läses: "`om x så y`")
- $\neg x$: negation (läses: "`inte x`")

Det finns även konstanter: atomer, siffror, och flyttal. Atomer skrivs med liten först bokstav, så `anders` är en atom.

Det finns även variabler, dessa skrivs med stor första bokstav, så `X` är en variabel.

Predikat skrivs `p(X,Y)`. Exempelvis kan vi ha ett predikat `undervisar(T,C)` där `T` representerar en lärare och `C` en kurskod. Så följande uttryck bör betraktas som sant `undervisar(anders,da2004)`. Ett predikat kan ses som en funktion till sanningsvärden, men predikat är mer generella än vanliga funktioner (mer om det nedan).

Vi har även två kvantifikatorer:

- $\forall X_1, \ldots, X_n.\ p(X_1, \ldots, X_n)$: allkvantifiering (läses: "`för alla tilldelningar av värden till` $X_1, \ldots, X_n$ `så håller` $p(X_1, \ldots, X_n)$")
- $\exists X_1, \ldots, X_n.\ p(X_1, \ldots, X_n)$: existenskvantifiering (läses: "`för någon tilldelning av värden till` $X_1, \ldots, X_n$ `så håller` $p(X_1, \ldots, X_n)$")

Exempel:

- $\forall$`C. undervisar(anders,C)`: "anders undervisar alla kurser" (som tur är är detta falskt).
- $\exists$`C. undervisar(anders,C)`: "anders undervisar någon kurs" (detta är sant då jag undervisar både da2004 och da4003).

Obs: $\exists$ betyder inte *unik* existens, det betyder bara att det finns *minst* en tilldelning som gör påståendet sant.

10.2.1.1 Funktioner, relationer och predikat

Om f är en funktion kan vi definiera den genom en samling ekvationer på formen $f(x) = a$ (som i Haskell), men vi kan även definiera den som en samling par $(x, a) \in f$ (dvs, genom dess *graf*). Vi kan även specificera den som ett predikat $f(x, a)$. Så vi kan alltså definiera funktioner med hjälp av predikat.

En viktig egenskap hos matematiska funktioner är att de givet en indata ger bara en utdata. Men hos generella predikat kan vi associera många saker till en given "indata", exempelvis har vi

`undervisar(anders,da2004)` och `undervisar(anders,da4003)`. Så `undervisar` är inte en funktion från personer till kurskoder.

Så hur kan man använda predikatlogik för att göra beräkningar?

Exempel 1: Låt oss säga att `undervisar(anders,da2004)` och `undervisar(anders,da4003)`. Vad är det då för beräkning som hänger ihop med påståendet $\exists$`C. undervisar(anders,C)`? Jo, det är ju en funktion som antingen returnerar `da2004` eller `da4003`. Så genom att bevisa påståendet får vi ett program som räknar upp alla konstanter som uppfyller det.

Exempel 2: Låt oss säga att vi vet att $\forall$`T. undervisar(T,da2004)` $\rightarrow$ `undervisar(T,da4003)` och `undervisar(anders,da2004)`, då kan vi ju dra slutsatsen `undervisar(anders,da4003)`. Så vi ser att implikation hänger ihop med funktionsapplikation.

10.2.1.2 Horn klausuler

Prolog bygger på ett delsystem till predikatlogik som använder formler på en speciell form som kallas *Horn klausuler*. En sådan ser ut på följande sätt:

$$B_1 \wedge \cdots \wedge B_n \rightarrow A$$

Vi säger att

- A är *huvudet*
- $B_1, \ldots, B_n$ är *kroppen*

Dessa låter oss specificera *regler* för att definiera mer komplexa predikat som impliceras av enklare:

$\forall$`X Y.`$\exists$`Z. far(X,Z)` $\wedge$ `mor(Z,Y)` $\rightarrow$ `farmor(X,Y)`

Detta läses som "`farmodern till X är Y om fadern till X är (någon) Z och modern till Z är Y`".

Det som gör denna typ av formler så användbara för programmering är att man kan läsa dom på följande "procedurellt" sätt: "för att visa att `Y` är farmor till `X` behöver vi visa att det finns `Z` som är far till `X` och som har `Y` som mor". Så denna typ av formler kan ses som specifikation av en procedur/algoritm för att bevisa huvudet genom att först bevisa allt i kroppen. Denna observation är grunden för logikprogrammering.

10.3 Prolog

Prolog är det mest kända och välanvända språket inom logikprogrammering.

10.3.1 Installation

Vi ska använda SWI-Prolog i den här kursen: https://www.swi-prolog.org/

Det är en specifik implementation av Prolog och filtypen associerad till Prolog filer är `.pl`

10.3.2 Dokumentation och tutorials

Det finns massor med dokumentation och tutorials för Prolog online. Nedan följer några resurser:

- Referensmanual: https://www.swi-prolog.org/pldoc/doc_for?object=manual
- Gratis bok: http://learnprolognow.com/

- En tutorial: https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html

10.3.3 Hello, World!

Som vanligt börjar vi med att skriva "Hello, World!" programmet. Precis som med Haskellprogram kan vi köra Prologprogram i en tolk. Den heter `swipl` för SWI-Prolog:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- write("Hello, World!").
Hello, World!
true.
```

Vi kan även lägga vår kod i en fil och ladda den i tolken. Låt oss säga att vi har en fil `hello_world.pl` som innehåller

```
:- initialization hello_world.

hello_world :- write('Hello, World!\n').
```

och sedan kör följande kommando

```
$ swipl hello_world.pl
Hello, World!
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- halt.
```

Som ni ser skrivs `Hello, World!` ut innan tolken laddas. Vi skriver sedan in `halt.` för att lämna tolken. Hade vi istället haft en fil med

```
:- initialization hello_world, halt.

hello_world :- write('Hello, World!\n').
```

då hade följande hänt

```
$ swipl hello_world.pl
Hello, World!
```

Vi ser att `initialization` instruktionen säger åt Prolog vilken regel den ska börja med. I det här fallet är det `hello_world`. När den är klar med det körs kommandot `halt` och körningen avslutas.

10.4 Fakta, regler och frågor

Det finns bara tre grundläggande konstruktioner i Prolog: fakta, regler och frågor. En samling fakta och regler är en kunskapsbas (eller databas) och Prolog programmering handlar om skriva såna. Så ett program i Prolog är bara en kunskapsbas och vi använder frågor (eng. *queries*) för att "köra" ett sådant program. Vi har allså:

- Fakta: grundläggande uttryck i kunskapsbasen (e.g. "Socrates was a man")
- Regler: implikationer och inferenser mellan fakta (e.g. "All men are mortal")
- Frågor: frågeställningar till kunskapsbasen (e.g. "Was Socrates mortal?")

Detta påminner om databasspråk som SQL, men det faktum att Prolog har en automatisk teorembevisare under skalet gör det väldigt kraftfullt. När vi ställer en fråga till Prolog så kommer systemet gör en uttömmande sökning genom kunskapsbasen med hjälp av backtracking och unifikation.

Det här låter nog som ett ganska konstigt sätt att programmera, speciellt om man är van vid imperativ programmering där vi bara ger datorn en lista med instruktioner som körs en efter en. Vi ska se att vi faktiskt kan skriva intressanta program i Prolog. Låt oss börja med några exempel på fakta, regler och frågor.

10.4.1 Exempel 1: undervisar

Om vi startar `swipl` så kan vi lära Prolog lite nya fakta:

```
?- assert(undervisar(anders,da2004)).
true.

?- assert(undervisar(anders,da4003)).
true.
```

Att `true.` skrivs ut är bara Prologs sätt att säga att allt har gått bra (så på samma sätt som vi kör `return 0;` i C när vi är klara). Vi kan sedan ställa olika frågor till Prolog:

```
?- undervisar(anders,da4003).
true.

?- undervisar(anders,da2004).
true.

?- undervisar(anders,mm2001).
false.
```

Som vi ser svarar Prolog `true` för all fakta som den känner till och `false` för allt annat. Självklart vet inte Prolog vad `anders`, `da2004` och `da4003` är för något. Dessa är bara abstrakta logiska konstanter och Prolog kan bara dra slutsatser om sånt som vi har lärt den. Prolog följer en sluten värld modell (eng. *closed-world assumption*), så den betraktar allt som vi inte specificerat som falskt.

Vi kan även fråga Prolog om vilka kurser jag undervisar:

```
?- undervisar(anders,X).
X = da2004 ;
X = da4003.
```

Som vi ser returneras en uppräkning av alla kurser `X` där vi har `undervisar(anders,X)` i kunskapsbasen. För att få fram fler svar skriver man in ";" i `swipl`. Detta är ett exempel där predikatet `undervisar` inte direkt motsvarar en funktion från personer till kurskoder.

Vi kan även vända på frågan och få fram en lista på alla lärare som undervisar `da2004`:

```
?- undervisar(X,da2004).
X = anders.
```

Om vi låtsas att `undervisar` är en funktion ser vi här att vi enkelt kan vända på våra "funktioner" och få deras invers. Detta kan man vanligtvis inte göra i andra språk.

I verkligheten är det dock inte sant att bara jag ger `da2004` då den ges flera gånger per termin med olika lärare. Vi kan lätt lära Prolog om detta:

```
?- assert(undervisar(kristoffer,da2004)).
true.

?- assert(undervisar(lars,da2004)).
true.

?- undervisar(X,da2004).
X = anders ;
X = kristoffer ;
X = lars.
```

Vi kan även fråga prolog om alla `X` och `Y` som uppfyller `undervisar(X,Y)`:

```
?- undervisar(X,Y).
X = anders,
Y = da2004 ;
X = anders,
Y = da4003 ;
X = kristoffer,
Y = da2004 ;
X = lars,
Y = da2004.
```

Vi kan även lägga till regler:

```
?- assert(undervisar(X,da4003) :- undervisar(X,da2004)).
true.
```

Detta ska läsas som en Horn klausul $\forall$`X. undervisar(X,da2004)` $\rightarrow$ `undervisar(X,da4003)`. Detta är så klart inte sant i verkligheten, men vi kan lära Prolog vad vi vill. Vi kan sedan fråga Prolog om vem som undervisar `da4003`:

```
?- undervisar(X,da4003).
X = anders ;
X = anders ;
X = kristoffer ;
X = lars.
```

Varför fick vi `anders` två gånger? Då det finns två sätt att få `undervisar(anders,da4003)`, ett genom regeln vi nyss la till och ett genom faktat vi la in tidigare, returnerar Prolog två lösningar. Vi kan glömma

bort faktat vi la in med hjälp av `retract`:

```
?- retract(undervisar(anders,da4003)).
true .

?- undervisar(X,da4003).
X = anders ;
X = kristoffer ;
X = lars.
```

Att lägga till fakta och regler i tolken direkt är inte lika bra som att göra det i en fil då vi blir av med allt när vi stänger `swipl`. Om vi istället har följande i en fil som heter `undervisar.pl`:

```
undervisar(anders,da2004).
undervisar(kristoffer,da2004).
undervisar(lars,da2004).

undervisar(X,da4003) :- undervisar(X,da2004).
```

då kan vi köra:

```
$ swipl undervisar.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- undervisar(X,da4003).
X = anders ;
X = kristoffer ;
X = lars.
```

10.4.2 Exempel 2: släktträd

Låt oss säga att vi har följande fil med släktinformation:

```
far(john,erik).
far(erik,lars).
far(anders,erik).
far(olle,anders).
far(lisa,anders).

mor(john,johanna).
mor(anders,linda).
mor(lisa,eva).
mor(olle,eva).
```

Då kan vi ladda den och ställa lite frågor (flaggan `-q` skippar all info i början av en session):

```
$ swipl -q slakttrad.pl
?- far(X,erik).
```

```
X = john ;
X = anders.
?-
```

Obs: Prolog vet såklart inte om `far(john,erik)` ska läsas som `john är far till erik` eller `johns far är erik`. Vi kan gissa oss till vilket det ska vara från `far(lisa,anders)`, men Prolog vet så klart inte detta. Därför kan det vara bra att lägga in en kommentar, detta görs genom att börja raden med `%`:

```
% far(X,Y) betyder att Y är far till X
far(john,erik).
...
```

Låt oss nu lägga till regeln $\forall$`X Y.`$\exists$`Z. far(X,Z)` $\wedge$ `mor(Z,Y)` $\rightarrow$ `farmor(X,Y)`. Detta görs genom att skriva:

```
farmor(X,Y) :- far(X,Z), mor(Z,Y).
```

Vi kan sedan ladda om filen genom att köra `make.` och då kommer vi ha tillgång till `farmor` predikatet:

```
?- make.
true.

?- farmor(lisa,X).
X = linda.

?- farmor(olle,X).
X = linda.

?- farmor(anders,X).
false.
```

Notera att Prolog säger `false` om den inte hittar något svar på frågan.

Vi ser alltså för att skriva en Horn klausul på formen

$$B_1 \wedge \cdots \wedge B_n \rightarrow A$$

så skriver vi följande i Prolog

```
A :- B_1, ..., B_n.
```

Detta är ett vettigt sätt att skriva på då vi kan tänka att för att lösa `A` så måste vi först lösa alla uttryck `B_1`, ..., `B_n`.

På liknande sätt kan vi lägga till `farfar` predikat:

```
farfar(X,Y) :- far(X,Z), far(Z,Y).
```

Med detta kan vi nu göra ett nytt predikat som returnerar alla personer som vi vet är barnbarn till någon i kunskapsbasen. I detta fall är det endast personer som har en `farmor` *eller* `farfar` som är barnbarn, men vi kan enkelt lägga till `mormor` och `morfar`. Precis som i Haskell används _ för "vad som helst" (dvs, det är ett "wild-card"):

```
barnbarn(X) :- farmor(X,_).
barnbarn(X) :- farfar(X,_).
```

Vi kan nu fråga Prolog om en lista över alla som är barnbarn till någon:

```
?- barnbarn(X).
X = olle ;
X = lisa ;
X = john ;
X = anders ;
X = olle ;
X = lisa.
```

Vi ser alltså att *konjunktion* skrivs med , i klausuler och *disjunktion* som nya klausuler med radbryt mellan.

Hade vi istället skrivit

```
barnbarn(X) :- farmor(X,Y).
barnbarn(X) :- farfar(X,Y).
```

så hade vi fått varningen:

```
?- make.
Warning: /tmp/slakttrad.pl:15:
        Singleton variables: [Y]
Warning: /tmp/slakttrad.pl:16:
        Singleton variables: [Y]
true.
```

En "singleton variable" är en variabel som bara finns en gång i en klausul. Då detta är ett vanligt sätt att introducera buggar ger Prolog en varning om man råkar göra det. Som vi såg ovan kan denna typ av variabler alltid bytas ut mot _ och vi får då ingen varning.

Obs: precis som för mönstermatchning i Haskell spelar ordningen roll i Prolog! Konjunktioner (dvs klausuler) läses *vänster till höger* och disjunktioner (dvs regler) läses *uppifrån och ned*.

10.4.2.1 Användbart kommando i `swipl`: `apropos`

Ett väldigt användbart kommando i `swipl` är `apropos`. Låt oss säga att vi inte visste att `make` laddar om filen, då kunde vi skriva följande för att få fram all info om "reload":

```
?- apropos(reload).
% SWI reload_library_index/0              Force reloading the index after modifying the set of library
% LIB reload_foreign_libraries/0          Reload all foreign libraries loaded (after restore of a stat
% SEC editreload                          The test-edit-reload cycle
% LIB http_reload_with_parameters/3       Create a request on the current handler with replaced search
% SEC loadrunningcode                     Reloading files, active code and threads
% LIB rdf_make/0                          Reload all loaded files that have been modified since the la
true.
```

10.5 Aritmetik i Prolog

Prolog har precis som många andra språk både heltal och flyttal. Dessa kan manipuleras med vanliga operationer och för att jämföra ett värde med en beräkning används `is`:

```
?- 2 is 1 + 1.
true.
```

```
?- 4 is 2 * 2.
true.

?- -2 is 8 - 10.
true.

?- 3 is 6 / 2.
true.

?- 3 is 7 / 2.
false.

?- 3.5 is 7 / 2.
true.

?- 1 is mod(7,2).
true.
```

Obs: funktionen `is` tar ett tal eller en variabel som första argument och ett uttryck som andra argument. Detta kan leda till förvirrande resultat om man inte använder den rätt.

```
?- 1 + 1 is 2.
false.
```

Vi kan även använda Prolog som en avancerad miniräknare:

```
?- X is 1 + 1.
X = 2.

?- X is 7 * 12.
X = 84.
```

Detta låter oss skriva enkla aritmetiska funktioner som predikat. Exempelvis kan vi lägga följande funktion i en fil

```
add_3_and_double(X,Y) :- Y is (X + 3) * 2.
```

och sen köra

```
?- add_3_and_double(1,Y).
Y = 8.

?- add_3_and_double(2,Y).
Y = 10.
```

Obs: vi har sett tidigare att man kan vända på funktioner i Prolog. Detta funkar *inte* för aritmetiska funktioner och `is`. Så därför händer följande:

```
?- add_3_and_double(X,10).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [9] 10 is (_8318+3)*2
ERROR:    [7] <user>
```

```
ERROR:
ERROR: Note: some frames are missing due to last-call optimization.
ERROR: Re-run your program in debug mode (:- debug.) to get more detail.
```

Anledningen till detta är att aritmetik hanteras på specialsätt så att det är snabbt i Prolog. Det är generellt sett väldigt svårt att automatiskt lista ut inversen till en aritmetisk operation på ett sånt sätt att aritmetiken ändå är snabb (exempelvis är inversen till en funktion som beräknar kvadraten av ett tal dess kvadratrot vilket generellt beräknas genom iterativ approximering!).

10.5.1 Jämförelseoperatorer i Prolog

Precis som i många andra språk kan vi jämföra saker i Prolog.

```
?- 3 > 2.
true.

?- 3 >= 2.
true.

?- 2 < 3.
true.

?- 3 < 2.
false.

?- 1 + 1 =:= 2.
true.

?- 1 + 1 =\= 3.
true.

?- 1 + 1 =\= 2.
false.
```

Som vi ser används `=:=` för att testa likhet mellan två uttryck, så skillnaden mot `is` är att båda sidorna kan vara uttryck. Vi kommer se att vi använder `is` för att binda en obunden variabel till något uttryck, medans `=:=` används för jämförelse. Detta är samma som skillnaden mellan `x = 2` och `x == 2` i ett imperativt språk.

Vi kan även skriva uttryck som körs efter varandra med hjälp av ”,” (kom ihåg att dom körs vänster till höger).

```
?- X is 2, Y is X + X.
X = 2,
Y = 4.

?- X is 2, 1 + 1 =:= X.
X = 2.

?- X is 2, 1 + 2 =:= X.
false.
```

Det finns massor med aritmetiska operationer i Prolog och man kan läsa mer om dom här: https://www.swi-prolog.org/pldoc/man?section=arithpreds

Det finns även en = operator i Prolog, men den fungerar annorlunda än i andra språk.

```
?- 2 = 2.
true.

?- 2 = 3.
false.

?- 1 + 1 = 2.
false.

?- 1 + 1 = 1 + 1.
true.
```

Vi ser alltså att = inte försöker evaluera sina argument, det är istället vad `=:=` och `is` gör. Så vad gör egentligen "=" i Prolog?

Svaret är att = försöker *unifiera* sina argument:

```
?- X = 3 + 2.
X = 3+2.

?- 3 + 2 = X.
X = 3+2.

?- X = Y * Y.
X = Y*Y.

?- X * X = Y * Y.
X = Y.

?- X + Y = Z + W.
X = Z,
Y = W.

?- 2 * X = Y * 2.
X = Y, Y = 2.
```

Här ser vi att Prolog listar ut ett sätt att binda variablerna så att uttrycken blir detsamma. Det är exakt det som unifikation handlar om.

Detta betyder **inte** att Prolog kan lösa godtyckliga ekvationer:

```
?- X + X = Y * Y.
false.
```

Detta hade ju kunnat lösts med `X = 0` och `Y = 0`, men det förstår inte Prolog då man behöver ha specialkunskap om hur tal fungerar för att inse detta. Det unifikation gör är istället att hitta den mest generella lösningen till en ekvation utan att använda någon specialkunskap, mer om detta i senare föreläsningar.

Kapitel 11

Logikprogrammering i Prolog 2

11.1 Rekursion i Prolog

Precis som i Haskell finns det inga loopar i Prolog, så för att göra något många gånger måste man använda rekursion.

11.1.1 Ättlingar

Vi kan definiera ett släktträd som i förra föreläsningen (Obs: lite modifierat släktträd idag):

```
far(john,erik).
far(erik,lars).
far(anders,erik).
far(olle,anders).
far(lisa,olle).

mor(linus,lisa).
mor(anders,anna).
mor(eva,lisa).
```

Om vi vill definiera ett predikat `attling(X,Y)` som säger om `X` är ättling till `Y` (dvs, släkt i rakt nedstigande led) kan vi använda rekursion:

```
attling(X,Y) :- mor(X,Y).
attling(X,Y) :- far(X,Y).
attling(X,Y) :- mor(X,Z), attling(Z,Y).
attling(X,Y) :- far(X,Z), attling(Z,Y).
```

De två sista fallen är rekursiva då vi anropar `attling` i sig själv.

Tester:

```
?- attling(john,erik).
true

?- attling(linus,anders).
true
```

```
?- attling(linus,erik).
true

?- attling(linus,john).
false.

?- attling(X,erik).
X = john ;
X = anders ;
X = linus ;
X = eva ;
X = olle ;
X = lisa ;
false.
```

Obs: för att få fram alla ättlingar till `erik` använder vi `;` (att skriva `n` funkar också). Detta ska ni göra på labben också. När det inte finns fler fall skrivs `false` ut.

Ordningen vi skriver reglerna i `attling` spelar roll. De läses uppifrån och ner, medans konjunkterna läses vänster till höger. Kan du komma på ett släktträd där ordningen på reglerna i `attling` påverkar ordningen saker skrivs ut?

Vi måste vara försiktiga i hur vi skriver regler. Om vi istället hade skrivit:

```
attling(X,Y) :- mor(X,Y).
attling(X,Y) :- far(X,Y).
attling(X,Y) :- attling(X,Z), mor(Z,Y).
attling(X,Y) :- attling(X,Z), far(Z,Y).
```

Och sen hade frågat

```
?- attling(linus,john).
```

då hade vi hamnat i en oändlig loop! För att stoppa tryck `CTRL-c` en massa gånger och sen `a`.

För mer information om denna typ av problem och generella tips hur man kan undvika dem se: http://learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse10

11.1.2 Peanotal

Ett "Peanotal" är ett tal som antingen är `z` ("zero") eller `s` ("successor") av ett annat Peanotal.

Vi kan definiera funktioner genom rekursion:

```
smaller_than(z,s(_)).
smaller_than(s(X),s(Y)) :- smaller_than(X,Y).
```

Tester:

```
?- smaller_than(z,z).
false.

?- smaller_than(z,s(z)).
true.
```

```
?- smaller_than(s(z),s(z)).
false.

?- smaller_than(s(z),s(s(z))).
true.
```

Obs: till skillnad från andra språk som Haskell behöver vi inte lägga till någon `data`-deklaration. Prolog har inga typer och allt är antingen fakta eller regler. Så operationer på en ny "typ" som Peanotal specificeras bara genom att lägga till nya regler för hur konstanterna `z` och `s` ska fungera.

Quiz: vad hade hänt om vi istället skrivit `smaller_than(z,_)` i basfallet?

Vi kan även definiera en relation `pred(X,Y)` som säger `true` om `Y` är föregångare till `X` (dvs om `Y` har ett `s` mindre än `X`).

```
pred(s(z),z).
pred(s(s(X)),s(X)).
```

Tester:

```
?- pred(s(s(s(z))),s(s(z))).
true.

?- pred(s(s(s(z))),s(z)).
false.
```

Obs: vi kan skriva `pred` enklare genom:

```
pred(s(X),X).
```

Tänk på detta när ni gör labben. Behöver ni alla fall ni har skrivit? Ju färre fall ni har desto enklare lösning och desto lättare att debugga!

Vi kan skriva ett predikat `convert_int(X,Y)` som relaterar ett Peanotal `X` med ett vanligt Prologtal `Y` genom:

```
convert_int(z,0).
convert_int(s(X),Y) :- convert_int(X,Z), Y is 1 + Z.
```

Tester:

```
?- convert_int(z,X).
X = 0.

?- convert_int(s(z),X).
X = 1.

?- convert_int(s(s(s(s(z)))),X).
X = 4.

?- convert_int(X,0).
X = z

?- convert_int(X,1).
```

```
X = s(z)

?- convert_int(X,5).
X = s(s(s(s(s(z)))))
```

Vi kan även definiera `plus(X,Y,Z)` och `times(X,Y,Z)` så att `Z` är summan/produkten av `X` och `Y` genom:

```
plus(z,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

times(z,_,z).
times(s(M), N, Q) :-
    times(M, N, P),
    plus(P, N, Q).
```

Tester:

```
?- plus(z,s(s(z)),X).
X = s(s(z)).

?- plus(s(s(z)),z,X).
X = s(s(z)).

?- plus(s(s(z)),s(s(s(z))),X).
X = s(s(s(s(s(z))))).

?- times(z,s(z),X).
X = z.

?- times(s(s(z)),z,X).
X = z.

?- times(s(s(z)),s(s(s(z))),X).
X = s(s(s(s(s(s(z)))))).
```

Vi kan även använda `plus` för att subtrahera:

```
?- plus(s(s(z)),X,s(s(s(z)))).
X = s(z).
```

Vi kan även lätt få alla par av `X` och `Y` vars summa är 3:

```
?- plus(X,Y,s(s(s(z)))).
X = z,
Y = s(s(s(z))) ;
X = s(z),
Y = s(s(z)) ;
X = s(s(z)),
Y = s(z) ;
X = s(s(s(z))),
Y = z ;
false.
```

Vi hade istället skrivit `plus(X,Y,Z)` och `times(X,Y,Z)` genom:

```
plus(X,Y,Z) :-
    convert_int(X, XN),
    convert_int(Y, YN),
    ZN is XN + YN,
    convert_int(Z, ZN).

times(X,Y,Z) :-
    convert_int(X, XN),
    convert_int(Y, YN),
    ZN is XN * YN,
    convert_int(Z, ZN).
```

Men då hamnar:

```
?- plus(s(s(z)),X,s(s(s(z)))).
X = s(z) ;
...
```

i en oändlig loop. Anledningen är vi inte mönstermatchar på ett bra sätt i den här definitionen av `plus` och Prolog kommer då försöker med alla möjliga värden på `X`. Den första definitionen av `plus` är alltså bättre eftersom vi hjälper Prolog mer i hur själva beräkningen ska göras.

11.1.3 Fakultet

Som vi såg i en av Haskellföreläsningarna är standardexemplet på en rekursiv funktion fakultet:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Dvs, 5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1 = 120

En matematiker skriver:

$$n! = \begin{cases} 1 & \text{, om n är 0} \\ n * (n-1)! & \text{, annars} \end{cases}$$

I Prolog skriver vi

```
factorial(0,1).
factorial(N,F) :-
    N1 is N - 1,
    factorial(N1,F1),
    F is F1 * N.
```

Test

```
?- factorial(5,X).
X = 120
```

Obs: vad händer om vi anropar `factorial` med ett negativt tal?

Obs 2: följande fungerar inte:

```
?- factorial(X,120).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [8] factorial(_3766,120)
ERROR:    [7] <user>
```

Detta beror på att heltalsaritmetik hanteras speciellt i Prolog för att vara snabbare. På grund av detta fungerar de inte lika bra för att söka efter lösningar.

11.1.4 Fibonacci talen

En annan klassisk rekursiv definition är Fibonacci talen:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Det n:te Fibonacci-talet, $F(n)$, är summan av de två tidigare Fibonacci-talen. Därför kan vi skriva

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

I Prolog skriver vi:

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

Tester:

```
?- fib(2,X).
X = 2

?- fib(3,X).
X = 3

?- fib(4,X).
X = 5

?- fib(5,X).
X = 8

?- fib(6,X).
X = 13
```

Detta blir snabbt långsamt. För en guide hur man gör definitionen mycket snabbare genom memoisering: https://www.swi-prolog.org/pldoc/man?section=tabling-memoize

11.2 Listor

Precis som många andra språk har Prolog inbyggda listor. Till skillnad från Haskell kan dessa innehålla element av olika "typ" då Prolog inte har några typer.

```
?- XS = [ 1, 2, 1.231, hej, X, [1,-2,-3] ].
XS = [1, 2, 1.231, hej, X, [1, -2, -3]].
```

På detta sätt påminner Prologs listor om Pythons listor. Precis som i Haskell kan vi dock lätt plocka ut första elementet (huvudet) och resten av listan (svansen). Detta görs genom "pipe" (dvs, "|") och kan ses som en form av mönstermatchning.

```
?- [X | XS] = [ 1, 2, 1.231, hej, X, [1,-2,-3] ].
X = 1,
XS = [2, 1.231, hej, 1, [1, -2, -3]].
```

Obs: notera att värdet på X har ändrats i XS!

Man kan även hämta ut fler än bara det första elementet med hjälp av "," på följande sätt:

```
?- [X , Y , Z | XS ] = [ 1, 2, 1.231, hej, X, [1,-2,-3] ].
X = 1,
Y = 2,
Z = 1.231,
XS = [hej, 1, [1, -2, -3]].
```

Vi kan även slå ihop listor med hjälp av append:

```
?- append([1],[2,3,4],X).
X = [1, 2, 3, 4].

?- append([1,hej],[2,3,4],X).
X = [1, hej, 2, 3, 4].
```

Och testa om element finns i en lista med member:

```
?- member(42,[1,3,-1,42,hej]).
true

?- member(12,[1,3,-1,42,hej]).
false.
```

11.2.1 Enkla listfunktioner

En funktion som beräknar längden på en lista kan skriva så här (det finns även så klart redan en inbyggd length funktion):

```
mylength([],0).
mylength([_ | XS],N) :-
    mylength(XS,NXS),
    N is NXS + 1.
```

Tester:

```
?- mylength([1,2,3,4],N).
N = 4.
```

```
?- string_to_list("anders",X), mylength(X,N).
X = [97, 110, 100, 101, 114, 115],
N = 6.
```

Vår egen version av Haskells `take` funktion:

```
take(0,_,[]).
take(_,[],[]).
take(N,[X | XS],[X | YS]) :-
    M is N - 1,
    take(M,XS,YS).
```

Tester:

```
?- take(2,[1,2,3,4],XS).
XS = [1, 2]

?- take(5,[1,2,3,4],XS).
XS = [1, 2, 3, 4]

?- take(-1,[1,2,3,4],XS).
XS = [1, 2, 3, 4]

?- take(0,[1,2,3,4],XS).
XS = []
```

Vi kan även skriva `drop` rekursivt genom:

```
drop(0,XS,XS).
drop(N,[_ | XS],YS) :-
    N1 is N - 1,
    drop(N1,XS,YS).
```

Tester:

```
?- drop(2,[1,2,3,4],XS).
XS = [3, 4]

?- drop(0,[1,2,3,4],XS).
XS = [1, 2, 3, 4]
```

Vi kan dock skriva drop på ett annat sätt som inte involverar rekursion. Detta är ett exempel på deklarativ programmering. Den lista `YS` som vi vill ha är ju lösningen till ekvationen `ZS ++ YS == XS` där längden på `ZS` är `N`. Vi kan skriva detta med hjälp av `length` och `append`:

```
drop_alt(N,XS,YS) :-
    length(ZS,N),
    append(ZS,YS,XS).
```

Tester:

```
?- drop_alt(2,[1,2,3,4],XS).
XS = [3, 4].
```

```
?- drop_alt(0,[1,2,3,4],XS).
XS = [1, 2, 3, 4].
```

På ett liknande sätt kan vi även skriva reverse:

```
reverse([],[]).
reverse([X | XS],YS) :-
    reverse(XS,ZS),
    append(ZS,[X],YS).
```

```
?- reverse([1,2,3,4],XS).
XS = [4, 3, 2, 1].
```

Detta kan optimeras med hjälp av en ackumulator: http://learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse25

11.2.2 Summera jämna element i en lista

Nedan följer ett litet program för att summera alla jämna element i en lista:

```
sum_even_elements([], 0).
sum_even_elements([X | XS], Sum) :-
    0 is X mod 2,
    sum_even_elements(XS, SumXS),
    Sum is X + SumXS.
sum_even_elements([X | XS], Sum) :-
    0 =\= X mod 2,
    sum_even_elements(XS, Sum).
```

Test:

```
?- sum_even_elements([1,2,3,4],XS).
XS = 6
```

11.3 Strängar

Strängar i Prolog är listor av bokstäver på samma sätt som i Haskell, med en väldigt viktigt skillnad. En bokstav representeras med dess heltalsvärde i ASCII/Unicode tabellen! Det vill säga, bokstaven a representeras av siffran 97, etc.

Det finns smidiga funktioner för att konvertera en sträng till en lista och en lista med siffror till en atom:

```
?- string_to_list("anders",X).
X = [97, 110, 100, 101, 114, 115].

?- string_to_atom([97, 110, 100, 101, 114, 115],X).
X = anders.
```

11.4 Övningar

- Skriv om definitionen av `factorial` så att `false` returneras om `N` är negativt.
- Gör samma för `fib`.

- Skriv en funktion som summerar alla udda tal i en lista.
- Skriv en funktion som summerar alla tal på udda index i en lista.

Kapitel 12

Logikprogrammering i Prolog 3

12.1 Generera och testa

Vi har sett att Prolog kan användas för att räkna upp lösningar till problem. Detta kan även användas för att lösa problem. Till exempel kan vi sortera en lista genom att generera alla permutationer av den och välja den permutation som är sorterad. Detta sätt att programmera på kallas *generera och testa*.

Vi definierar ett predikat `permutation(X,Y)` som ska vara sant om `Y` är en permutation av `X`. Vi gör detta genom rekursion på `X` och utgår från följande logiska karaktärisering av permutation: Y är en permutation av X om huvudet på X finns i Y, och om vi tar bort det från Y då är den resulterade listan en permutation av svansen på X.

I Prolog skriver vi:

```
permutation([], []).
permutation([X | XS], YS) :-
  permutation(XS, XS_PERM),
  append(XS1, XS2, XS_PERM),
  append(XS1, [X | XS2], YS).
```

Tester:

```
?- permutation([1,2,3],X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Obs: Prolog använder *backtracking* för att generera alla lösningar. Internt har Prolog ett träd över alla val som gjorts och genom att traversa trädet och ändra på ett val får vi ny utdata. På detta sätt kan Prolog räkna upp alla lösningar till ett problem. Genom att använda "`;`" säger vi åt Prolog att backtracka och ge oss en ny lösning. Som vi ser ovan kommer Prolog backtracka ända tills vi uttömt alla fall och då får vi ut `false`. Om vi istället skriver "`.`" säger vi åt Prolog att vi är nöjda och den kommer inte backtracka mer.

```
?- permutation([1,2,3],X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] .
```

Vi kan definiera ett predikat för sorterade listor enligt:

```
sorted([]).
sorted([_]).
sorted([X, Y | L]) :-
  X =< Y,
  sorted([Y | L]).
```

Tester:

```
?- sorted([1,2,3]).
true .

?- sorted([1,3,2]).
false.
```

Vi har nu allt vi behöver för att definiera permutationssortering. Detta bygger på den matematiska definitionen av sortering: att sortera en lista kan definieras som att beräkna en sorterad permutation av den ursprungliga listan.

```
perm_sort(X, Y) :- permutation(X, Y), sorted(Y).
```

Tester:

```
?- perm_sort([4,3,2,1],X).
X = [1, 2, 3, 4] .

?- perm_sort([6,11,3,-4,22,45,99,-100],X).
X = [-100, -4, 3, 6, 11, 22, 45, 99] .
```

Detta är så klart väldigt ineffektivt. Delvis då det är ett väldigt naivt sätt att sortera på och för att Prolog inte inser att det bara finns en lösning och fortsätter gå igenom alla möjliga lösningar om vi skriver “;”. Om vi testar att köra

```
?- perm_sort([6,11,3,-4,22,45,99,-100,1,2],X).
X = [-100, -4, 1, 2, 3, 6, 11, 22, 45|...] ;
false.
```

så ser vi att det tar ett tag för Prolog att svara `false`. Det vore mycket bättre om vi kunde få Prolog att sluta när den hittat en lösning. För detta använder vi *snitt* vilket vi kommer gå igenom i nästa stycke.

Generera och lösa metoden är väldigt kraftfull och man kan på ett väldigt kompakt sätt skriva program som löser alla möjliga typer av problem. Exempelvis kan vi skriva en Sudoku-lösare genom att skriva ett predikat som testar om en Sudoku är löst och sedan generera alla möjliga sätt att sätta in värden i en Sudoku. Gör vi detta på ett smart sätt kan lösaren även bli ganska snabb.

12.2 Snitt

Vi har sett att Prolog bygger på backtracking för att hitta alla lösningar till problem. Detta är väldigt kraftfullt, men ibland kan det bli långsamt om man inte är försiktig då Prolog kan utforska möjligheter som inte leder någonstans. Vi kan kontrollera hur backtracking hanteras i Prolog med hjälp av *snitt* (eng. *cut*) som skrivs med "!".

Vi såg ovan att vi inte vill söka vidare när vi hittat den sorterade listan i `perm_sort`. Vi kan åstadkomma detta genom att lägga till ett snitt i slutet:

```
perm_sort(X, Y) :- permutation(X, Y), sorted(Y), !.
```

Test:

```
?- perm_sort([6,11,3,-4,22,45,99,-100,1,2],X).
X = [-100, -4, 1, 2, 3, 6, 11, 22, 45|...].
```

Vi får nu ingen möjlighet att trycka ; och vi kan alltså inte säga åt Prolog att backtracka. Snitt skär alltså bort backtrackingen för predikatet i vilkets kropp den förekommer och kan på så sätt användas för att förbättra prestandan hos predikatet.

12.2.1 `member`, med eller utan snitt

Här kommer en implementation av `member` som testar om ett element finns i en lista:

```
myMember(H, [H | _]).
myMember(X, [_ | T]) :- myMember(X, T).
```

Om vi kör följande får vi många svar:

```
?- myMember(2, [1,2,3,1,2,3]).
true ;
true ;
false.
```

För att eliminera svaren efter det första skriver vi:

```
myMemberCut(H, [H | _]) :- !.
myMemberCut(X, [_ | T]) :- myMemberCut(X, T).
```

Om vi kör testet från ovan får vi ingen möjlighet att få ut fler svar när vi har hittat den första tvåan:

```
?- myMemberCut(2, [1,2,3,1,2,3]).
true.
```

12.2.2 `intersecting`, med eller utan snitt

Vi kan testa om två listor överlappar varandra genom:

```
intersecting(XS, YS) :- member(X, XS), member(X, YS).
```

Om vi kör följande test får vi flera svar:

```
?- intersecting([1,2,3],[2,3,4]).
true ;
true ;
false
```

Precis som ovan kan vi använda ett snitt för att bara få ut ett svar:

```
intersecting(XS, YS) :- member(X, XS), member(X, YS), !.
```

Då får vi bara ett svar:

```
?- intersecting([1,2,3],[2,3,4]).
true.
```

Obs: om vi använder `myMemberCut` i `intersecting` blir det knas:

```
myMemberCut(H, [H | _]) :- !.
myMemberCut(X, [_ | T]) :- myMemberCut(X, T).

intersecting(XS, YS) :- myMemberCut(X, XS), myMemberCut(X, YS), !.
```

Test:

```
?- intersecting([1,2,3],[2,3,4]).
false.
```

Problemet är att då vi har ett snitt i `myMemberCut` kommer Prolog inte försöka backtracka efter att den testat om 1 finns i båda listorna. Så man måste vara försöktig när man använder `!`. Notera även att det inbyggda predikatet `member` funkar på följande sätt:

```
?- member(2,[1,2,2,3,2]).
true ;
true ;
true.
```

Det använder alltså snitt, men inte på samma sätt som i `myMemberCut` utan på följande sätt:

```
myMember(X, [X]) :- !.
myMember(H, [H | _]).
myMember(X, [_ | T]) :- myMember(X, T).
```

Test:

```
?- myMember(2,[1,2,2,3,2]).
true ;
true ;
true.
```

12.2.3 När behövs snitt?

Programmet nedan testar vilken av `X` eller `Y` som är störst och lägger resultatet till `Z`:

```
max(X, Y, Z) :- X > Y, Z = X.
max(_, Y, Z) :- Z = Y.
```

Men programmet är buggigt:

```
?- max(4,3,3).
true.
```

Problemet är att Prolog hoppar vidare till andra fallet då första misslyckas. Rättad implementation:

```
max(X, Y, Z) :- X > Y, !, Z = X.
max(_, Y, Z) :- Z = Y.
```

Tester:

```
?- max(4,3,3).
false.

?- max(4,3,4).
true.
```

Varför funkade detta? Om `X > Y` då kommer Prolog inte försöka med nästa rad eftersom det kommer ett snitt.

12.2.4 Mer on snitt

Här kommer ett litet påhittat program utan snitt:

```
s(X,Y) :- q(X,Y).
s(0,0).

q(X,Y):- i(X), j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```

Genom att köra `s(X,Y)` ser vi hur Prolog går igenom alla lösningar

```
?- s(X,Y).
X = Y, Y = 1 ;
X = 1, Y = 2 ;
X = 1, Y = 3 ;
X = 2, Y = 1 ;
X = Y, Y = 2 ;
X = 2, Y = 3 ;
X = Y, Y = 0.
```

Obs: jag har formatterat utdata lite för att göra det mer lättläst.

De olika lösningarna är alltså:

```
s(1,1)
s(1,2)
s(1,3)
s(2,1)
s(2,2)
s(2,3)
s(0,0)
```

Låt oss säga att vi nu lägger till ett snitt mitt i `q`:

```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), !, j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```

Då får vi följande utdata:

```
?- s(X,Y).
X = Y, Y = 1 ;
X = 1, Y = 2 ;
X = 1, Y = 3 ;
X = Y, Y = 0.
```

Vilket motsvarar:

```
s(1,1)
s(1,2)
s(1,3)
s(0,0)
```

Vi ser hur Prolog begränsar sin backtracking på grund av snittet. När Prolog kommer till ett snitt så kommer alla val den har gjort att sparas och ej göras om. I detta fall kommer vi ha valt `X = 1` när vi kommer till snittet och Prolog kommer därför aldrig försöka med `X = 2` eller `X = 3`. Detta betyder att snitt kan användas för att begränsa sökmängden och alltså snabba upp program.

12.3 Negation

Prolog har även negation, detta skrivs med `\+`.

```
even(X) :- X mod 2 =:= 0.

odd(X) :- \+ even(X).
```

```
?- even(4).
true.

?- odd(4).
false.

?- odd(5).
true.
```

Men som vi pratat om tidigare arbetar Prolog utifrån ett antagande om en sluten värld, så allt som Prolog inte vet något om hanterar den som falskt. Detta leder till att om vi skriver:

```
man(erik).
man(olle).

woman(X) :- \+ man(X).
```

så anser Prolog att:

```
?- woman(anders).
true.

?- woman(42).
true.

?- woman(car).
true.
```

Negationer kan vara användbara. Låt oss säga att vi vill skriva ett predikat som testar om två listor inte delar några element. Då kan vi ta negationen av `intersecting` från ovan:

```
disjoint(XS, YS) :- \+ intersecting(XS,YS).
```

Det här fungerar som vi tänkt oss:

```
?- disjoint([1,2,3],[3,4,5]).
false.

?- disjoint([1,2,3],[4,5,6]).
true.
```

Varning: Prolog är inte tillräckligt smart för att alltid kunna räkna upp saker om man använder negation. Låt oss säga att vi har följande predikat:

```
nonMember(X,XS) :- \+ member(X,XS).
```

Då händer följande:

```
?- nonMember(X,[1,3]).
false.
```

även om det finns massor med saker som inte finns i listan:

```
?- nonMember(2,[1,3]).
true.
```

Så var försiktiga om ni använder negation.

12.4 Unifikation

I första föreläsnigen om Prolog såg vi att det finns flera olika sätt att jämföra saker i Prolog och att `X = Y` används för att *unifiera* `X` och `Y`. Vi ska nu prata lite mer om vad detta betyder.

Som vi såg i första föreläsningen används = för att hitta lösningar till ekvationer med variabler men att inget evalueras när man gör det:

```
?- X = 3 + 2.
X = 3+2.
```

```
?- 3 + 2 = X.
X = 3+2.

?- X = Y * Y.
X = Y*Y.

?- X * X = Y * Y.
X = Y.

?- X + Y = Z + W.
X = Z,
Y = W.

?- 2 * X = Y * 2.
X = Y, Y = 2.
```

Prolog klarar även av att unifiera mer komplexa uttryck:

```
?- k(s(g),Y) = k(X,t(k)).
Y = t(k),
X = s(g).

?- k(s(g),t(Y)) = k(X,t(k)).
Y = k,
X = s(g).
```

Men om vi skriver in följande händer något oväntat:

```
?- p(X) = X.
X = p(X).
```

Vi har fått Prolog att acceptera en oändlig definition. Exempelvis om `p` vore `s` (successor) från labben så är `X` ett tal så att `s(X) = X`. Enda möjligheten för detta är att `X = s(s(s(...)))` i all oändlighet. Så varför accepterar Prolog detta?

Anledningen är att Prolog implementerar en lite speciell unifikationsalgoritm som tar en genväg för att vara snabbare. Det Prologs `=` gör är att den skippar ett viktigt test som heter *occurs check*. Detta test går till så att om man unifierar en variabel med en mer komplex term så kollar man först att variabeln inte finns i den komplexa termen. Om variabeln gör det så försöker man inte unifiera uttrycken och returnerar `false`. I fallet ovan så skulle `p(X) = X` ej uppfylla detta test då `X` finns i `p(X)` och occurs check skulle alltså bli `false` och termerna unifierar inte. Detta enkla test används för att undvika oändliga definitioner som den ovan, men Prologs `=` struntar som sagt i det för att vara snabbare.

Det finns fall när man vill göra detta test och som tur är har Prolog ett inbyggt predikat för att unifiera två uttryck med hjälp av occurs check, `unify_with_occurs_check`: https://www.swi-prolog.org/pldoc/man?predicate=unify_with_occurs_check/2

Detta predikat fungerar precis som `=`, bortsett från att occurs check alltid görs och oändliga uttryck som den ovan ej accepteras:

```
?- unify_with_occurs_check(k(s(g),t(Y)),k(X,t(k))).
Y = k,
```

```
X = s(g).

?- unify_with_occurs_check(k(s(g),Y),k(X,t(k))).
Y = t(k),
X = s(g).

?- unify_with_occurs_check(p(X),X).
false.

?- unify_with_occurs_check(k(s(X),Y),k(X,t(k))).
false.

?- k(s(X),Y) = k(X,t(k)).
X = s(X),
Y = t(k).
```

Obs: några av er kanske har fått ett felmeddelande i Haskell som sa något i stil med *Occurs check: cannot construct the infinite type: ...*. Nu vet ni vad det handlade om!