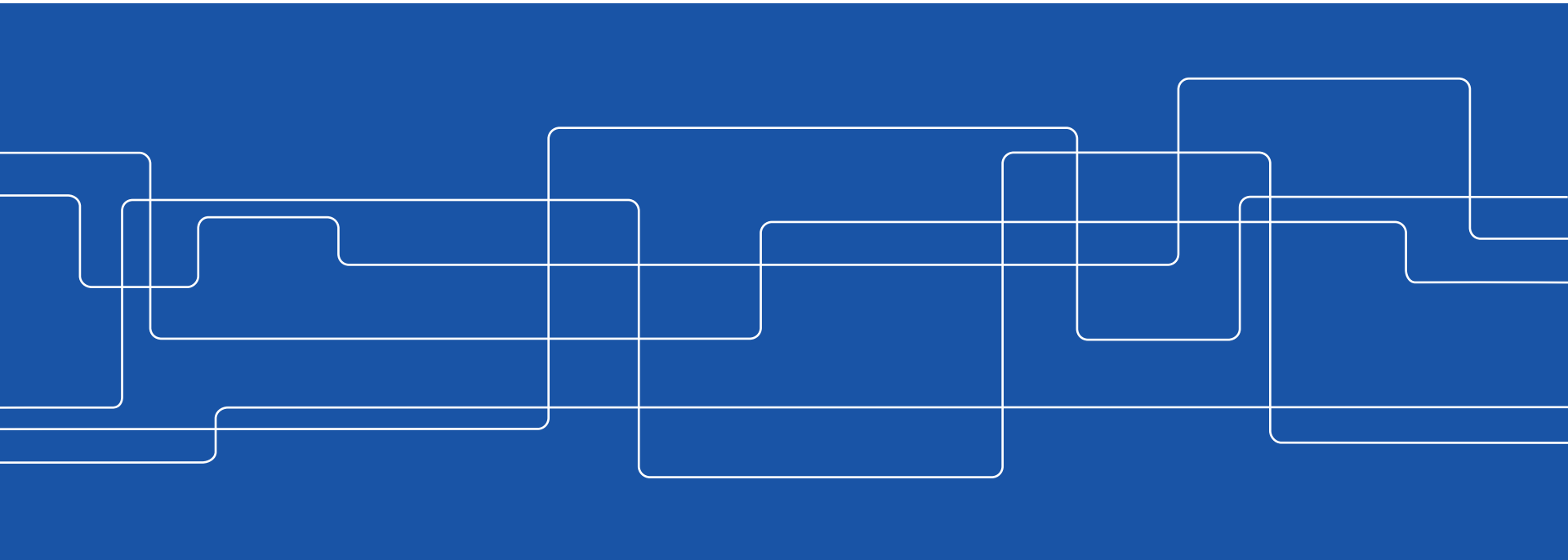
Föreläsning 2

Operativsystem och programmering



Behov av operativsystem

En dator så som beskriven i förra föreläsningen är nästan oanvändbar. Processorn kan bara ges enkla instruktioner såsom hämta data på den platsen i minnet, lägg ihop datat tolkat som ett heltal med datat i det registret osv. Dessutom skall dessa instruktioner ges som en räcka ettor och nollor.

Vi behöver program som kan tolka våra instruktioner som beskriver det vi vill göra (ex visa innehållet i en textfil på skärmen, skriva ut det på en skrivare, spela upp ett ljud,…) och översätta det till instruktioner till processorn.

Detta görs av en mängd program i flera nivåer från våra krav ner till ettor och nollor som styrenheten kan tolka till instruktioner. Det yttersta lagret som vi som användare upplever som datorn kallas för ett operativsystem

Operativsystemet måste vara anpassat efter datorns arkitektur och processorns instruktioner.

Operativsystemets funktioner

Ett operativsystem skall dock inte erbjuda allt vi vill göra med datorn utan oftast använder vi ett specifikt program för att utföra en specifik uppgift. Operativsystemets uppgift är då den miljö programmet körs i. Det tillhandahåller funktioner som programmet kan anropa för att få saker gjorda (läsa filer, använda portar, kommunicera med andra program, visa grafik, …), det övervakar att programmet inte använder för mycket av datorns resurser och att det inte förstör operativsystemets eller andra programs funktion.

Operativsystemet och hårdvaran blir tillsammans en virtuell maskin som faktiskt går att använda både för användare och programmerare.

Exakt vilka program som ska räknas till ett operativsystem är inte helt klart. Man kan dela in ett operativsystem i tre delar:

- kärnan - hanterar datorns resurser såsom processortid, minnesaccess,…
- användarprogram – basprogram såsom filhantering, konfigurering, texteditor,…
- systembibliotek – tillhandahåller grundläggande funktionalitet till övriga program

Exempel:

Windows, Linux, MacOs, Unix, VxWorks

BIOS

Så med ett operativsystem blir vår dator en användbar maskin. Problemet är att när vi slår på datorn är den för dum för att kunna läsa in operativsystemet från hårddisken.

Det första som händer när en PC slås på är att ett enklare mer grundläggande program kallat BIOS laddas in och körs (maskinen bootar). BIOS ger datorn bara grundläggande funktionalitet, bland annat så att den kan ladda in och köra operativsystemet.

BIOS måste vara skrivet för en viss datorkonfiguration (processor, moderkortsarkitektur). Förr låg det på ROM men idag används flash.

BIOS kan användas av operativsystemet som ett mellanlager för att utnyttja hårdvara. Det används dock normalt inte för detta av dagens operativsystem.

Maskin- och assemblerkod

Ett program som processorn ska förstå måste vara giltiga instruktioner på binär form, dvs en räcka ettor och nollor. Riktigt så primitivt kan man inte programmera men de första programmen skrevs i det man kallar maskinkod. Man anger varje instruktion till processorn med en operationskod normalt på hexadecimal form. Tal anges också i hexadecimal from. Istället för variabler anger man minnesadressen där ett visst värde ligger (såklart i hexadecimal form).

För att möjliggöra skapandet av större program skapades assemblerspråk. Ett assemblerspråk är unikt för en viss dator (datorarkitektur) eftersom det innehåller instruktioner till processorn. Men här har varje instruktion fått ett namn, tal kan skrivas i decimal form, vi har variabler och även aritmetiska uttryck kan skrivas (beräknas dock i assembleringen inte i exekveringen). Hoppsatser behöver inte heller hoppa till en adress utan kan anges med ett namn.

Idag har vi högnivåspråk som tillåter en mycket större abstraktion som vi kompilerar till maskinkod. Fortfarande är det ibland framför allt av effektivitetsskäl och säkerhetsskäl nödvändigt att programmera direkt i assembler. Det gäller då ofta mindre processorer i inbyggda system eller hårdvarunära rutiner i operativsystemet eller på grafikkortet. Vi kan då med assembler få full kontroll exakt på vad processorn gör i varje klockcykel. Något som inte går i högnivåspråk. Antalet tillfällen då vi tvingas använda assembler minskar hela tiden vartefter processorer blir kraftfullare och ersätts då ofta av C-programmering som är ett högnivåspråk med ganska låg lägstanivå (hög kontroll över hårdvara).

(assemblerkod kallas ofta maskinkod)

Högnivåspråk

Högnivåspråk utvecklades för att möjliggöra skrivandet av större mera komplexa program och lösandet av större och mera komplexa problem med datorer. Till skillnad från assembler tar de inte avstamp i vad hårdvaran kan göra utan är främst skapade utifrån vilka verktyg vi behöver för att kunna lösa problem. Dessa program måste då sedan översättas till maskinkod innan de kan köras.

Exempel på högnivåspråk är Fortran, C, C++, C#, Java, Javascript, Swift

Gemensamt för dessa och de flesta högnivåspråk är möjligheten att kunna skriva läsbar och lättförståelig kod med variabler, villkor, upprepningar, matematiska uttryck, funktioner osv.

Syntax och semantik

Definitionen av ett programspråk kan grovt delas upp i två delar:

- Syntaxen: definierar formen hos språket och bestämmer hur man får skriva. Ett programspråks syntax beskriver hur formellt korrekta program får formuleras.

- Semantiken: definierar vad som händer när man skrivit en viss sak. Ett programspråks semantik beskriver vad som krävs för att ett formellt korrekt program ska vara meningsfullt och vad som händer när det körs.

Syntaxen definieras formellt och exakt medans semantiken oftast beskrivs informellt i våra naturliga språk. I programmering krävs att syntaxen följs exakt. Minsta avvikelse gör programmet okörbart.

Kompilerande - interpreterande

I ett kompilerande språk översätter en kompilator (ett program) textfilen som innehåller programmet till (maskinkod), dvs en körbar (exekverbar) fil. Denna fil kan nu endast köras på en viss sorts dator. Vill man köra programmet på en annan dator måste koden kompileras till denna. Kompileringen görs ofta i fler steg av flera program (preprocessor, kompilator, länkare).

I ett interpreterande språk körs textfilen i en interpreterare som översätter koden dynamiskt varje gång programmet körs.

Samma språk kan ha både kompilatorer och interpreterare. Exempel på typiskt kompilerande språk är C och Fortran. Exempel på interpreterande är Basic och Javascript. Java är lite av ett mellanläge. Koden kompileras till maskinkod för en virtuell maskin som inte finns. När man sedan ska köra koden körs den på denna virtuella maskin (ett program) som då agerar interpreterare. Dessutom har man också börjat med just-in-time compiling.

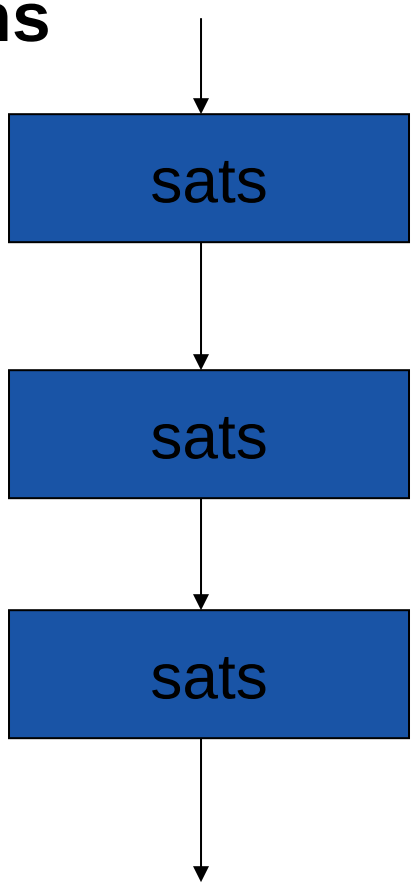
Algoritmer

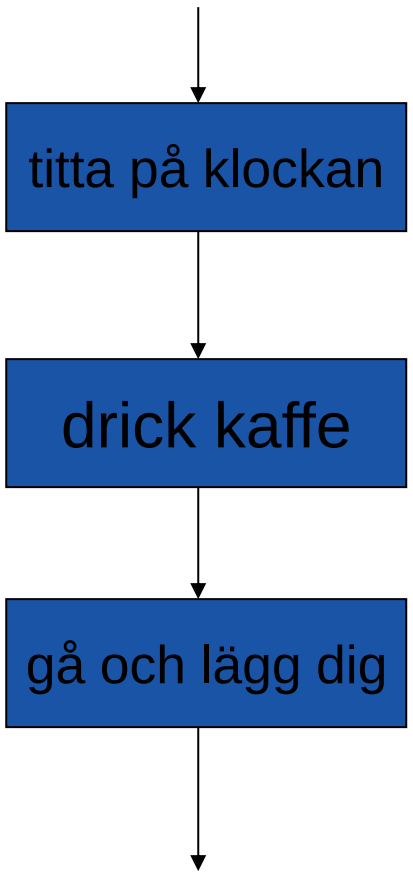
En algoritm är en metod för att lösa ett problem genom att utföra ett antal elementära operationer (en idiot ska förstå) i en av algoritmen given ordning. Metoden skall efter ett ändligt antal operationer nå lösningen.

Det är en instruktion som man utan att tänka ska kunna följa steg för steg och då få fram resultatet. Låter som gjort för en dator (en idiot ☺).

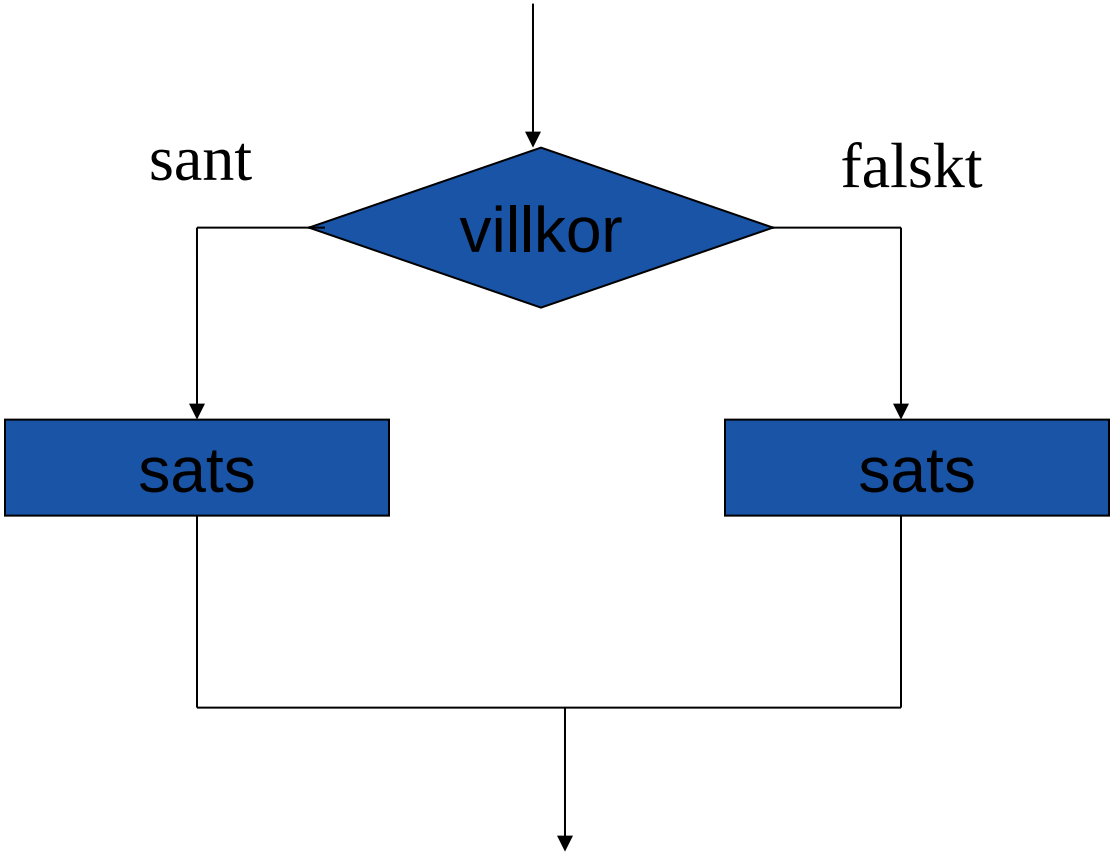
För att vi för en dator ska kunna beskriva i vilken ordning den ska utföra de olika stegen är det vissa konstruktioner som återfinns i alla programmeringsspråk. Lab 1 låter er öva på dessa.

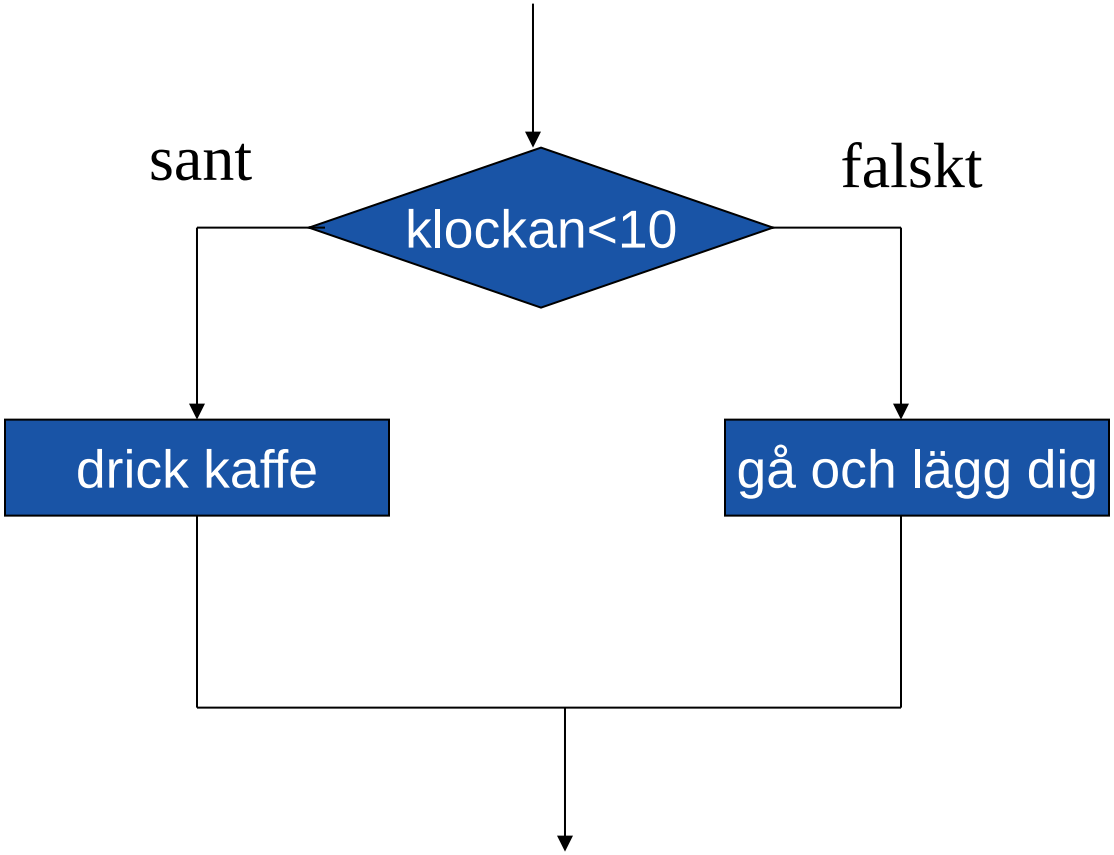
Sekvens





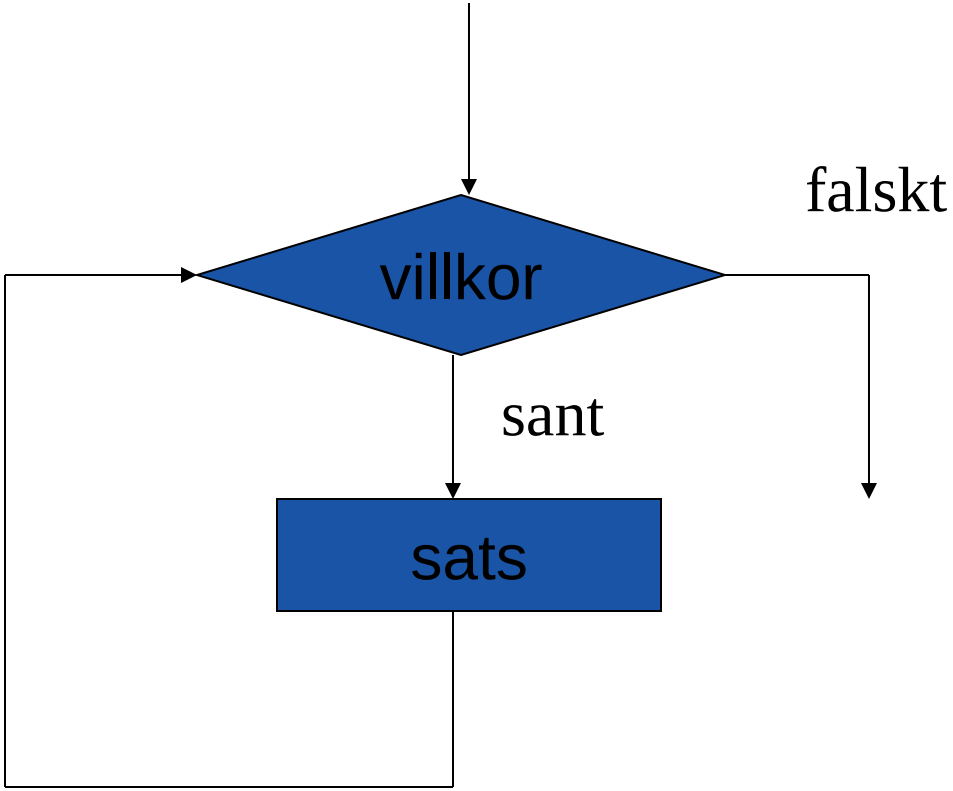
Selektion (Om…annars…)

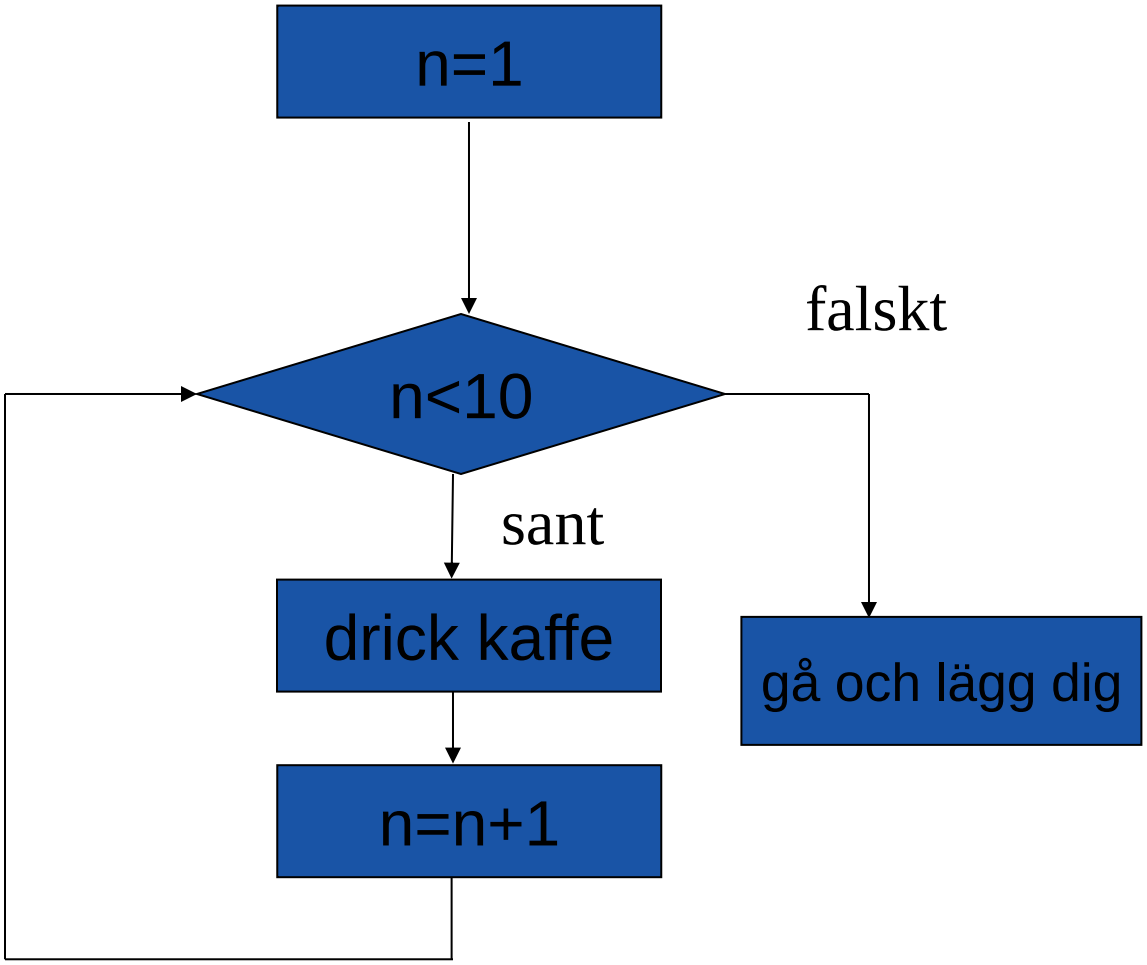




Om klockan är mindre än 10 drick kaffe annars gå och lägg dig

Iteration (Upprepa)




n=1
n<10
falskt
sant
drick kaffe
gå och lägg dig
n=n+1

Javascript och HTML

Efter att ni är klara med laboration 1 och grafisk blockprogrammering ska vi pröva lite riktig programmering. Vi ska då använda Javascript vilket är ett interpreterande scriptspråk med ganska fri syntax vilket kan vara både bra och dåligt för en nybörjare. Anledningen att vi tar våra första steg som programmerare i detta språk är att det är lätt att programmera grafik och lätt att hantera grafiska användargränssnitt. Syntaxen är också mycket lik C som vi senare ska programmera i. Javascript exekveras normalt i en webläsare inbäddat i ett html-dokument.

Ett html-dokument är text med taggar som beskriver hur texten ska visas och används för dokument som ska visas i en webläsare. Ett väldigt enkelt dokument kan se ut så här:

```
<html>
        <head><title> Test </title></head>
        <body>
                <p>Hej</p>
        </body>
</html>
```

Vi kan nu enkelt lägga in ett Javascript på vår sida med taggen <script>:

```
<html>
        <head><title> Test </title></head>
        <body>
                <p>Hej</p>
                <script>
                        document.write("Hello, World!");
                        alert(Date());
                </script>
        </body>
</html>
```

Lab 2

I denna laboration ska ni följa fem tutorials på youtube för att gå från att lära er programmera till att göra ert första spel. Programmet skriver ni direkt i browsern och resultatet ser ni bredvid.

Öppna ett browserfönster med:
http://spelprogrammering.nu/koda och välj: skapa nytt

Titta på Youtube-filmerna i ett annat fönster:
https://www.youtube.com/user/coderaptors

Gör tutorial 1-5.

När ni är klara med tutorial 5 visar ni resultatet för lärare eller assistent och vips är ni klara med laboration 2. Gör gärna resten också!

Inför nästa gång

Nästa gång ska vi börja med C-programmering. Då kommer vi behöva ladda ner en programmeringsmiljö till datorn. För att inte alla ska försöka det samtidigt föreslår jag att ni alla gör det innan nästa gång.

Gå in på http://www.codeblocks.org/downloads/26

Välj att ladda ner codeblocks med kompilator (mingw):

codeblocks-16.01mingw-setup.exe för windows.

Kör filen.