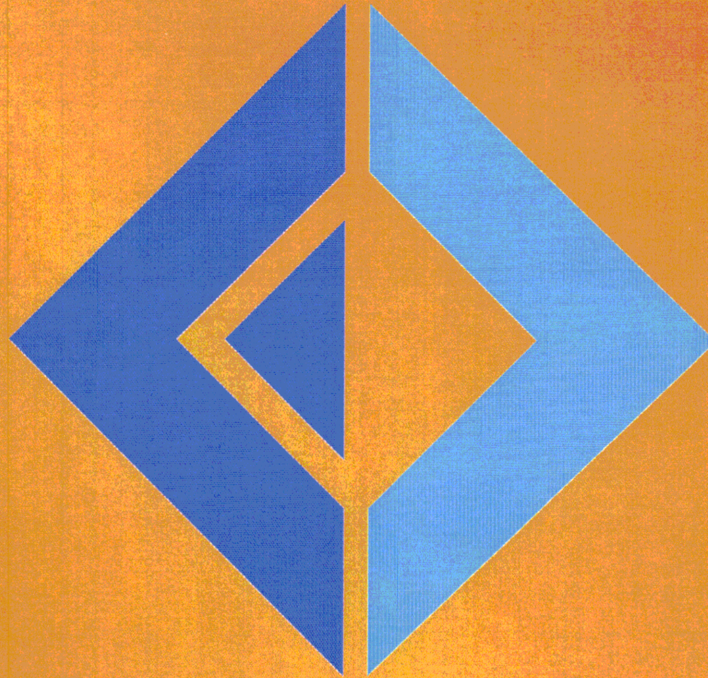
DKuug nr. 176

NYT

November-December 2014



Dansk Forum for Åbne Systemer
DKUUG - Unix Brugere (Linux/BSD) system administratorer
Community for IT-specialister og IT-interesserede.

Mono, dotNET, F-sharp og virtuelle maskiner side 4
F# side 6
Shell Shock/chok side 16
Kommandoliniecentralen side 19
- og mere

DKUUG

Kære Læser

Begreberne flyver omkring -Viden fordi det er sjovt - men det er også nyttigt

Under arbejdet med at finde dokumentation for, hvad der er de vigtigste egenskaber ved Mono og .NET frameworket, blev det klart at det var nødvendigt med kendskab til CPU-arkitektur og maskin-instruktioners opbygning for bare at forstå sætningerne i teksten. Her i redaktionen ville vi gerne have givet dig, kære læser, endnu mere information på en let overskuelig måde om virtuelle maskiner, et begreb, der ofte nævnes i forbindelse med .NET programmer (apps) som en slags trylleformular for, hvordan man kan gøre noget bedre.

I dette tilfælde er "bedre" en omskrivning af "så det fungerer ordentligt". Virtuelle maskiner er en måde at lave ét program én gang og bruge mange gange: et program, som kender den hardware, det kører på, og lader alle andre programmer udnytte denne viden.

Der er forskellige måder at bruge det princip - man kan udnytte det til at lave virtuelle hosts, typisk en webserver i en serverfarm hos et hosting-firma, som kører på samme hardware som en stribe andre webservere.

En anden Virtuel Maskine (VM) kendes fra Java. SUN, som "opfandt" Java, havde tænkt sig engang at komme med en silicium-implementation af JavaVM - men det blev vist aldrig til noget. Af flere forskelllige grunde er Intels CPUer bedre og billigere end de fleste alternativer - nåja, bortset fra AMD, og Intel har jo også overtaget ARM processoren, som ikke ligefrem var et Intel design ... sådan er det så meget.

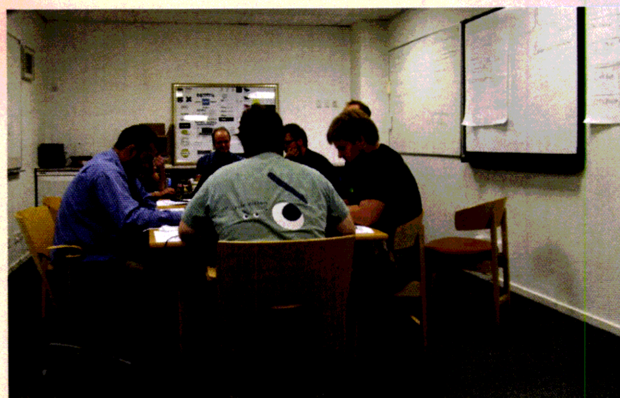
I et kommende nummer vil DNyt kaste sig over fænomenet Virtuelle Maskiner, men også vise de alternativer, som har eksisteret på Unix siden kommandoen ***chroot*** blev "opfundet".

Problemet er, at alle IT systemer i dag er mere komplicerede end noget som helst menneskeheden tidligere har set.

Poul Henning Kamp, som vi ynder at citere, nævner som en kendsgerning de fleste ikke bryder sig om at tænke på, at vi har expanderet software-komplexiteten i en grad, så vi ikke kan overskue konsekvenserne. Kamp siger (citeret efter hukommelsen): Den lille mobil-telefon, som vi har i lommen - selv de mindste og billigste - har samme komplexitet som 12 hangarskibe. Udbredelsen af iPhones og iPads er ikke nogen garanti for, at vi er et foregansland i IT-henseende, siger han også. Hør mere på Radio24syv.dk - man kan downloade programmet Aflyttet uge 44, gå fx. 36 minutter ind og hør Poul Henning Kamp fortælle om CSC og komplexitet.

Allerede Joseph Weizenbaum gjorde opmærksom på de ofte oversete konsekvenser af computersystemers stigende kompleksitet. Hans mest slående eksempel (fremsat i bogen *Computer Power And Human Reason*, Cambridge 1975) går på at de mange bankers børshandels-systemer er en tidsindstillet bombe under økonomien - fordi, som han siger, de udgør en automatisk deterministisk (men uoverskuelig) maskine, som ved et kursfald vil få alle til at sælge og derved udløser endnu mere kursfald.

- Altså, det samme, som udløste børskrak i 1929 og igen i 2008, som dog udløstes af helt andre årsager end børskrakket i 1929. Noget har man lært: IT systemerne lukker ned, hvis kursfald er for voldsomme og uforudsigelige, men afhængighederne fra den ene globale spiller til de andre har vi ikke fået under kontrol.



Der arbejdes i vores kontor

Dette nummer ...

F# - F sharp - har fået en fremtrædende plads i dette nummer, sammen med baggrundsstof om Mono - Open Source impelementering af .NET - selv om emnerne ikke er nyheder. Det nærmeste, vi kan komme, er at det er en langsom revolution, som foregår mens vi ser på det.

Begrundelsen for at vælge dette emne er, at funktions-orienteret programmering, funktions-programmering (FP) for kortheds skyld, er en sag som slår mange professionelle med forundring og ubehag, det er noget underligt noget, som man er lidt valen overfor. Også selv om man godt ved, at dette at lære nye programmeringssprog er gavnligt, omend man ikke bruger det. Lidt ligesom at det er en fordel at kende lidt til Latin. Vi har en masse latinske låneord. Men Paul Graham har en lidt anden og mere interessant indfaldsvinkel på funktionel-stil. I en efterhånden 13-14 år gammel artikel, som man kan finde på nettet. Hans artikel handler ikke om F#, som ikke fandtes på det tidspunkt, men om Lisp, som også er et funktions-sprog (og som vi har omtalt i et tidligere nummer).

Der er ingen, der taler latin, skriver Paul Graham, men der er nogen, der "taler" Lisp. Hvis det kan give dig en konkurrence-fordel på markedet, så brug det - ellers lad det ligge.

Graham startede sammen med en ven firmaet Viaweb, som tilbød udvikling og hosting af webshops. Software er en meget konkurrencebetonet branche, skriver han, en branche, som ofte ender med at den bedste har en slags monopol. Når man starter et firma op, mærker man dette meget tydeligt, det tenderer mod alt eller intet.

Så drastisk vil vi nu ikke formulere begrundelserne for at kende flere sprog. I en helt anden boldgade fandt vi flg. citat om programmering: Hvad der adskiller Funktionel Programmering (FP) fra andre sprog er den fiksering på variable, som ikke kan variere! Jeg har aldrig skrevet et program, som ikke indeholdt "immutable" (uforanderlige) objekter, men vi kalder dem altid bare ***konstanter. [...]*** Jeg kan ikke forestille mig at skrive noget komplekst uden variable og konstanter. Men at skrive et enormt antal 3-liniers funktioner for at undgå en simpel variabel er ikke et fremskridt, skriver David Clarkd, BSc, RCC consulting.

Det skal dog understreges, at David Clarkd finder mange nyttige ideer i funktionel programmering, ligesom det skal understreges, at Paul Graham gik fra Lisp til C, da ordredelen i web-shop systemet skulle udvides.

Vi håber at læserne får en god oplevelse med bladets eksempler på F# programmering

Donald Axel

DKUUG

DKuug-NYT er medlemsblad for DKuug, foreningen for Åbne Systemer og Internet
Nr. 176 - Novmber 2014

Udgiver:
DKUUG
Fruebjergvej 3
2100 København Ø
Tlf. 39 17 99 44
email: blad@dkuug.dk
Redaktion:
Donald Axel (ansvarshavende)

Forsidecredits:
redaktionen, logo fra fsharp.org

Design og layout:
DKUUG/Donald Axel med LibreOffice
Annoncer:
pr@dkuug.dk
Tryk:
Lasertryk i Aarhus
Oplag:
400 eksemplarer

Artikler og inlæg i DKUUG-Nyt er ikke nødvendigvis i overensstemmelse med redaktionens eller DKUUGs bestyrelses synspunkter.

Eftertryk i uddrag med kildeangivelse er tilladt.

Deadline for nr. 177: ons. 14. januar 2015

Medlem af Dansk Fagpresse
DKUUG-Nyt
ISSN-1395-1440

Vores møder og **foredrag** holdes - med mindre andet udtrykkeligt angives - på vores adresse:

DKUUG
SYMBION
Fruebjergvej 3
2100 København Ø

Hvis man kommer lidt før, er der tid til en snak på kontoret. DKUUG bor i en virksomhedsfarm, Symbion, hvor der er åbne døre indtil kl.18 eller 19 (afhængig af mødetidspunkt). Efter den tid har vi på foredragsaftener en vagt ved døren.

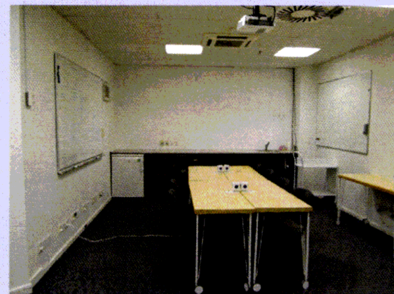
INDHOLD:

Designfilosofi bag Mono - en platform for applikationer
af Donald Axel 4
F# eller F sharp
af David Askirk 6
Klip om Open Data, Open Knowledge 15
Shell Shock - chok 16
Kommandocentralen 19

Arrangementer:

Onsdag d. 26. November: Generalforsamling.

Andre arrangementer vil blive annonceret på web og via mail.



Gør-det-selv foredrag:

Vores kontor i Symbion giver mulighed for hands-on workshops, både dag og aften. Skriv til pr@dkuug.dk eller til bestyrelsen i DKUUG, bestyr@dkuug.dk, og hør om lokalet er ledigt den dag du vil arrangere et møde. Der er hurtig internetforbindelse, både wired og wireless. Er der større tilmelding, kan vi leje mødelokaler i Symbions mødested. Spørg os!

Designfilosofi bag *Mono* - en platform for applikationer

Hvad ønsker en applikationsprogrammør? og hvordan prøver Mono på at gøre det lettere at udvikle applikationer?

Af Donald Axel

Det erklærede mål bag *Mono* er at være en platform (og en udviklingsplatform, *monodevelop*) som gør det lettere og hurtigere at udvikle en applikation. Men hvilket programmeringssprog har ikke det mål? Det særlige ved Mono er, at de resulterende programmer skal kunne udnytte Internettet så let som muligt - plus et par ting mere, såsom at kunne køre på flere typer hardware og operativ-systemer, bruge touchscreens mv.

Miguel de Icaza sagde ved et foredrag i 2010 om den fortsatte udvikling af Mono *(Youtube video Mono Edge, 26 min.inde)* :

> *Innovation er at få markedsført idéer. Man får masser af idéer når man tager brusebad eller går på arbejde. Men innovation er at få det til at ske, få det ud til forbrugerne.*
>
> *Det, som vi ønskede at opnå med Mono for mange år siden, var at gøre programmører mere produktive. Hvordan kunne vi hjælpe programmører med at komme fra idé til marked, - til et produkt, der kan markedsføres? Jeg ved ikke om vi har svaret på det, men jeg vil fortælle jer hvad vores intentioner er, fortælle om basics og hvordan vi kan hjælpe jer med at komme dertil.*

Det har Icaza gentaget ved alle senere konferencer. Han uddyber: At lave et produkt består i at tage et program, rette fejlene og skrive dokumentationen, hvilket som regel tager 3 gange så lang tid som at skrive programmet.

[Ingen blandt publikum protesterer ... Her på redaktionen var der en protest; der findes udviklingsmiljøer, som arbejder i omvendt rækkefølge, først dokumentation, så kode!]

Icaza fortsætter:

> *Hvis man laver en fremragende feature, men ikke skriver dokumentation, kan man lige så godt lade være med at skrive koden, for ingen vil opdage, at den feature findes.*
>
> *For at lave et godt produkt skal man følge konventioner, dokumentere, arbejde sammen, bruge aftalt API (systemkald og kald til hjælpefunktioner, Application Programming Interface).*

Det er også gammelkendte sandheder: standard-libraries - C++ libraries er udviklet og justeret gennem de sidste 20 år.

Der findes andre applikations-miljøer, som har gjort det let at udvikle og dokumentere API'er, moduler, afhængigheder. C, C++ og Java kan køre på mange devices.

Der har været multiplatform miljøer siden 1978, da University of California, San Diego (UCSD) Institute for Information Systems udviklede UCSD Pascal for at de studerende skulle kunne arbejde i et kendt miljø på de mange forskellige typer PC'er, som den gang eksisterede, såvel som på universitetes minicomputer. Det system blev kendt som UCSD p-System.

Men så må der i det mindste være noget nyt ved Mono?

Nu vil vi have den tekniske forklaring, hvis der er en, og ikke blot stille os tilfredse med at Mono svarer til .NET for Linux og Android!

Factbox om Mono

Mono er et Open Source projekt, som styres af Xamarin, tidligere af Novell, og oprindelig af Ximian, gående ud på at skabe ECMA standard-compliant .NET framework-kompatibelt sæt af udviklingsværktøjer, med blandt andet en C# compiler, og en Common Language Runtime, **CLR**, en virtuel maskine, som er specielt designet til at afvikle applikationer (Se nærmere i factbox side 17).

Men hvad går Mono ud på?

Men Mono har ambitioner om at være mere end .NET-for-Linux/Android - det er også for iPhone med Xamarin proprietary library.

Mono kører i dag på Android, BSD, Windows, Solaris, og endda game-konsoller som Playstation 3, Wii, og Xbox-360.

Mono-logoet er en stiliseret abe - mono er spansk for abe.

Svaret kan koges ned til ét ord, er: CIL. (-er det ikke et ord? Nå så tre ord da!) ***Common Intermediate Language.*** Et intermediært sprog er et mellemled mellem programmørens sprog og de maskininstruktioner, som CPU'en ender med at læse, afkode og udføre.

Vi kommer ikke uden om en vis nødvendig grundviden her, men et billede er bedre end mange ord: Et lille F# program (ikke særlig nyttigt) oversættes, først til .exe fil, så kigger vi på den og derefter genererer vi x86 assembler ud fra .exe filen med ***mono***. Kommando-sekvensen

```
// Program som skal give genkendelig AOT code (Ahead Of Time compilation).

// Parenteser om arg til printf-format string er obligatorisk.

let myfirstval = 26
let mysecondval = 7

let divider myv1 myv2 =
    printfn "Division med heltal giver heltal, 26/7: %d" (myv1/myv2)


divider myfirstval mysecondval
```

Her ses et minimalt F# program; den sidste linie sætter programmet igang ved at kalde funktionen "divider"

⬇

```
fsharpc daxdivide.fs -o daxdivide.exe
monodis daxdivide.exe > daxdivide.cil
```

```
    // method line 4
    .method public static
           default void main@ ()  cil managed
    {
        // Method begins at RVA 0x2084
        .entrypoint
        // Code size 9 (0x9)                    0x1a = 16 + 10 => 26
        .maxstack 8
        IL_0000:  ldc.i4.s 0x1a
        IL_0002:  ldc.i4.7
        IL_0003:  call void class Daxdivide::divider(int32, int32)
        IL_0008:  ret
    } // end of method $Daxdivide::main@

  } // end of class <StartupCode$daxdivide>.$Daxdivide
}
```

Common Intermediate Language - kun den sidste linie af programmet, `divider myfirstval mysecondval` ***som omsættes til et funktionskald, call void class Daxdivide::divider ...***

`ldc.i4` betyder load en 4 bytes integer (parametret til vores kald af funktionen) med værdien 0x1a (hexadecimal for 26) - det er *myfirstval* som vi satte til 26, den er her blevet *optimeret* til en konstant.

⬇ →

Når man i *mono* miljøet laver en programfil, er navnet på sourcefilen en del af identifikationen af objekterne i filen. man skal huske at alle mulige programmeringssprog kan bidrage til et større projekt. Hvis ikke man brugte filnavnet, kunne man risikere nameclash - at en identifier kom til at hedde det samme som en anden variabel (eller F# funktion mv.) og derved ændrede programmet ved at skygge for den oprindelige ting med det navn. Derfor ses i disassembleringen (CIL-listningen) af divider-programmet, at ordet "Daxdivide" har sneget sig ind. Det var filnavnet på det lille demonstrationsprogram og har ikke nogen dybere mening, bortset fra at det er nemt at finde egne programmer, hvis man prepender dem med sine initialer.

Næste trin er at generere native code, assembler-instruktioner til den CPU, som programmet kører på. Programmet *mono* (som egentlig er et symbolsk link, et "shortcut" om man vil, til ***mono-sgen***) oversætter **A**head **O**f **T**ime, **AOT** (i forvejen).

Det er en af de særlige fordele ved Mono at den ***i forvejen*** kan generere native code, maskinkode til den maskine, man ønsker at køre på. Det er ikke en måde at deploye til markedet, eftersom der er så mange maskin-forskelle, man lokalt er nødt til at tage hensyn til. Optimering af koden spørger den aktuelle CPU, hvilke instruktionsgrupper, den har, for at sikre sig at kunne optimere mest muligt.

```
$ mono --aot=writesymbols,\
outfile=daxdivide.exe.so \
     --optimize=sse2 daxdivide.exe
...
$ file daxdivide.exe.so
daxdivide.exe.so: ELF 32-bit LSB shared
object, Intel 80386, version 1 (SYSV),
dynamically linked, not stripped
$ objdump -d daxdivide.exe.so >\
daxdivide.exe.so.dis
```

```
add    $0x23b4,%ebx
sub    $0x8,%esp
push   $0x7
push   $0x1a
call   119e <plt_Daxdivide_divider_int_int>
add    $0x10,%esp
lea    -0x4(%ebp),%esp
```

Et klip fra en meget længere disassembly-fil: Kaldet til divider-funktionen; assembler-kommandoen sub $0x8 læses* subtraher værdien 8 fra stackpointeren *- derved skabes plads til to 4-byte variable; register-navnet esp (for extended stackpointer) viser at der er tale om x86 kode.

Fordelen ved CIL-kode er med andre ord, at den kan køre på mange platforme (i modsætning til native kode, maskinens eget instruktionssæt). Den erfarne programmør ved, at man allerede i 1978 havde et p-Code system på universitetet i San Diego. Dengang var der mange slags hjemmecomputere, og for at de studerende kunne køre alle mulige øvelsesprogrammer, - og spil - lavede man et system, som dannede byte-kode, d.v.s. kommandoer, som var i byte-form, og som blev udført af en fortolker - d.v.s. et program, som læste byte-koderne og udførte dem. Til sidst ender alting i at være maskin-instruktioner af den slags, som maskinen er bygget til (native code).

Mono kan køre CIL kode på flere måder

Læg mærke til, at den CIL fil, vi klippede i på foregående side, ikke var den .exe fil, som F# compileren (fsharpc) danner, men derimod en tekstfil genereret af ***monodis***-kommandoen.

På Unix (Linux/BSD/OS-X/AIX m.fl.) kan man køre et ***mono-program*** på flg. måder fra kommandolinjen:

```
./daxdivide.exe
```

eller

```
mono daxdivide.exe
```

Normalt skal man ikke forsøge at køre .exe kommandoer på et Linux system (med mindre man ved nøjagtigt hvor den kommer fra og hvad dens formål er.) Spørg, hvad det er for en type fil, vi har genereret, med ***file*** kommandoen:

```
file daxdivide.exe
daxdivide.exe: PE32 executable (console)
Intel 80386 Mono/.Net assembly, for MS
Windows
```

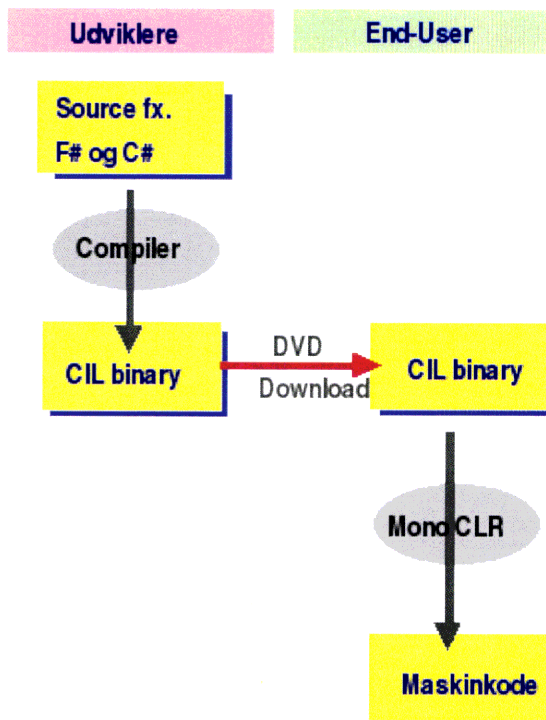
PE32: Portable Executable, en fil som indeholder CIL bytekoder + diverse environment information og symboler.

Hvordan kan Linux se, at det er et program, som kræver Common Language Infrastructure? Prøv igen:

```
./daxdivide.exe
lastcomm | head -n 2
cli              0.16 secs Wed Nov 12 13:40
binfmt-detector  0.00 secs Wed Nov 12 13:40
```

Der er klippet et par kolonner væk for at screen-dump kan være i bladet - men det vigtigste er med: Programmet ***binfmt-detector*** er kørt for at finde ud af, hvad det nu er for en fil - for den er jo ikke en almindelig (Linux/Unix) programfil.

De programmer, som er kørt sidst, står øverst, og øverst står ***cli,*** som er *Common Language Infrastructure.* Det er denne infrastruktur, som i sidste ende åbner for funktioner til kommunikation via en touch-screen - og dermed muliggør start og kørsel af programmer, *apps,* på en mobiltelefon ved tryk på skærmen.



Common Language Infrastructure skal naturligvis være specifikt tilpasset hardwaren og vil normalt følge med Android systemer og Microsoft Mobile

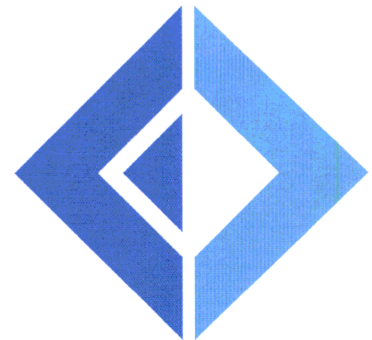
Som det fremgår af illustrationen, kan flere sprog kompileres til CIL. Man skal huske, at det vil være specielle dialekter af disse sprog (med undtagelse af C# som er "født" til CIL).

➥***fortsættes side 17***

F# eller F sharp

Kan man få både funktionsprogrammering, imperativ- og objektorienteret programmering i ét og samme programmeringssprog?

Af David Askirk



F# logo - fra fsharp.org

I denne artikel kommer der en kort intro til F#. F# er et sprog fra Microsoft som kører på .NET platformen. F# er et funktionelt sprog så tankegangen bag sproget er anderledes end andre sprog.

"Jamen hvorfor sidder jeg DKUUG-nyt og læser om noget fra Microsoft der kører på .NET?" Jo, F# kører på .NET platformen, så derfor kører det også på Mono, som er en cross-platforms implementation af .NET. Derfor kan vi bruge F# på mange platforme, og ikke kun på windows. Læs mere om Mono andetsteds i bladet.

For at komme igang med F# skal det installeres. Det kan man gøre fx. med apt-get.

Når F# er installeret er den nemmeste måde at komme igang at bruge det interaktive miljø.

Det startes med kommandoen **fsharpi**.

Nu kommer der en prompt frem og systemet er klart:

```
>
```

Kommentarer kan skrives efter // - dobbeltslash:

```
> // filnavn: dkuug-fsharp-eksempel.fs
```

Til at starte med, lad os undersøge nogle operatorer. I F# bliver linjerne afsluttet med `;;` (to semikoloner).

Vi kan skrive

```
> 2+2;;
```

og systemet giver os resultatet 4.

```
> 2+2;;
val it : int = 4
```

F# fortæller os at værdien (val) med navnet **it** er en int og den har værdien 4. It er standard navnet der blot henviser til sidste resultat. F# gætter på typen, og gætter på at typen af resultatet er en int.

Endnu et regnestykke:

```
> 2.0/0.1;;
val it : float = 20.0
```

Her bliver resultatet en float med værdien 20.0.

Et sprog et ikke meget bevendt, hvis vi ikke kan gemme de værdier vi finder. F# er et sprog der har meget fra matematikken, herunder også variabel-erklæringer.

En variabel erklæres således:

```
> let a = 2;;
```

Systemet svarer så dette tilbage:

```
val a : int = 2
```

Her fortæller F# at a har værdien 2 som er en integer.

Beregninger

Vi kan også gemme resultatet af en beregning:

```
> let b = 2 + 2;;
```

Systemet fortager beregningen og svare dette tilbage:

```
val b : int = 4
```

Hvad nu hvis vi gerne vil lave mere avancerede ting?

Funktioner

F# er et funktionelt sprog, og derved er funktioner en vigtig del af sproget. I funktionelle sprog bliver der lagt op til at funktioner ikke har nogle side-effekter. Dette betyder en funktion altid giver samme resultat givet samme input.

En funktion i F# erklæres med denne syntaks:

```
let <funktions navn> <parametre> = <funktions body>
```

Eksempel:

```
let foo a b = a + b;;
```

Dette giver en funktion der hedder foo som tager to parametre a og b. Den laver så beregningen a + b og dette bliver returneret.

```
> let add a b = a + b
- let sub a b = a - b
- let mul a b = a * b
- let div a b = a / b;;
val add : a:int -> b:int -> int
val sub : a:int -> b:int -> int
val mul : a:int -> b:int -> int
val div : a:int -> b:int -> int
```

Læg mærke til her at vi erklærer fire funktioner på en gang. Det kan man nemlig godt, det er først ved den sidste hvor der bliver afsluttet med `;;` at det bliver sat i værk.

F# fortæller os vi har fire funktioner.

Typen af add er:

```
val add : a:int -> b:int -> int
```

Dette skal læses som:

Add er en værdi der 1) tager en int (nemlig a), og 2) returnerer ***en funktion der tager en int (b), som returnerer en int.***

Ekspansionen eller forklaringen af, hvad *add* er, kan fås ved at taste ***add;;*** **-** den angiver kun værdier/typer, ikke operation på de to parametre. Med andre ord, *sub* og de andre funktioner er i dette tilfælde af samme opbygning.

Dette virker som en anderledes forklaring i forhold til at funktionen blev defineret med to parametre. Læg mærke til at vi ikke har fortalt F# at vores funktion arbejder med integers, men det gætter F# selv på.

I et sprog som javascript ville det se nogenlunde således ud:

```
function add (a) {
        return function (b) {
                return a+b;
        };
}
```

Med *nogenlunde* menes, at der ikke umiddelbart er andre sprog, hvor man nemt kan skrive noget der ligner.

Lad os prøve dem af:

```
> add 1 2;;
val it : int = 3
```

Her kalder vi add med to parametre 1 og 2. Disse tal bliver lagt sammen og resultatet 3 bliver returneret.

Her kommer den første lille skægge ting med funktioner. Vores **add** er ikke bare en funktion som får to int og returnerer summen (som heltal, int).

Typen af **add** er (som vi så på side 7) mere end bare det, den er *en funktion der returnerer **en funktion, der returnerer en int.***

Dette gør, at vi kan køre funktioner halvt, kan man sige.

```
> let add2 = add 2;;
val add2 : (int -> int)
```

Her bliver der lavet en ny funktion add2, som er en add hvor der mangler en parameter. Dette gør, at vi har bundet den ene parameter, og kun holder én fri, som vi kan bruge på et andet tidspunkt.

```
> add2 3;;
val it : int = 5
```

Her kalder vi vores ny funktion med værdien 3 og resultatet 5 kommer frem, helt som ventet. Dette kaldes currying af funktioner.

En anden måde at binde funktioner sammen er en måde som er kendt fra unix verden. Nemlig muligheden for at pipe funktioner sammen. Nu vil vi gerne fortage denne beregning:

(3 - (2 + 3))

Dette kan vi gøre på to måder:

```
> sub 3 (add 2 3);;
val it : int = -2
```

Eller ved at pipe resultaterne sammen:

```
> add 2 3 |> sub 3;;
val it : int = -2
```

I begge tilfælde bliver add funktionen kørt først og derefter sub funktionen. Hvad man skal bruge afhænger af konventioner samt af situationen.

Et andet eksempel er brug af **printfn**:

```
> printfn "Res: %d" 34;;
Res: 34
val it : unit = ()
```

Her kan vi også fortage beregningen og derefter skrive den ud på skærmen:

```
> 2+2 |> printfn "Resultat: %d";;
Resultat: 4
val it : unit = ()
```

Type systemet i F#.

Da F# kører på .NET platformen er alle de typer, man er vant til fra .NET, til stede.

Dog er der en speciel type. Da alt i F# skal have en type, eller returnere en type, kan man ikke have void (typeløse) funktioner.

Derfor er der den specielle type der hedder **unit**, som bliver brugt når man andre steder ville skrive void. **Unit** er stadig en type der bliver returneret, dog signalerer **unit** at der ikke er en værdi.

Et eksempel er print funktionen.

```
> printfn "Hello World";;
Hello World
val it : unit = ()
```

Der er flere ting ved denne funktion. For det første har den en side effekt (den skriver til skærmen) og for det andet returnerer den unit som type, dvs. der kommer ingen værdi tilbage fra funktionen andet end unit.

Lister

Da F# er et funktionelt sprog er lister en stor del af sproget.

En liste i F# erklæres således:

```
> let a = [];;
val a : 'a list
```

Her fortæller F# os at listen a er en liste af en anonym type. Vi erklærede en tom liste uden elementer med [].

En liste uden elementer er ikke sjov, så lad os lave en liste med noget i.

```
> let a = [1;2;3];;
val a : int list = [1; 2; 3]
```

Her er en liste med tre elementer. Tre integers med værdierne 1, 2 og 3. Når en liste erklæres på denne måde, bliver elementerne adskilt af `;` (semikolon).

Dette kan også læses som en tom liste, hvortil der er tilføjet talet 3, dernæst tilføjet 2 for til sidste at tilføje 1.

Dette kan skrive i F# således:

```
> let b = 1::2::3::[];;
val b : int list = [1; 2; 3]
```

`::` operatoren (dobbelt-kolon) tager et element og tilføjer det til en liste.

Når vi giver en liste med til en funktion har vi mulighed for at "skille" listen ad på denne måde.

Lad os skrive en funktion der finder det første element i en liste:

```
> let head x =
      match x with
            | [] -> failwith("Empty list")
            | a::[] -> a
            | a::rest -> a;;
val head : x:'a list -> 'a
```

Her kan vi se det er en funktion der tager en liste ind og returnerer et element.

Lad os prøve vores funktion:

```
> head [1;2;3];;
val it : int = 1
```

Vi kan se den virker.

Magien i denne funktion ligger i ***match .. with*** formen. Her går vi ind og prøver at matche det data vi får ind med de tre forskellige former:

1. En tom liste, så smider vi en fejl.
2. Hvis der kun er et element i listen.
3. Hvis der er mere end et element i listen.

Ved at bruge :: splitter vi listen op.

Sum af elementer i en liste

Lad os skrive endnu en funktion. Nemlig en laver en sum af elementer i en liste:

```
> let rec sumOfList x =
      match x with
            | [] -> 0
            | a::rest -> a+(sumOfList rest);;
val sumOfList : x:int list -> int
```

Her kan vi se at F# fortæller os at det er en funktion der tager en liste af integer og returnere end int. Lad os prøve den:

```
> sumOfList [1;2;3];;
val it : int = 6
```

I denne funktion bliver der brugt det at man kan matche med ***match .. with*** så en liste bliver splittet op. Da det er en funktion vi kalder rekursivt, dvs. funktionen kalder sig selv, putter vi **rec** ind efter **let**.

Lad os lave en funktion der kører en funktion for hvert element i en liste og returnerer en ny liste:

```
> let myMap f x =
      let rec innerMap f x acc =
            match x with
            | [] -> acc
            | a::rest -> innerMap f rest ((f a)::acc)
      innerMap f x [];;
val myMap : f:('a -> 'b) -> x:'a list -> 'b list

> let square x = x*x;;
val square : x:int -> int
```

Her laver vi to funktioner. Den anden, kaldet square, tager et tal ind og ganger det med sig selv.

Den første funktion er en funktion med en funktion inden i sig. Dette kan lade sig gøre da funktioner i sig selv også er værdier. Her er der en ydre funktion der tager i mod en liste og en funktion. Læg mærke til at F# kan se at funktionen tager en værdi ind og returnerer en anden værdi.

Vi kan ikke se noget om den indre funktion, da den er skjult. Dette gør også, at vores midlertidige lager, kaldet acc, er skjult. Dette er et eksempel på en closure.

Fakta-box

In programming languages, a **closure** (also **lexical closure** or **function closure**) is a function or reference to a function together with a *referencing environment*—a table storing a reference to each of the non-local variables (also called free variables or upvalues) of that function.

Lad os bruge denne map funktion:

```
> myMap sqaure [1;2;3;4];;
val it : int list = [16; 9; 4; 1]
```

Her kan vi se at vi får listen af kvadrat tal, men i omvendt rækkefølge. Dette skyldes denne linje i den indre funktion:

```
| a::rest -> innerMap f rest ((f a)::acc)
```

Den linje siger:

kør funktionen på resten af listen og dernæst, tag og kør funktionen f på det første element og tilføj det til den interne liste.

Egne typer

I F# er det muligt at definere sine egne typer. Dette gør det nemt at lave et program der passer til et bestemt domæne. Her vil der blive lavet et kryds og bolle spil.

Et spil skrevet i F#

Nu vil vi gerne erklære typerne til et kryds og bolle spil:

```
> type Cell =
    | X
    | O
    | E;;
type Cell =
  | X
  | O
  | E
```

Der skal ikke mere til. Nu findes type **Cell** der kan have tre værdier: X, O **eller** E.

I vores kryds og bolle spil bruger vi en liste til at gemme brædtet i.

Vi giver positionerne tal således:

```
0 | 1 | 2
---------
3 | 4 | 5
---------
6 | 7 | 8
```

Ved at gøre dette kan vi mappe positionerne i en liste til positionerne på et spille brædt.

Lad os oprette det tomme brædt, altså sådan et brædt ser ud når vi starter et nyt spil.

```
> let emptyBoard = [E;E;E;E;E;E;E;E;E];;
val emptyBoard : Cell list = [E; E; E; E; E; E; E; E; E]
```

Her kan vi se det er en liste af vores type Cell. Der er ni i alt. Dette er et tomt brædt som det ser ud inden spillet går igang.

For at se hvem der har vundet skal man undersøge alle mulighederne, vandret, lodret og diagonalt.

Fra frokoststuen:

A general user calling: "Who is General Failure, and why is he reading my disk?"

Lady calling in: "FAT bread failed? That is an insult!"

Kollega i direktionen: Jeg forstår ikke, der står tryk på ***alt + F1*** ... men når jeg trykker på alt, trykker jeg jo også på F1?

Fornærmet kunde: De siger, at ***alt*** er under ***kontrol***, men jeg synes, de sidder ved siden af hinanden!

Kryds og bolle spil:

Dette kan gøres således i F#:

cell (med lille forbogstav) er instantiering af noget - det er altså ikke en Cell-type med mindre vi gør den til det senere? Kan erstattes af whatever.)

```
> let hasWon cell board =
  match board with
        | [a;b;c;_;_;_;_;_;_] when a = cell && b = c && a = c -> true
        | [_;_;_;a;b;c;_;_;_] when a = cell && b = c && a = c -> true
        | [_;_;_;_;_;_;a;b;c] when a = cell && b = c && a = c -> true
        | [a;_;_;_;b;_;_;_;c] when a = cell && b = c && a = c -> true
        | [_;_;a;_;b;_;c;_;_] when a = cell && b = c && a = c -> true
        | [a;_;_;b;_;_;c;_;_] when a = cell && b = c && a = c -> true
        | [_;a;_;_;b;_;_;c;_] when a = cell && b = c && a = c -> true
        | [_;_;a;_;_;b;_;_;c] when a = cell && b = c && a = c -> true
        | _ -> false;;
val hasWon : cell:'a -> board:'a list -> bool when 'a : equality
```

Her kan vi se det er en function der tager et element at teste imod, en liste af Cell ind og returnerer sand eller falsk.

Vi laver samligningen ved at sige vi ønsker at tage fat i de tre første variabler - og vi er ligeglade med resten.

Dette bliver gjort i denne:

```
[a;b;c;_;_;_;_;_;_]
```

Da vi gerne vil undersøge om de tre valgte variabler er ens, samt om de er det samme som den brik vi undersøger for ligger vi en guard condition ind:

```
when a = cell && b = c && a = c
```

Denne siger at når a matcher den *cell* vi kigger efter, samt b er lig med c og a er lig med c så er der tre ens på plads 0, 1 og 2, og derved er der en spiller der har vundet.

Til sidst bliver der angivet hvad der returneres hvis systemet kan matche med de værdier vi har givet ind. Hvis der er fundet en vinder bliver der returneret true.

Hver mulighed i matchingen bliver adskilt af **|** (pipe) og den sidste er en catch-all der matcher alt og returnerer falsk. Den returnerer falsk hvis der ikke er fundet en vinder.

Med dette i hånden er det nemt at lave de to funktioner for at se om X eller O har vundet.

Vi benytter os af den delvise funktions opbygning således:

```
> let hasXWon = hasWon X;;
val hasXWon : (Cell list -> bool)
```

og

```
> let hasOWon = hasWon O;;
val hasOWon : (Cell list -> bool)
```

Ved at gøre dette, får vi to funktioner mere, som kan teste om enten X eller O har vundet. Læg mærke til at vi skrev en generel funktion og brugte currying til at give os to specifikke tilpassede funktioner.

For at fortage et træk på vores brædt skal vi lave en funktion der går ind og erstatter et element i en liste:

```
> let rec replaceNth l pos elem =
      match (l, pos) with
              | ([], _) -> []
              | (a::rest, 0) -> elem :: rest
              | (a::rest, n) -> a::(replaceNth rest (n-1) elem);;
val replaceNth : l:'a list -> pos:int -> elem:'a -> 'a list
```

Denne funktion giver vi en liste, en position der skal erstattes på og det element som skal sættes ind på det sted i listen.

Ved matchingen laver vi et trick, da vi gerne vil matche på listen og på vores position. Vi pakker dem sammen i en tupple, så vi kan bruge begge værdier i vores matching.

Nu kan den funktion der fortager et træk laves:

```
> let placeMove cell pos board = replaceNth board pos cell;;
val placeMove : cell:'a -> pos:int -> board:'a list -> 'a list
```

Den kalder blot replaceNth, dog pakket ind på en pænere måde.

Lad os lave et kald:

```
> placeMove X 5 emptyBoard;;
val it : Cell list = [E; E; E; E; E; X; E; E; E]
```

Den sætter X'et lige der hvor vi gerne vil have det.

Så har vi vores funktioner til at styre vores spil.

Disse funktioner kan man så kalde i den interaktive shell og man har derved et kryds og bolle spil i F#.

Herunder er et eksempel på et spil:

```
> let newBoard = placeMove X 4 emptyBoard;;
val it : Cell list = [E; E; E; E; X; E; E; E; E]

> hasXWon newBoard;;
val it : bool = false

> hasOWon newBoard;;
val it : bool = false

> let newBoard = placeMove O 2 newBoard;;
val it : Cell list = [E; E; O; E; X; E; E; E; E]

> hasXWon newBoard;;
val it : bool = false

> hasOWon newBoard;;
val it : bool = false

> let newBoard = placeMove X 3 newBoard;;
val it : Cell list = [E; E; O; X; X; E; E; E; E]

> hasXWon newBoard;;
val it : bool = false

> hasOWon newBoard;;
val it : bool = false

> let newBoard = placeMove O 8 newBoard;;
val it : Cell list = [E; E; O; E; X; E; E; E; O]

> hasXWon newBoard;;
val it : bool = false

> hasOWon newBoard;;
val it : bool = false

> let newBoard = placeMove X 5 newBoard;;
val it : Cell list = [E; E; O; X; X; X; E; E; O]

> hasXWon newBoard;;
val it : bool = true
```

→→→→ *forts.næste side* →→→→

Spillet slutter med X der vinder.

Vi har kigget på F# som er en anden måde at programmere på. I modsætning til objekt orienteret programmering er funktioner i højsædet i F#, samt adskillelsen af kode med side-effekter og det som ingen sideeffekter har.

Da F# kører på Mono er det til rådighed på alle platforme samt giver mulighed for at skrive kode i F# som så efterfølgende kan bruge i C# eller andre sprog på .NET/Mono platformen.

F# er et funktionelt sprog, der lægger op til at man adskiller kode, der har sideeffekter fra kode der ikke har. Det er generelt en god måde at programmere på, også i ikke funktionelle sprog. Så er det nemmere at have en funktion der beviseligt er korrekt.

Links og bøger til at komme videre med F#:

http://www.tryfsharp.org/

https://www.microsoft.com/learning/en-us/book.aspx?ID=16172&mc=US

http://shop.oreilly.com/product/0636920024033.do

http://fsharpforfunandprofit.com

Klip om Open Data Open Knowledge

De fleste styrelser har åbnet for data; Energistyrelsen viser informationer om olieudvinding

Open Government Data -

Her er nogle pluk fra styrelser, ministerier og institutioner:

Digitaliseringsstyrelsen

Open Government Danmark har i lighed med mere end 60 andre lande tilsluttet sig det internationale initiativ, Open Government Partnership (OGP), som blev stiftet i 2011 af den amerikanske regering i samarbejde med syv andre lande, heriblandt Norge og Brasilien.

Energistyrelsen

Sidste år havde vi en artikel, hvor åbning af GIS (Geografisk Informations-System). Energistyrelsen, ens.dk, har også offentliggjort data om olieudvinding. Der er interaktivt kort med oplysninger, men også ganske almindelige rapporter. Der blev fundet olie i Danmark (Vesterhavet) 1966, og der er udvundet olie siden 1972. Alle rapporter gennem årene er offentligt tilgængelige online med link til download.

Københavns Universitet

Teknologiske trusler kan afværges med åbenhed om viden.

FRI VIDENSDELING

Som led i 100 års jubilæet for Niels Bohrs atommodel samler Københavns Universitet den 4.-6. december internationale forskere og beslutningstagere til konferencen "An Open World". Konferencen følger op på Bohrs kamp for en verdensorden, der bygger på åbenhed. Højaktuelle hovedtalere skal debattere fri cirkulation af viden – målet er at skrive et fælles brev til FN med anbefalinger til, hvordan international åbenhed kan bidrage til at håndtere teknologiske og videnskabelige udfordringer.

-

Konferencen "An Open World: Science, Technology and Society in the Light of Niels Bohr's Thoughts", tager udgangspunkt i et åbent brev, som den verdensberømte fysiker skrev til FN i 1950. I brevet opfordrer Bohr, på baggrund af udviklingen af atombomben, til åbenhed omkring forskning og teknologi for at forhindre destruktive kapløb drevet af mistillid og frygt og i stedet samarbejde om at realisere produktive muligheder.

- Med inspiration fra Bohrs brev til FN skal konferencens hovedtalere sende et nyt brev til FN, der giver anbefalinger til, hvordan internationale udfordringer kan håndteres med åbenhed og fri vidensdeling. Målet med konferencen og brevet til FN er at skabe debat om åbenhed. I modsætning til Bohr vil vi afsende et fælles budskab, der giver konkrete anvisninger, siger Ole Wæver.

På konferencen vil en lang række prominente internationale forskere og praktikere diskutere muligheder og risici ved åbenhed i forholdet mellem videnskab, teknologi og samfund. Heriblandt:

- *Jimmy Wales, stifter og talsperson for Wikipedia*
- *Irina Bokova, generalsekretær for UNESCO*
- *Susan Crawford, Obamas tidligere rådgiver for teknologi og videnskab*
- *Rolf-Dieter Heuer, direktør for CERN*
- *Sir Nigel Shadbolt, medstifter og bestyrelsesformand for The Open Data Institute*
- *Wael Ghonim, cyberaktivist med central rolle i den egyptiske revolution*
- *Richard Rhodes, Pulitzer-vindende forfatter og historiker med speciale i atomvåben*
- *Dennis Meadows, professor og medforfatter til den revolutionerende bog 'Limits to Growth'*
- *Abdallah Daar, kirurg og professor i folkesundhedsvidenskab med fokus på global sundhed*

OKFN, Open Knowledge Foundation Danmark

Der afholdes en dataworkshop i samarbejde med Informations (avisens) Datablog 26 november 16:30 i Informations lokaler i København. Man kan tilmelde sig hos Niels Erik Kaaber Rasmussen.

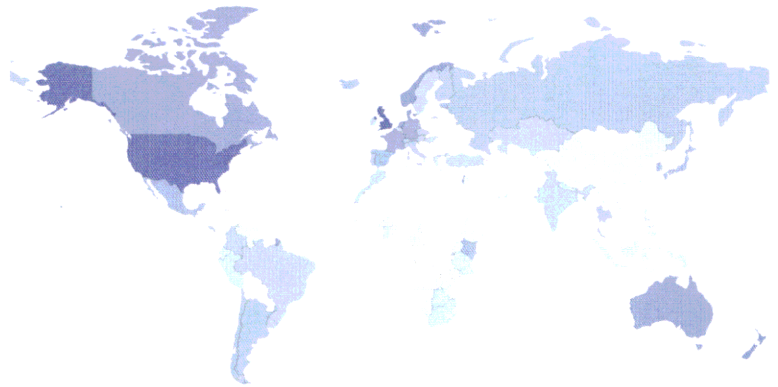
> *I Open Knowledge arbejder vi for udbredelse af åben viden. Vi arbejder for åben adgang, åben regeringsførelse, åben uddannelse, åbne data og åben-alt-muligt.*

dk.okfn.org

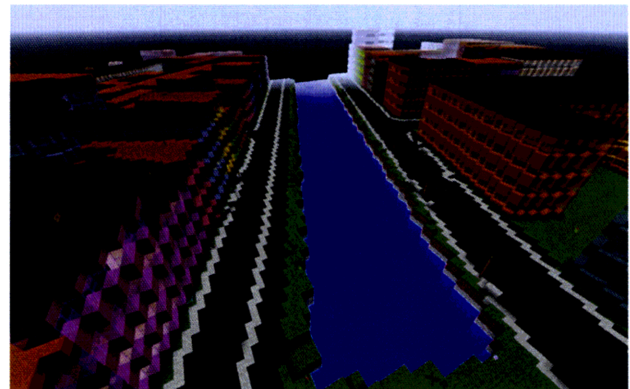
Internationalt

Open Data Barometer tager et overblik på Open Government Data (OGD) over hele verden ved at se på tilgængeligheden af 14 nøgleemner.

I rapporten for 2013 finder man dette kort:



Hvis man vil deltage i den offentlige debat eller bedre, forske i hvordan økonomiske og politiske forhold er globalt set, er der andre kilder til internationale informationer: United Nations Development Programme (UNDP) og US CIA factbook.



Since the US government opened its troves of public data we've seen some pretty neat projects like climate-change prediction tools and deforestation-monitoring systems. Denmark, on the other hand, has taken a different approach: the Danish Geodata Agency used internally developed topographic maps and elevation models to build a 1:1 recreation of the happiest country within Minecraft's blocky confines.

Havde det ikke været fordi Svensk Politi havde fået kløerne i Warg til at begynde med, ville vi idag ikke have anet at der var indbrud hos CSC, ej heller at eller hvilke filer der var blevet kopieret.

Poul-Henning Kamp 31. okt. 2014 (Version2).

Shell Shock - eller chok

Shell Shock (på dansk Shell chok) er en familie af sikkerhedshuller i Bash

Den første offentliggørelse af dette sikkerhedshul kom 24. september 2014. Som med alle sikkerhedshuller var det meningen, at der skulle være en periode, hvor rapporter om denne kun blev sendt til virksomheder og institutioner med servere.

Men den dato, 24. sept. er jo ikke datoen for hvornår sikkehedshullet er opstået! Det går længere tilbage, meget længere, ligesom med Heartbleed buggen er det et sikkerhedshul, som meget vel kan være udnyttet af diskrete agenturer i fjendtligtsindede spiontjenester. Nu vil redaktionen på bladet her jo helst ikke blive kendt for at være tilhænger af konspirationer og lignende, men vi kan i det mindste sige så meget, at vore egne sikkerhedstjenester gør opmærksom på aktiviteter, som kan tyde på at andre stater har etableret servere, som kan medvirke til at stoppe trafikken på Internettet. Sådanne mere diskrete institutioner kan have et helt arsenal af sårbarheder, som kan udnyttes, hvis det skulle ønskes.

```
@sat2:~$ ls -l
total 8
-rw-r--r-- 1 donax donax  239 Mar 13  2013 MVI_1274.mov.kino
-rw-r--r-- 1 donax donax 2627 Nov 14 02:56 shellshock_test.sh
@sat2:~$ X='() { (a)=>\' bash -c "echo date"
bash: X: line 1: syntax error near unexpected token `='
bash: X: line 1: `
bash: error importing function definition for `X'
donax@sat2:~$ echo $X

@sat2:~$ ls
echo  MVI_1274.mov.kino  shellshock_test.sh
@sat2:~$ cat echo
Fri Nov 14 03:04:41 CET 2014
@sat2:~$ ls
echo  MVI_1274.mov.kino  shellshock_test.sh
@sat2:~$ █
```

Kommandolinie eksempel som viser sikkerhedshullet

Eksemplet viser kommandolinien

```
$ X='() { (a)=>\' bash -c "echo date"
```

Bemærk placeringen af single-quotes (apostrof, kaldes den ofte på dansk). Det skyldes at der er space mellem parenteserne og de krøllede parenteser. Derimod gør det ikke noget, at der er "hul" mellem den sidste singlequote og ordet bash.

Der er *heller ikke en slut - krøllet parentes!* (Krøllede parenteser = braces). Det er selvfølgelig en fejl, men sikkerhedshuller består ofte i at udnytte, hvordan et program reagerer på fejl-input. Her er der yderligere fejl - efter den fejlagtige funktions-definition (slutter efter sidste singlequote) er der en kommando - og bash vil forsøge at gøre et eller andet med den.

Træd tilbage og se!

Der sker det, at bash danner en fil, "echo", med indholdet at output fra ***date*** kommandoen!

I eksemplet er indholdet af directory vist *inden* sårbarheden udnyttes. Der er *ikke* en fil ved navn echo.

Efter den mislykkede definition af en funktion X køres en bash-kommando, som sender output af kommandoen ***date*** til en fil, som hedder *echo.*

Det er en ganske almindelig fil, som tilhører den konto, bash kører på (user og group). Det åbner for mange problemer.

Naturligvis burde bash ikke køre nogen kommando og burde blot afvise hele den mislykkede funktionsdefinition.

Det er kun én af de sårbarheder, som har med denne mekanik at gøre og den kan udnyttes på mange måder.

Desværre kan vi her på redaktionen konstatere, at den samme sårbarhed findes for den pdksh, som vi ellers har været så glade for!

Desværre kan vi også konstatere, at den version af bash, som vi opdaterede med, ***ikke*** *var blevet rettet så den kunne modstå scripting med Shell Shock!*

Prompte reaktion

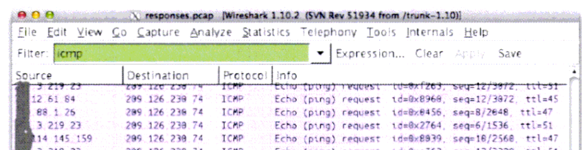
Ikke nok med at der er et sikkerhedshul! - Der er mange flere problemer med Shell Shock: Det kan konstateres, at der efter prenotifikation opstod en læk - en af de informerede parter sendte sårbarheds-specifikationen videre ud på nettet og blot timer efter kunne man konstatere, at der var angreb, ondsindede hackere, som etablerede bot-nets, d.v.s. lange lister med maskiner, som var blevet "åbnet" med diverse scripts som jo kan udføres, hvis denne bug ikke er rettet. Det medfører spekulationer om hvordan man kan sikre server-community mod lækager, og det er en sørgelig udvikling.

Flere variationer af Shell Shock

Det er ikke kun ved distribution af underlige shell-linjer (fx. i programpakker), at man kan angribe. Shell-sårbarheden kan udnyttes gennem HTTP protokollen, som har mange typer headers. Troy Hunt, website for "Alt hvad du behøver at vide om sårbarhed X" viser et eksempel:

```
target = 0.0.0.0/0
port = 80
banners = true
http-user-agent = shellshock-scan\
(http://blog.erratasec.com/2014/09/bash-
shellshock-scan-of-internet.html)
http-header = Cookie:()\
 { :; }; ping -c 3 209.126.230.74
```

Det var nødvendigt at bryde linjerne. Se mere på Troyhunt.com



Filtrering af tcpdump - CC - Creativ Commons, Troy Hunt

Navnet spiller selvfølgelig at det er samme ord som "granat chok", den traumatiske tilstand, som opstår hvis man har været udsat for tæt beskydning.

IPv6 i medierne

TLDs with IPv6 nameservers: 714
Percentage of TLDs with IPv6 nameservers: 96.0%

Screen-capture fra Tunnelbroker.net ■

➥ *fortsat fra side 5*

Mono status og udvikling

Mono version 3.6.0 blev releaset i august 2014. Denne version giver kernen af .NET Framework og der er nu support for flere sprog:

har også support for Visual Basic.NET og C# version 2 3 og 4.0 Desuden har den LINQ support til objekter (fx. in-memory tabeller og lister). XML og SQL er en del af distributionen.

C# 4.0 er default modus for C# compileren. Windows Forms 2.0 er også mulige at bruge, men der udvikles ikke aktivt på det, og alt i alt er Monos understøttelse af Windows Forms ikke komplet.

Mono sigter efter at kunne give fuld support for .NET 4.0 med undtagelse af Windows Presentation Foundation (WPF) (som Mono-team'et ikke planlægger at supportere på grund af opgavens størrelse.

Der er underprojekter i Mono, som arbejder på implementering af de manglende dele af .NET.

Et af elementerne i .NET er en streaming media afspiller, Microsoft Silverlight, og der har været forsøgt en tilsvarende Open Source *Moonlight*, som Icaza i 2011 annoncerede at man ikke ville fortsætte, blandt andet kritiserede han det for at udvikle sig i retning af ***bloatware,*** d.v.s. software, som prøver for meget og fejler i et orgie af broken features.

Ud af disse og andre *gamle* nyheder kan man se at .NET både er et framework for applikationer på mobile enheder af meget forskellige fabrikater og typer, og at en vigtig del her som i andre applikationsmiljøer er beherskelse af markedet.

Alt i alt må man dog konkludere, det subset af .NET, som fungerer på Mono platformen, giver producenter af apps mulighed for at løse de fleste opgaver på mobile platforme.

- Og man kan tilføje, at navnet Mono måske i højere grad leder tankerne hen på et ensartet miljø for mange platforme, **mono**kultur.

Hvorfor er det godt at Mono supporterer flere sprog?

Når man ser nogle af de mange F# introduktioner, man kan finde på nettet, bl.a. på YouTube, opdager man hurtigt, at F# har mulighed for at bruge C# queries og andre libraries, som hører til .NET (og Mono) miljøet.

Denne egenskab gør sproget mere egnet til specielle opgaver, fx. et mailfilter skrevet i F# og et system til håndtering af indgående betalinger skrevet i C.

Paul Graham, som har arbejdet for bl.a. Yahoo Store, oprettede i midten af 1990'erne virksomheden ***Viaweb,*** der senere blev solgt til Yahoo. Han valgte *Lisp* - F# var ikke en mulighed dengang - men Lisp var (og er) et funktionsorienteret sprog ligesom F#, og Lisp kan (ligesom F# i Mono-miljøet) bruges sammen med andre sprog i en større applikation.

Graham skriver:

> *Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The first version was mostly Lisp, because the ordering system was small. [...]*

Ovst. er fra en fodnote, men er nødvendig for at forstå resten:

> *So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.*
>
> *What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a WYSIWYG online store builder that ran on the server and yet felt like a desktop application. Our competitors had CGI scripts. And we were always far ahead of them in features.*
>
> *Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too. It must have seemed to our competitors that we had some kind of secret weapon--that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.*

Det er grunden til at Lisp (**LIS**t **P**rocessing) i dets mange varianter har spillet en stor rolle indenfor AI forskning. Et andet eksempel: Richard Stallman valgte Lisp som extension sprog for Emacs editoren, som han begyndte på i 1984. Og det var effektivt! Tænk lige på hvor små selv store computere var dengang ☺

I/O i F#

Men man **kan** lave I/O med F#. Se følgende minimal-eksempel fra Rosettacode.org:

```
open System.IO

[<EntryPoint>]
let main argv =
    File.ReadLines(argv.[0]) |> Seq.iter (printfn "%s")
    0
```

Men hvis man søger efter dokumentation af stream-I/O på fsharp.org, bliver man henvist til MSDN, Microsoft Developer Network - og dér finder man ikke altid dokumentation og eksempler for F#. Der står fx.:

> *The following examples show how to read text synchronously and asynchronously from a text file using .NET for desktop apps [...]*

og lige nedenunder:

> *No code example is currently available or this language may not be supported.*

➥ **forts. næste side**

etwork Technologies Downloads Programs Community Documentation Samples Sign in

ary
opment
ework 4.5
t Guide
Essentials
eam I/O
I/O Tasks
o: Copy Directories
o: Enumerate Directories
iles
o: Read and Write to a
Created Data File
o: Open and Append to a
ile
o: Write Text to a File
to: Read Text from a File
o: Read Characters from a
o: Write Characters to a
o: Add or Remove Access
ol List Entries
o: Compress and Extract
sing Streams
o: Convert Between .NET
ework Streams and Windows

How to: Read Text from a File

.NET Framework 4.5 | Other Versions ▾ | 20 out of 38 rated this helpful - Rate this topic

The following examples show how to read text synchronously and asynchronously from a text file using .NET for the instance of the StreamReader class, you provide the relative or absolute path to the file. The following example the same folder as the application.

These code examples do not apply developing for Windows Store Apps because the Windows Runtime provides di For an example that shows how to read text from a file within the context of a Windows Store app, see Quickstart: show how to convert between .NET Framework streams and Windows runtime streams see How to: Convert Betwe Runtime Streams.

◢ Example

The first example shows a synchronous read operation within a console application.

C# | VB | F#

No code example is currently available or this language may not be supported.

The second example shows an asynchronous read operation within a Windows Presentation Foundation (WPF) a

C# | VB | F#

No code example is currently available or this language may not be supported.

Men andre steder kan man finde gode vejledninger og dokumentation; det vigtigste er specifikation af, hvordan F# programmoduler kan sættes i system med andre, og det sker typisk via en database. F# supporterer LINQ, og det er et nyttigt emne ☺ - som en smagsprøve er her et klip fra MSDN for F#:

```
// Use the OData type provider to create
// types that can be used to access the
// Northwind database.
open Microsoft.FSharp.Data.TypeProviders

type Northwind =
 ODataService <"http://s.org/North.svc">
let db = North.GetDataContext()

// A query expression.
let query1 =
  query { for customer in db.Customers do
                    select customer }
```

Query expressions er veldokumenterede - og der findes mange, mange, mange:

query expressions

```
// F# query expression
let parents = query {
    for m in family do
    where (m.Age > 18)
    sortByDescending m.Age
    select m
    distinct
    take 2
}
```

many more
query operators

Lånt fra YouTube DevelopMentor1: Introduction to F# med Wallace Kelly, interessant og let at følge

En lignende query i C# kan fx. se sådan ud:

```
var queryLondonCust =
  from cust in customers
      where cust.City == "London"
      select cust;
```

Overordnet set er sprogene i Mono - miljøet sideordnede. De kan tale sammen via class interfaces/libraries.

Med tanke på artiklen i dette blad om F# kan det konkluderes, at F# skal i Mono kunne det samme som de andre sprog (særligt tænkes her på C#) men som vi har set, er der områder, hvor eksempler og vejledninger mangler, iøvrigt for operationer, som man typisk vil udføre med andre sprog eller system-tools.

Man kæder de forskellige (Mono-)sprog sammen via ***interface*** definitioner, svarende til prototypedefinitioner i header-filer for C-programmer. Interface, abstrakte typer, mv. er måden at administrere store projekter - de kan deles op i mindre, som derved er lettere at holde styr på.

I webserver miljø, hvor programmer kan kommunikere via pipes eller filer, vil man også kunne kombinere Mono-miljøet med andre (typisk ældre) funktionsmoduler.

Opsummering

Filosofien bag Mono er at være et nyttigt, effektivt værktøj til cross-platform programmering, som gør det muligt at afvikle programmer skrevet i forskellige sprog - også database query sprog mv. Vi ser det hver dag på smartphones.

Grunden til at disse programmer kører godt og ikke kræver omskrivning, hver gang der kommer en ny device på markedet, er at Mono - som er Open Source i modsætning til .NET - danner CIL kode, som derefter kan afvikles (eller oversættes) på brugerens device uden større forsinkelse. Ideen er ikke ny, men omfanget af libraries er stort og foreløbig er der udstrakt kompatibilitet; ***derfor*** opnår udviklere af apps og andre applikationer en større effektivitet og større udbredelse på alskens devices.

Med Mono bliver man en del af et stort netværk, og dermed også bliver man også afhængig af, at projektet fortsætter og at der vil være support-classes/libraries for nye devices. Derfor må man spørge sig selv om forretningsmodellen går fri af licenser og copyright i alle dele af systemet. (Fx. Apple-monotouch er ikke Open Source) Prisen er overkommelig, også for mindre virksomheder. Gevinsten er et stort marked for apps.

Sidste nyt! ... i skrivende stund:

Fremtiden for Mono

Netop i denne måned - faktisk i skrivende stund! - er der kommet en annoncering fra Microsoft Developer Net Blog, en af dem ses her: ***We are happy to announce that .NET Core will be open source, including the runtime as well as the framework libraries.***

Er det begyndelsen til enden af Mono, Open Source CIL-platform?

Server & Tools Blogs > Developer Tools Blogs > .NET Framework Blog

Executive Bloggers	Visual Studio	Application Lifecycle Management	Languages	.NET Framework	Platform Development

.NET Framework Blog

A first hand look from the .NET engineering teams

Like
Follow
Follow

.NET Core is Open Source

Immo Landwerth [MSFT] 12 Nov 2014 7:39 AM 60

RATE THIS

Translate t

Spanish

Microsoft®

Today is a huge day for .NET! We're happy to announce that .NET Core will be open source, including the runtime as well as the framework libraries.

This is a natural progression of our open source efforts, which already covers the managed compilers (C#, VB, and F#) as well as ASP.NET:

- C# & Visual Basic ("Roslyn")
- Visual F# Tools
- ASP.NET 5

.NET Dov

.NET NuG

.NET SDK

■

Kommandocentralen

... i dag med eksempler på ting, som er lettere at gøre fra kommandolinjen:

Har du nogen gange siddet og tastet på en laptop med touchpad, og pludselig var cursoren langt borte: "det var som bare ... "

Hvor skal du støtte hånden uden at ramme touchpad og derved få cursoren til at fare forvildet henover skærmen?

syndaemon er et program, som overvåger tastaturet, og hvis der er aktivitet på tastaturet, bliver touchpad disablet.

Man kan indstille, hvornår den skal slås til igen - 2 sek. efter sidste tastetryk er default, så aktiveres touchpad. Poll-interval kan også justeres.

Det er et af de programmer, som man ikke aner kører hvis ikke man ser efter med proces-monitoreringsprogrammer.

Proces listning

Hvordan ser man, om den proces kører?

```
# ps -ef | grep synd
```

Fordelen ved denne metode er, at man ikke behøver at læse så meget. En taskmgr, hvor man kan søge på programnavne, vil selvfølgelig gøre næsten det samme.

Proces overvågning

Hvordan ser man tilstanden - hvad kan man få at vide om en kørende proces? Kort eksempel (man kan ***meget*** mere!):

```
# ps -lC syndaemon
```

Det viser bl.a. nice value, altså, hvor højt eller lavt prioriteret processen er. Nice value -20 er *superpower* - den får det meste.

Hvordan ser man, hvad lige netop den proces har "kostet" i tid?

```
# ps u -C syndaemon
```

De fleste brugere af Unix kender ***top*** - programmet, som viser hvilken proces, der belaster mest. Men somme tider er det minimale output fra ps lettere at bruge fx. i et script:

```
# ps --no-headers -o pcpu -C syndaemon
```

Denne kommandolinie vil ***kun*** skrive en procent (procent cpu, pcpu) på skærmen. -o betyder brug de felter, som jeg angiver i en kommasepareret liste efter -o fx.:

```
# ps -o pcpu,pmem,pid,ppid,cmd
```

--no-headers instruerer om at man ikke ønsker linjen med betegnelser (typisk i et script, hvor man bare vil have tallet).

-C søger på kommandonavnet. Det kan give lidt hovedpine en gang imellem at stave til et program, som man måske har startet via en GUI menu. I så fald må man gribe til andre midler fx:

```
# ps -o pcpu -p $(pgrep synda)
```

hvor -p betyder *ud fra proces ID (PID)*, og pgrep leverer PID ud fra et navn eller bare en del af et navn.

Hørt i support-afdelingen:

Idle-processen tager nogen gange 99% cpu tid! ***dovne maskine ...***

--o--

En supporter skulle have en bruger til at taste et "større-end"-tegn. Brugeren mener ikke, at han har et sådan tegn på sit tastatur. "Jeg har et lille N og et stort N, men jeg har ikke et større N!"

■

SUPERUSERS Skal du til at udvikle Web Apps?
www.superusers.dk/web_apps.htm

Web Apps har større fokus end nogensinde – Web Apps giver platformsuafhængighed.
Client-side i browser: Programmeres i HTML5, CSS3 og JavaScript samt JavaScript .js-frameworks.
Server-side på en/flere servere: Programmeres i PHP/Perl/Python/Ruby/C# eller JavaScript/Node.js

SU-901 Web App Programmering Grundkursus (inkl. Mobile Device)

Du lærer begreber og teknologier og arbejder med nogle af markedets førende tools inden for udvikling af Web Apps.

Varighed / Pris: 2 dage / 8.900,- kr. (ekskl. moms)

Afholdelsesgaranti på understregede datoer:

8–9/12 • 8–9/1 • 5–6/2 • 9–10/3 • 9–10/4

SU-902 Web App Programmering af Datahåndtering

Du udvikl. en prof. Web App m. fokus på datalagring på mobil-device i filer og SQL-db samt brug af web-services og netværk.

Varighed / Pris: 3 dage / 11.100,- kr. (ekskl. moms)

Afholdelsesgaranti på understregede datoer:

26–28/1 • 23–25/3 • 27–29/5

SU-904 Web App Programmering af Brugergrænseflader – Foundation og Bootstrap

Du laver professionelle brugergrænseflader og sender information imellem forskellige skærmbilleder. Du anvender grafik.

Varighed / Pris: 3 dage / 11.100,- kr. (ekskl. moms)

Afholdelsesgaranti på understregede datoer:

19–21/11 • 23–25/2 • 20–22/4

SU-093 jQuery – Det samlede client webudviklingsforløb

Du bruger jQuery, det centrale JavaScript framework på client-side mellem frontend-layout og JavaScript funktionalitet.

Varighed / Pris: 3 dage / 12.600,- kr. (ekskl. moms)

Afholdelsesgaranti på understregede datoer:

19–21/1 • 16–18/3

SU-905 Web App Programmering af Hardware / Services – Phonegap

Du laver Web Apps, der håndterer hardware/services: fx kontakter/kalender, GPS/maps, kamera/video, mikrofon, tlf, mail ...

Varighed / Pris: 3 dage / 13.500,- kr. (ekskl. moms)

Afholdelsesgaranti på understregede datoer:

19–21/11 • 14–16/1 • 11–13/3 • 4–6/5

SU-095 Node.js – Det samlede server webudviklingsforløb (inkl. Raspberry Pi computer)

Du lærer at bruge Node.js, et højperformance server-framework, som gør at al server-side webudvikling foregår med JavaScript.

Varighed / Pris: 3 dage / 13.500,- kr. (ekskl. moms)

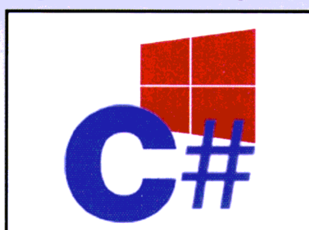
Afholdelsesgaranti på understregede datoer:

1–3/12 • 18–20/2

Bestil dine kurser nu, hvor du husker det, og mens der er plads

- **Via web:** superusers.dk
- **Via mail:** super@superusers.dk
- **Via telefon:** 48 28 07 06

Kurser i mobile platforme – se: www.superusers.dk/mobile-platforme.htm



Karlebogaard: Karlebovej 91 • 3400 Hillerød & **Kampehøjgaard**: Krajbjergvej 3, Vorre • 8541 Skødstrup
Tlf.: 48 28 07 06 • **Mail:** super@superusers.dk • **Web:** www.superusers.dk • **Alle priser er ekskl. moms**