Flödesstyrare
(sequencers)

- Förändrar det normala kontrollflödet.
- Hopp
- Escapes
- Undantag

Hopp

- Explicit överföring av kontroll
  - **Label** L
  - **Goto** L
- Utan restriktioner kan det ge hopplösa program.
  - Enbart existensen av hopp i programspråk medför att dess semantik blir mycket mera komplicerad.
  - Den vanliga tolkningen av t ex sekvens $C_1;C_2$ är inte längre självklar.
  - Labeln motsvarar inte enbart en punkt i programmet utan en punkt i en specifik aktivering av just det blocket.

Begränsade hopp

- Vanlig regel; får ej hoppa in i block.
- Får man hoppa in i sammansatta satser?
  -
```
begin
if E1 then C1 else begin C2; goto L end;
C3;
while E2 do
     begin
     C4;
   L:     C5
     end
end
```
  - OK i Algol 60, fel i Pascal.

Problem

- Hopp ut ut ett block (t ex en funktion): a-posten måste tas bort, alla lokala variabler spolas osv.
- Hopp ut ur rekursiva proc.: alla aktiveringar måste avslutas.
- Om destinationen är i en rekursiv proc: vilken aktivering?

Escapes

- Terminerar textuellt omslutande block.
- Ger oss *singel entry-multiexit.*
- **Ada:**
  - **exit** teminerar minsta omlutande loop men kan även avsluta namngiven loop
  - **return** ur procedurer eller funktioner (**return** E för funktioner)
  - **halt** ur programmet
- C, Java
  - **break**/**continue** bryter loop / bryter pågående itteration

Undantagshantering

- Ett program som inte klarar av händelser som avviker från det normalt förväntade är inte robust.
- Önskvärt att i programmet
  - kunna specificera vad som ska göras när vissa ”undantag” uppträder
  - kunna skilja detta från huvudalgoritmen

Centrala frågor:

- Hur deklararar man ett undantag?
- Vilken räckvidd har det?
- Hur aktiveras ett undantag?
- Hur definierar vi de enheter som ska aktiveras när ett undantag aktiveras?
- Hur binds ett aktiverat undantag till en hanterare?
- Var fortsätter exekveringen efter hanteringsrutinens slut?

Utan undantag

- Hantera med villkorssats

```
doA();
   if (doA went wrong) then
         handle the doA problem
   else
         doB();
         if (doB went wrong) then
               handle the doB problem
         else
               doC();
               if (doC went wrong) then
                     handle the doC problem
```

Med undantag

```
try {
   doA();
   doB();
   doC();
}
catch (doA-exception) {handle doA problem};
catch (doB-exception) {handle doB problem};
catch (doC-exception) {handle doC problem};
```

PL/I:

- Första språket med undantagshantering.

  ON CONDITION (name) *handler*
  - Deklarerar undantaget *name* och motsvarande hanterare.

  SIGNAL CONDITION (name)
  - Aktiverar hanteraren.
- Exekveringen fortsätter sedan efter SIGNAL-satsen!

Ada

- Inbyggda undantag
- Programdefinierade undantag
- Hanterare knyts till ett kommando (block)
- När ett undantag aktiveras avslutas kommandot, kontrollen går till hanteraren
- Om ingen hanterare finns propageras undantaget till omslutande kommando / anropande enhet.

```
procedure P is
    BAD_FORMAT: exception;
    procedure Q is
          begin
           :
          if S/=" " then raise BAD_FORMAT; end if
           :
          end Q
    procedure R is
          begin
                Q;
          exception when BAD_FORMAT=>handler body 1
          end R
begin
    R;
    Q;
exception when  BAD_FORMAT=>handler body 2
end P;
```

ML

- Hanterare knyts till uttryck

```
exception Prod
fun prod' [] = 1
  | prod' (0::_) = raise Prod
  | prod' (x::xs) = x * prod' xs

fun prod l = prod' l handle Prod => 0
```

- Går att returnera en parameter med undantag

```
exception SomeError of int;
  :
raise SomeError 42;
  :
handle SomeError x => ....
```

Continuations

- I funktionella språk används ofta *continuations* för att beskriva förändringar i kontrollflödet
- En continuation av ett uttryck är en abstraktion av vad systemet kommer att göra med uttryckets värde.
- Kan vara både ett programmeringssätt och inbygga funktioner.
- Continuations är ursprungligen ett sätt att beskriva generella kontrollstrukturer.

Exempel

```
fun prod [] = 1
  | prod (x::xs)  = x * prod xs

fun prod [] = 1
  | prod (0::xs)  = 0
  | prod (x::xs)  = x * prod xs

fun prod' [] c    = c 1
  | prod' (0::xs) = 0
  | prod' (x::xs) = prod' xs (fn s => s+x)

fun prod ys = prod' ys id
```