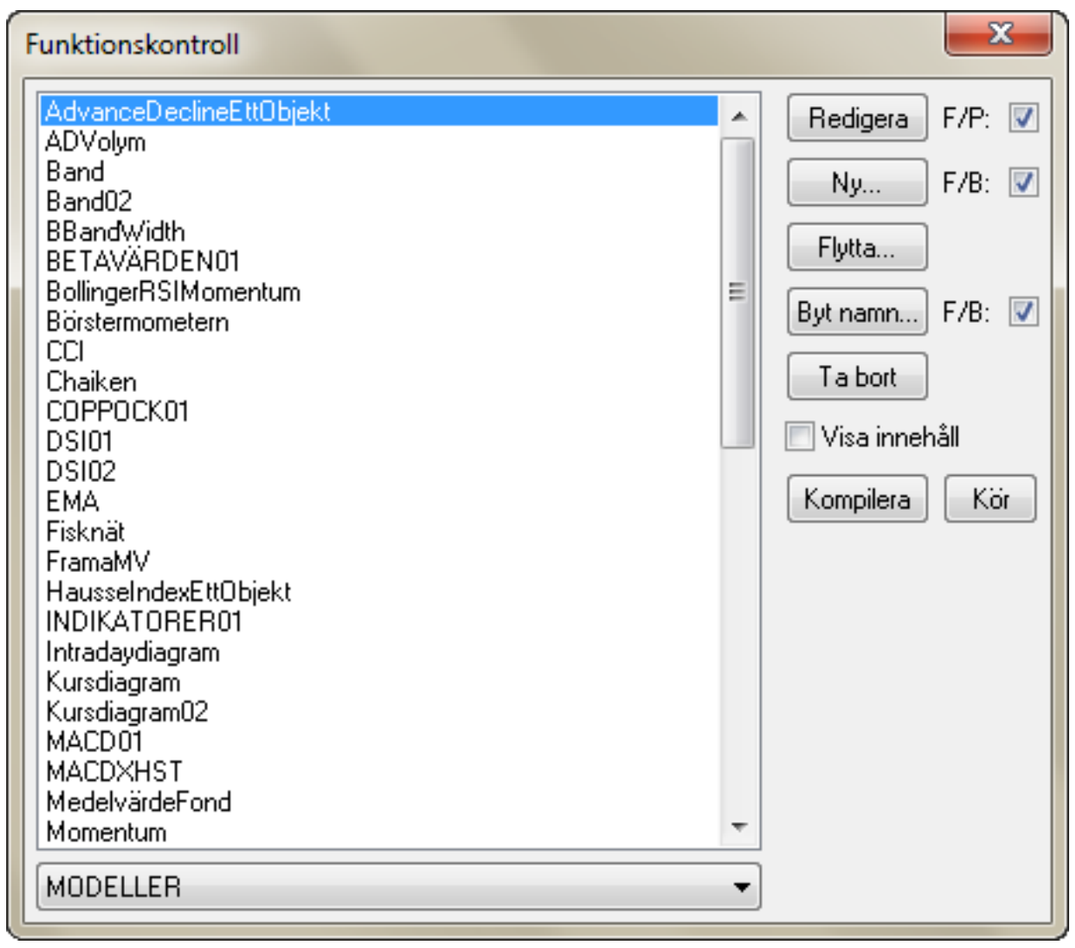
1. Inledning

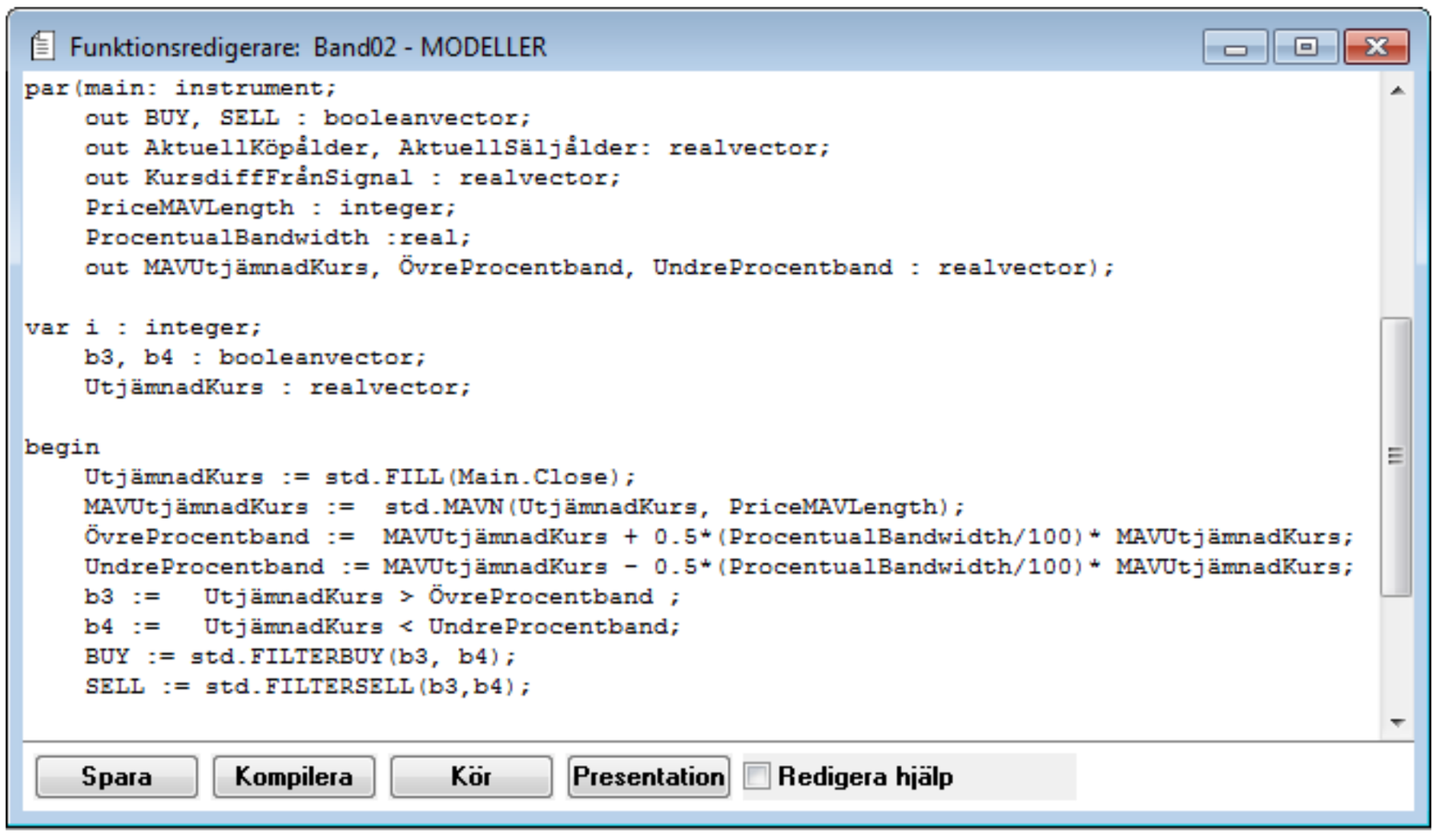


Futurelook är Vikingens eget modellspråk. Alla modeller i Vikingen är kodat i detta språk och du kan själv göra egna modeller i Vikingen med Future Look. Med modell avses diagram.

För alla som programmerat i C++, C#, Java, Pascal, Basic eller liknande programmeringsspråk, kommer det gå snabbt att sätta sig in i hur man utvecklar en egen analysmodell i Vikingen med hjälp av Futurelook.

I funktionsredigeraren får du möjlighet att skapa nya funktioner och redigera befintliga modellers källkod. När du byggt din analysmodell behöver du göra inställningar för hur diagram, tabeller och konfigurationsrutor skall se ut.

Detta gör att Futurelook är ett mycket flexibelt och kraftfullt redskap för att utveckla unika analaysmetoder för finansiella marknader.



Den här Onlineguiden är menad som en introduktion till programmeringsspråket Futurelook för användare utan tidigare programmeringsvana. Vi kommer att beröra alla nödvändiga begrepp och bygga upp guiden kring en mängd exempel för att göra inlärningen lättare för nybörjare. Uppdelningen av guiden är sådan att det finns 3 olika segment

Avsnitt 1, 2 Inledande om Futurelook och dess delar samt en första titt på programmering
Avsnitt 3,4,5 Informativa avsnitt rörande programmeringsspråket, beskrivning av verktygen som finns till förfogande
Avsnitt 6,7,8 Futurelook i praktiken, hur jobbar man med programmet för att ta en ide och göra den till en fullständig professionell modell.

b. Introduktion till Futurelook

Futurelook är ett mycket kraftfullt verktyg för finansanalytiker och kapitalplacerare. Med FutureLook är det möjligt att på ett enkelt sätt göra skapa agna algoritmer för att göra komplexa tidsserieanalyser och Teknisk Analys. FutureLook är uppbyggt som ett enkelt programmeringsspråk, med ett anpassat funktionsbibliotek lämpligt för analys av finansiella tidsserier. FutureLook är integrerat med Vikingen och sitter ihop med Vinstgeneratorn vilket gör det enkelt att vinsttesta algoritmerna.

[bild av menyn moduler]

Till det yttre består FutureLook av några filer som lagras i Vikingen. De är Modeller respektive Presentationer.

c. Introduktion till Futurelook – Funktionsredigeraren

Funktionsredigeraren är verktygen som man använder för att programmera sin Modell. Genom att starta Funktionskontrollen (’Moduler’ -> ’Funktionskontroll’) kan du bläddra genom det befintliga funktionsbiblioteket som finns i din Viking och skapa nya bibliotek eller funktioner. Funktionsredigeraren är ett enkelt textfönster som som används för att redigera och skapa egna modeller, kompilera dessa och skriva hjälptexter. Man kommer till Funktionsredigeraren antingen genom att skapa en ’Ny’ modeller eller genom att ’redigera’ en som redan finns.

Tips: genom att klicka i ’Visa innehåll’ kan man få se en förhandsvisning på funktionerna, vilket kan vara bra om man letar efter en bit kod att återanvända. Funktionsredigeraren har förutom själva textfältet knappar för att ’Spara’ funktionen efter man gjort förändringar, och ’Kompilera’ kod för att göra funktionen körbar i Vikingen. Man kan även använda ’Kör’-knappen för att snabbt starta funktionen med standardinställningar för att se att koden fungerar som man tänkt. Till slut finns en knapp ’Presentation’ som skickar en vidare till ’Presentationsredigeraren’ för att göra de grafiska inställningarna för modellen. Funktionsredigerar har två lägen, där ’Redigera hjälp’ kan användas för att redigera hjälpfilen instället för programkoden.

d. Introduktion till Futurelook – Presentationsredigeraren

När du i Vikingen öppnar "analytikern" och väljer en modell, så pekar du faktiskt på en presentation. Denna presentation innehåller information om vilken modell som används samt hur den ska presenteras och med vilka inställningar. Detta gör att man kan ha flera separata presentationer som använder sig av samma under liggande modeller. Utöver sin funktion som "interface" innehåller Presentationsmodellen även uppgifter om hur diagrammet eller tabellen ser ut samt vilka inställningar användaren kan göra som tex ändra tidsintervall, färger eller byta till logaritmisk skala. Presentationen innehåller även hjälpinformation, samt specifikationer för hur utdata skall hanteras i modelltabellen.

Grunderna i Futurelook

a. Programmens struktur, design

Funktioner i Futurelook har 3 delar som behöver vara kompletta för att man ska kunna kompilera programmet.

`Par();`	Parametrar: Hantering av indata och utdata. Definiera vilka indata som ska användas i funktionen, samt vilka dataserier som ska kunna tas ut ur modellen.
`Var`	Variabler: Definiera vilka variabler som ska användas lokalt i programmet, men inte som utdata.
`Begin & End;`	Start och slut för själva programkoden. Här kan du endast använda dig av objekt som definierats i segmenten ovan, och anropa funktioner som finns i Vikingens funktionsbibliotek.

b. Kommentarer

Olika programmeringsspråk använder sig av olika tecken för att signalera till kompilatorn att det som skrivs inte är programkod som den förstår, utan är menat till programmeraren. Text om varför kommenterad kod är viktig Futurelook använder sig av teckenkombinationerna // för att signalera att allting på samma rad är en kommentar, samt (* och *) för att signalera att allting som skrivs mellan tecknen är kommentarer.

Alltså:

```
// Detta är en kommentar!

(* Detta är en kommentar som sträcker sig över flera rader! *)
```

c. En första titt på: Parametrar

Det första segmentet i varje funktion i Futurelook är definitionen av parametrarna. Här specificerar du hur Vikingen skall hantera indata (ett objekt eller flera), vilka utdata som skall kunna tas ut ur funktionen (t.ex. i en modelltabell) samt vilka parametrar som skall vara tillgängliga i modellinställningar (antal perioder för MAV osv.) Stiliserat kan man säga att par-segmentet ska se ut som följande:

```
par( [flagga] identifierare : datatyp ; [flagga2] identifierare2 : datatyp2 ; .. ) : [returtyp] ;
```

Där [flagga] är en specifikation för hur Futurelook ska hantera datat:

`out :`	datat kan användas som utdata, t.ex. som graf i diagrammet eller som tabellkolumn i modelltabellen.
`var :`	datat används som en variabel, precis som om den definierats under var-segmentet ist. för par-segmentet. Skillnaden är att om man definierar den under par-segmentet med flaggan var är att den är synlig i presentationsredigeraren och således kan plockas fram senare om man vill.
`:`	avsaknad av flagga betyder att datat är någonting som ska in i modellen (tänk motsatsen till out)
`:returtyp`	genom att ange en datatyp efter : efter den avslutande parentesen, så definierar man vad som ska returneras av funktionen. Detta används då man ska anropa funktionen innuti en annan funktion, och kräver att man avslutar programmet med att definiera ’return [identifierare];’ innan den sista ’end;’ raden

Som exempel betyder out utvektor1 : realvector; att utvektor1 är namnet på en vektor där innehållet är av typen real och att denna vektor skall skickas ut ur modellen för att kunna användas grafiskt eller i modelltabellen.

d. En första titt på: Variabler

Variabler är data som du vill använda dig av inne i funktionen, men som inte behöver eller ska användas utanför. Detta kan vara data av typer integer, dvs. Heltal som används som räknare i en loop eller vektorer som används under tiden funktionen arbetar, men som inte behöver användas som utdata. Stiliserat är var-segmentet något enklare än par-segmentet och ser ut som följande:

```
var identifierare : datatyp ; ..
```

Notera att var segmentet inte avslutas med något speciellt tecken eller ord, utan alla deklarationer med syntaxen ovan som ligger efter var och innan begin, blir deklarerade variabler i programmet.

e. En första titt på: Funktioner

Alla funktioner i Vikingens funktionsbibliotek är tillgängliga att anropa i dina funktioner med syntaxen:

```
[biblioteksnamn].[funktionsnamn](parmeter1, parameter2, .. );
```

Som exempel kan man anropa funktionen ’MAVN’ som ligger i funktionsbiblioteket ’Std’ med anropet

```
std.MAVN(a,b);
```

vi vet att MAVN räknar ut ett naturligt medelvärde på en dataserie ’a’ av typen realvector sett över de senaste ’b’ perioderna. Vill vi ha ett 20-perioders medelvärde på

huvudobjektets slutkurs skriver vi således:

```
std.MAVN(main.close,20);
```

f. En första titt på: Operatorer

Många av de uttryck som man använder sig av i Futurelook kommer att innehålla operatorer. En operator är ett tecken som motsvarar en av en mängd olika saker, såsom ett matematiskt räknesätt eller en jämförelse mellan två saker. Operatorerna behöver ha två parametrar att arbeta med och kommer efter operationen att returnera ett värde. T.ex. kan man räkna ut indexkursen för S&P500 i SEK genom att multiplicera en vektor sp500 bestående av indexet med en vektor sekusd bestående av kronkursen.

```
SP500SEK := sp500 * sekusd;
```

Operatorn i detta fall är multiplikation *. Futurelook vet att både sp500 och sekusd är vektorer och ser till att resultatet av operationen även den är en vektor. Det är således upplagt för att visa upp SP500Sek på skärmen precis som man vill.

g. En första titt på: Ett exempelprogram

Vi har nu fått en första titt på de olika komponenterna som bygger upp ett program i Futurelook och kan därför börja fundera på helheten, hur ser en fullständig modell ut?

Som exempel så skriver vi en modell som räknar ut 2 olika glidande medelvärden på slutkurserna på en aktie och ett glidande medelvärde på volymen på samma aktie.

```
par(main : instrument; out MV1,MV2,MVVol : realvector; MAV1Length,MAV2Length,MAVVolLength : integer); var filledclose : realvector;   begin fi
```

Vi bryter ner koden för att undersöka alla delar efter vad vi lärt oss i det här avsnittet.

```
par(main : instrument;
```

Den första parametern som definieras är ett objekt av typen instrument, som vi kallar main. I en vanlig modell kommer denna rad alltid finnas, eftersom vi vill att modellen ska köras på ’Aktuellt Objekt’ i Vikingen

```
out MV1,MV2,MVVol : realvector;
```

Vi definierar även tre stycken vektorer som vi vill att ska kunna vara decimaltal, dvs realvector, och vi vill kunna se dem i chartet så vi anger prefixet out för dem.

```
MAV1Length,MAV2Length,MAVVolLength : integer);
```

Modellen ska använda sig av tre stycken parametrar som bestämmer hur modellen fungerar, nämligen de tre olika perioderna för de glidande medelvärdena. Genom att inte ange dem som out så säger vi indirekt att modellen kräver dessa tre som inparametrar. Glöm inte att den sista raden under par-segmentet ska avslutas med);

```
var filledclose : realvector;
```

I var-segmentet definierar vi en vektor som vi vill att ska kunna innehålla decimaltal dvs en realvector. Genom att definiera den under var-segmentet så kan vi inte rita upp vektorn i chartet, utan använder den bara inne i modellen

```
begin
```

Allting som står skrivet mellan begin och det sista end; är själva ritningen för vad modellen ska göra

```
filledclose := std.FILL(main.close);
```

Först använder vi standardfunktionen FILL för att ge vektorn filledclose alla slutkurser som finns i objektet main, och dessutom täppa till alla hål i vektorn om det skulle saknas data.

```
MV1 := std.MAVN(filledclose,MAV1Length);
```

Sedan tilldelar vi vektorn MV1 det glidande medelvärdet på filledclose med periodlängden MAV1Length

```
MV2 := std.MAVN(filledclose,MAV2Length);
```

därtill tilldelar vi vektorn MV2 det glidande medelvärdet på filledclose med den andra periodlängden MAV2Length

```
MVVol := std.MAVN(main.vol ,MAVVolLength);
```

och till sist tilldelar vi vektorn MVVOL värden som motsvarar det glidande medelvärdet på objektets volymvektor main.vol