UPPSALA
UNIVERSITET

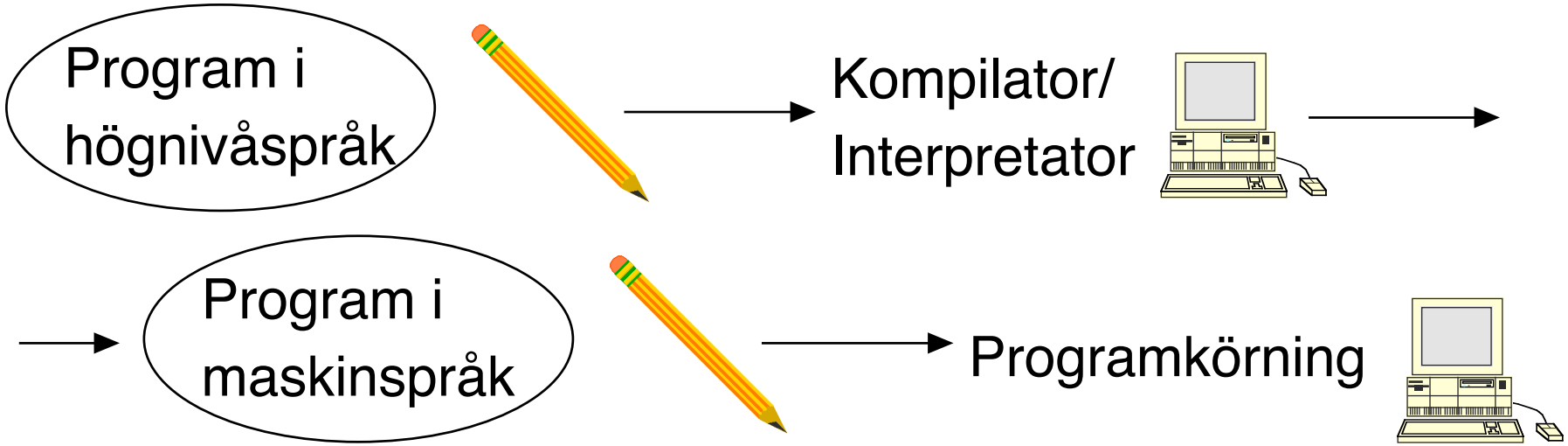
Programmeringsmetodik DV1
Programkonstruktion 1

Moment 1
Introduktion till programmering och programspråket Standard ML

Programspråk

Vid programmering används något av många olika *programspråk* för att beskriva algoritmer och datastrukturer.

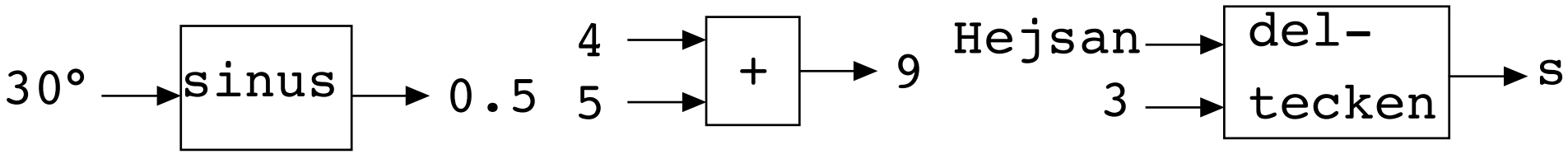
- C, C++, Java, ADA, ML, Pascal, BASIC, ...



Funktionell programmering

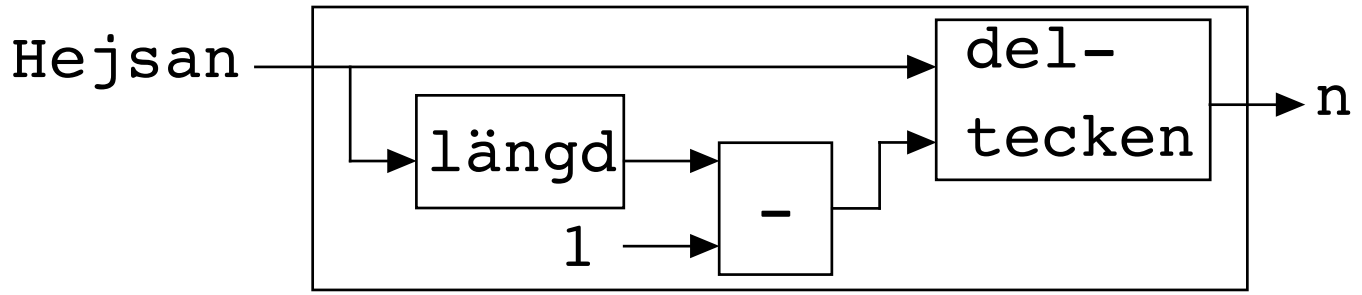
Programspråk skiljer i hur instruktioner är utformade och sätts ihop. Instruktionerna i funktionella programspråk är *funktioner*.

En funktion är ”något” som tar data (”argument”) och *beräknar* nya data (”värde”). Det är *inte* samma sak som funktioner i matematiken.



(Obs att datavetare ofta börjar räkna med 0....)

Funktionella program byggs genom *sammansättning* av funktioner.



Detta *dataflödesdiagram* visar beräkning av sista tecknet i en sträng

Användning av funktionell programmering

- Har funnits ”inom” datavetenskapen sedan lång tid (LISP 1962)
- Har fått växande använding även i ”praktiska” tillämpningar, t.ex.
  - Microsoft Excel (formelspråket är ett funktionellt språk)
  - Ericssons telefonsystem (programspråket Erlang)

Undersökningar har visat att program blir kortare och enklare och att utvecklingstiden blir kortare jämfört med ”traditionella” språk som C.

Funktionella språk har också en enklare *semantik* (betydelse).

Nackdelen är att programmen ibland utnyttjar datorn något mindre effektivt – spelar i dag mindre roll än effektivitet vid utvecklingen.

Inst. för informationsteknologi bedriver forskning kring funktionell programmering i samarbete med bl.a. Ericsson.

Programspråket Standard ML

- *Funktionellt* språk
- Programmen liknar (ytligt sett) matematiska *uttryck*.
- Få, enkla och tydliga begrepp – lämpligt för utbildning
- Enkel struktur hos program (i alla fall jämfört med många andra språk.....)
- Möjliggör snabb och effektiv programmering
- Den implementering av Standard ML vi använder på kursen kallas ”Moscow ML”.

Enkla ML-uttryck

Uttryck är ”formler” i språket ML som kan *beräknas* till ett *värde*.

ML kan arbeta med olika sorders data, bl.a.:

heltal:	`32+15`	beräknas till `47`
flyttal:	`3.12*4.3`	beräknas till `13.416`
text (*strängar*):	`size "Hej, hopp"`	beräknas till `9`
även sammansatta uttryck :	`1+(size "Hej, hopp"*2)`	beräknas till `19`

Värden kan inte beräknas vidare utan är data ”i sig själva”.

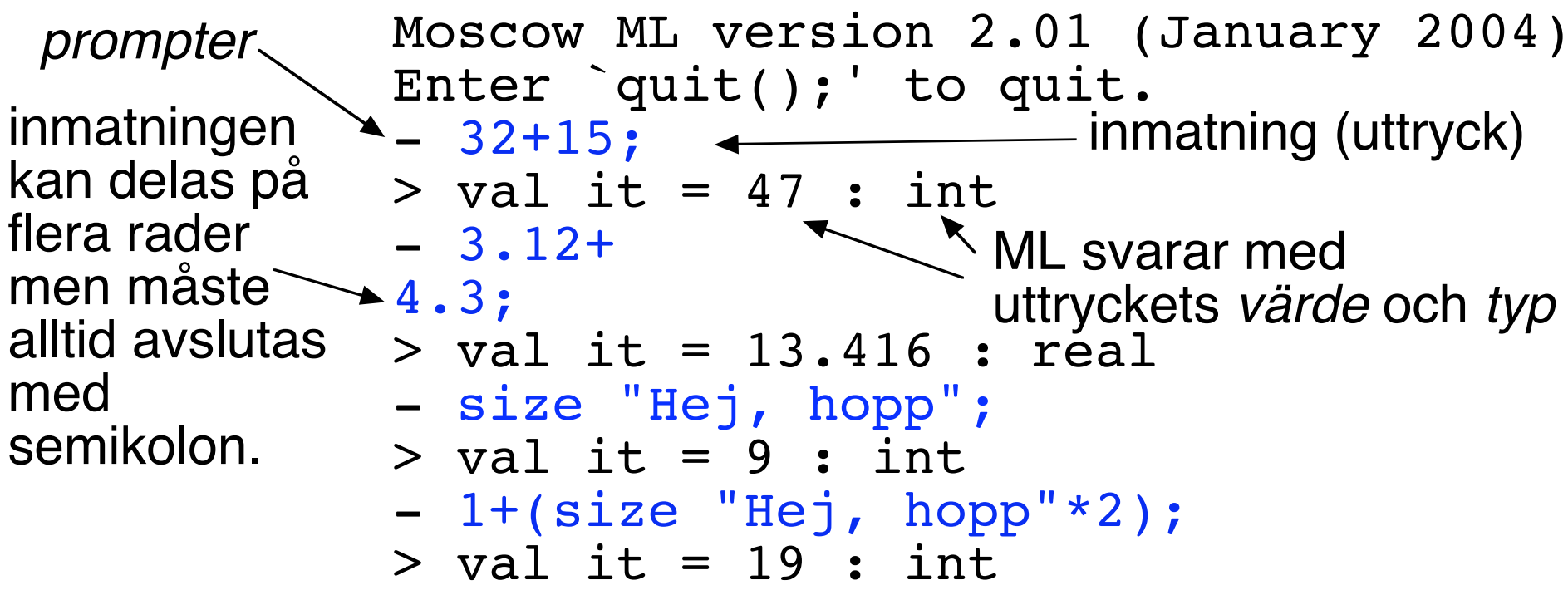
Funktionsvärden beräknas genom att man skriver funktionen (dess namn) före argumentet.

Vissa funktioner som i matematiken normalt skrivs *mellan* sina argument (t.ex. `+`, `*`) skrivs så även i ML. (*Infix* notation.)

Körning av Moscow ML-interpretatorn

På Unix-systemen startas Moscow ML med kommandot `mosml`.

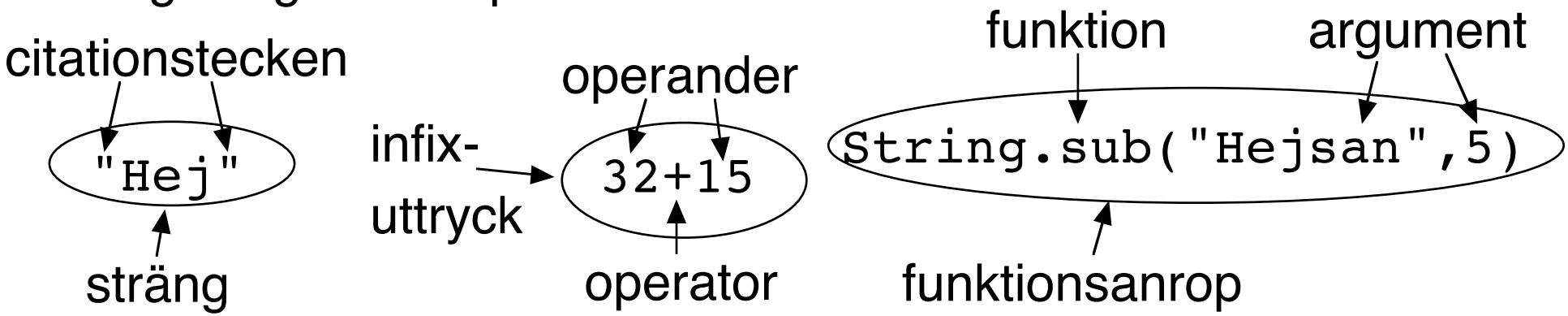
För tydlighet skrivs inmatning på OH-bilderna med blå färg.



Jag återkommer till vad en ”typ” är för något och vad `val it` betyder.

Syntax

Syntaxen hos ett programspråk är hur program *får se ut*.
Programspråk har normalt mycket strikt syntax som inte tillåter misstag. Några exempel:



Operatorer är funktioner som skrivs mellan sina argument (”infix”).

Argumenten kan själva vara uttryck, t.ex. `1+2*3`.
Argument, operander och *parametrar* är olika namn på samma sak.

Om funktionen har *ett* argument kan man i ML utelämna parenteserna – t.ex. `size "Hej"` eller `size("Hej")`.

Syntaxfel

Felaktig syntax gör att uttrycken inte kan tolkas.

ML-systemet ger ett *felmeddelande*.

```
- (1.0+ *3.0)/2.0;
! Toplevel input:
! (1.0+ *3.0)/2.0;
!  ^^^^^^^^^^
! Ill-formed infix expression
```

Meddelandena är inte alltid hjälpsamma – maskinen kan inte gissa vad man egentligen menar...

Ibland kan felskrivningen ligga på ett helt annat ställe än där ML-systemet misslyckas med att tolka ett uttryck.

Luring vid felaktig syntax

Felaktig syntax kan göra att ML-systemet verkar sluta fungera.

```
- 2.0*(3.0+4.0;
1+2;
size "Hej";
```

Här har man glömt en högerparentes! ML väntar förgäves på den och verkar därför ha slutat svara. I detta läge kan man avbryta inmatningen med ”kontroll-C” från tangentbordet och skriva in rätt.

```
^C> Interrupted.
- 2.0*(3.0+4.0);
val it = 14.0 : real
```

Glömmer man ett citationstecken får man liknande effekt...

```
- size "Hej, hopp;
1+2;
```

Semantik

Semantiken hos ett programspråk är vad program *betyder*.

Betydelsen av...

`+`	är en funktion som beräknar summan av operanderna
`size`	är en funktion som beräknar längden av argumentet
f X	är att funktionen f beräknas med argumentet X
X op Y	är att funktionen op beräknas med argumenten X och Y

alltså...

`32+15` betyder ”summan av 32 och 15”, vilket beräknas till 47.

`size "Hej, hopp"` betyder ”längden av strängen `Hej, hopp`”, vilket beräknas till 9.

Beräkning av uttryck

Uttryck i ML beräknas i steg från vänster till höger, inifrån och ut – dvs argument beräknas innan funktionen beräknas (eller *anropas*).

Undantag: operatorer kan ha olika prioritet (*precedens*), t.ex. beräknas multiplikationer före additioner.
Parenteser kan användas för att ändra beräkningsordningen.

```
1+size("Hej, " ^ "hopp")*2
—> 1+size "Hej, hopp"*2
—> 1+9*2
—> 1+18
—> 19
```

Blåmarkerade deluttryck är de som beräknas i nästa steg.
(—> är ingen symbol i språket ML utan visar beräkningssteg.)

Beräkningsfel

Alla uttryck går inte att beräkna även om syntaxen är riktig, t.ex. `1.0/0.0` eftersom division med noll inte är meningsfull. I så fall *avbryts* beräkningen av uttrycket och man får en felmeddelande med *felkod* istället för värde.

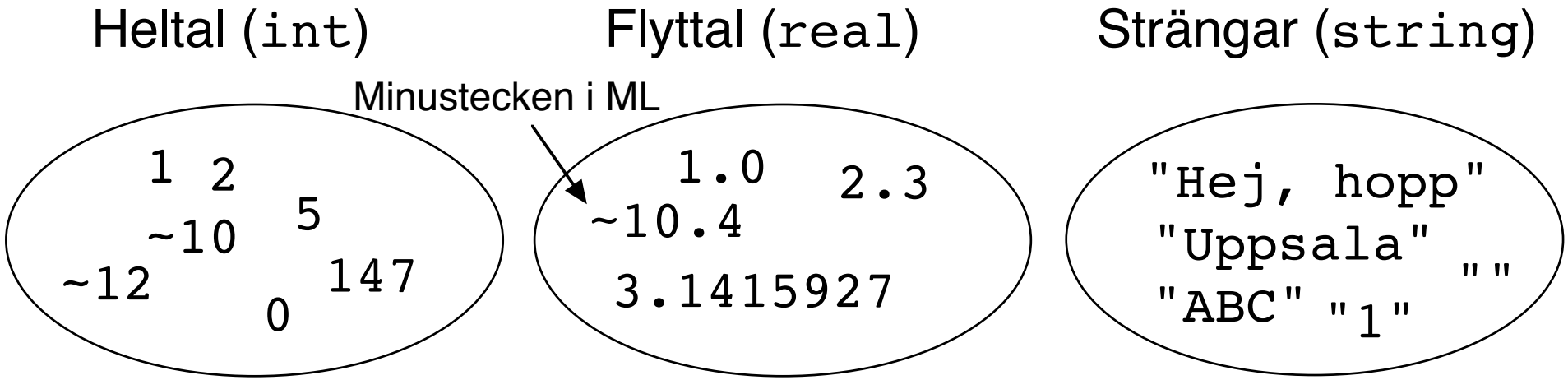
```
- (1.0 / 0.0) + 2.0;
! Uncaught exception:
! Div
```

I ML är felkoderna en speciell sorts data, kallade *undantag* (exceptions).

Beräkningsfel kallas ofta *exekveringsfel*.

Typer

Typer är mängder av data med samma slags egenskaper.
Elementen i en typ kallas typens *värden*.



ML-notation: `1:int` `1.0:real` `"1":string`

Obs! `1`, `1.0` och `"1"` är *alla olika*. Varför är `1` och `1.0` olika?
Jo, flyttal är *approximationer* med begränsad noggranhet!
1.0 + 10000000000000000.0 – 10000000000000000.0 blir 0.0 ?!?
(Representera *aldrig* ”riktiga” pengar som flyttal.)

Storleksbegränsningar

I praktiken finns det begränsningar i hur stora (små) tal kan vara och hur långa strängar kan vara. Det är sällan detta leder till problem, men man måste vara medveten om det – det kan leda till ”*overflow*”.

Begränsningarna kan variera mellan olika datorer/ML-system. Vanliga gränser för heltal är 1073741823 och –1073741824.

```
- 1073741822+1;
> val it = 1073741823 : int
- 1073741823+1;
! Uncaught exception:
! Overflow
```

Flyttal kan vara mycket större (i Moscow ML upp till $\pm 10^{307}$), men har begränsad noggranhet (i Moscow ML c:a 12 decimala siffror).

Typkontroll (typning)

Vad betyder `32+1.5`? `32+"Hej"`? `32+"1"`?

Att addera värden av olika typ är inte alltid meningsfullt.

Stark typning: Operationen *förbjuds* alltid.

Svag typning: Värden *görs om* till rätt typ *om det går*

t.ex. `32+"1"` —> `33`. `32+"Hej"` ger beräkningsfel.

Ingen typning: Additionen utförs *som om* argumenten hade rätt typ.

t.ex. `32+"1"` —> `81` om kodningen av tecknet `"1"` är 49!!

(I detta fall är det oftast beroende på vilken dator eller ML-system man använder vad resultatet blir.)

Stark typning gör att många programmeringsmisstag förhindras.

Svag/ingen typning *kan* ge kortare men även mer svårlästa program

ML är ett *starkt typat språk*. Alla uttryck har en *bestämd typ*.

Typfel

Liksom felaktig syntax gör typfel att uttrycken inte kan tolkas.

ML-systemet ger ett felmeddelande.

```
- 32+"Hej";
! Toplevel input:
! 32+"Hej";
!    ^^^^^
! Type clash: expression of type
!   string
! cannot have type
!   int
```

Detta betyder att ett stränguttryck (`"Hej"`) använts i ett sammanhang där det krävs ett heltalsuttryck.

Typfel (forts.)

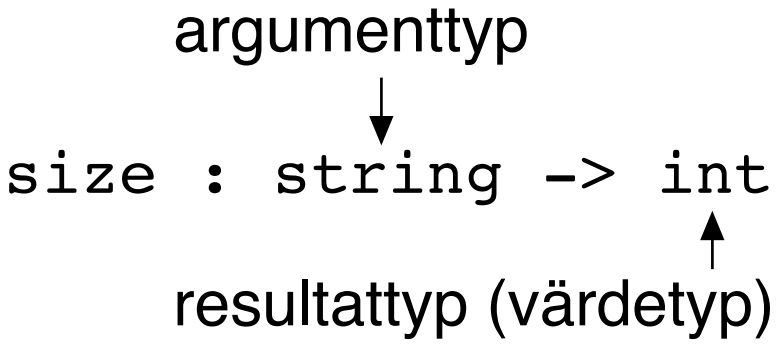
Meddelandena är inte alltid hjälpsamma – ML talar om *var den hittar* ett typfel, vilket kan vara ett helt annat ställe i programmet än där fel egentligen ligger!.

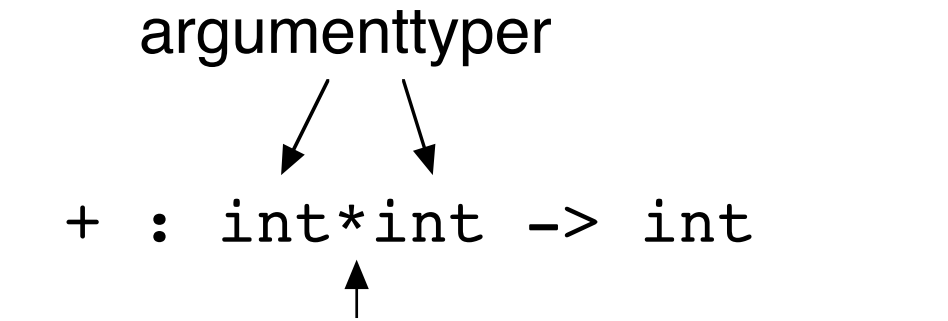
```
- 2*(3.0+4.0);
! Toplevel input:
! 2*(3.0+4.0);
!     ^^^
! Type clash: expression of type
!   real
! cannot have type
!   int
```

Felet här är att man skrivit `2` (ett heltal) i stället för `2.0` (ett flyttal), men ML upptäcker felet först när den tittar på flyttalet `3.0` och säger därför att felet är att `3.0` är ett flyttal där det borde vara ett heltal!

Funktionstyper

Också funktioner har typer:





+ kan även typas `real*real->real` + sägs vara *överlagrad*

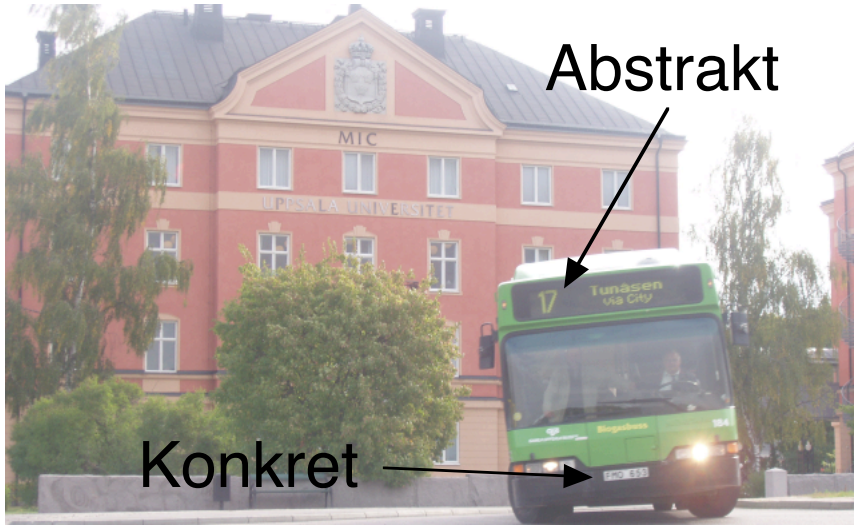
(Egentligen är addition av heltal och flyttal helt *olika* funktioner som har samma namn. Argumenttyperna styr vilken som används.)

Abstrakt och konkret

Med abstraktion i programmering menas att man *döljer* eller *skjuter åt sidan* vissa egenskaper hos något. Detta gör man för att förenkla programmeringen så att man inte skall behöva tänka på detaljer.

Du vill åka buss från Polacksbacken till Svartbäcken. Då behöver du inte veta vilken buss (alltså vilket fordon) du skall åka med – vilket kan vara olika från gång till gång – utan du åker med ”busslinje 17”.

”Linje 17” är en *abstraktion* av de bussar som faktiskt kör mellan (bl.a.) Polacksbacken och Svartbäcken.



Abstrakt och konkret (forts.)

I programmering abstraherar man data genom att dölja hur data *representeras*.

- strängen `"Hej, hopp"` representeras *egentligen* som en följd av teckenkoder, t.ex. (med ISO 8859-1 kod, mera senare) 72, 101, 106, 44, 32, 104, 111, 112, 112
- flyttalet `3.12` representeras *egentligen* som två tal, en *mantissa* och en *exponent*, t.ex. 312 och –2, där innebörden är $312*10^{-2}$

Hur data *verkligen* ser ut döljs av programspråket.
Är mantissan och exponenten 312 resp. –2 eller är de 0.312 och 1?

Det spelar ingen roll – det viktiga är hur man kan *arbeta* med dem.
(Den konkreta) representationen har *abstraherats bort*.

Bindningar

Man kan namnge värden – *binda* namn till värdet.

```
- val a = 3;
> val a = 3 : int
- a;
> val it = 3 : int
- a+1;
> val it = 4 : int
```

inmatning (deklaration)

ML svarar med bundet namn, värde och typ

Namnet `it` binds alltid till det senast beräknade värdet.

```
- it+1;
> val it = 5 : int
- val b = 5;
> val b = 5 : int
- a+b;
> val it = 8 : int
```

Bindningar är *inte* tilldelningar som i C, Java,... (Mer om det senare.)

Definitioner

Bindning kan ses som att namnet *definieras* att betyda ett värde.

Detta är en form av abstraktion som kallas *definitionsabstraktion*. Man bortser från det ”konkreta” värdet och använder istället ett ”abstrakt” namn, i syfte att…

- ge enkla/meningsfulla namn åt värden
  (t.ex. `Math.pi` i stället för 3.14159265… underlättar läsning.)
- visa på en speciell användning av ett uttryck
  (t.ex. `dagarPerVecka` eller `maxAntal` i stället för 7. Man kan lägg ändra värdet i en användning utan att riskera påverka andra.)
- undvika dubbla beräkningar (t.ex. `val a = 5*5; a*a;` beräknar 5*5*5*5 med bara två multiplikationer)

Symbolerna som utgör namn kallas *identifierare*.

(Rosens) namn

What's in a name? That which we call a rose
by any other name would smell as sweet.

—W. Shakespeare, Romeo och Julia

Namnet som sådant saknar i de flesta programspråk betydelse. Det är bara den mänskliga läsaren av ett program som kan tolka namnet. Därför är det viktigt att ett tolkningen av ett namn faktiskt stämmer överens med innebörden av det som namnet används till.

Man kan byta ut alla namn inom ett program (om det görs på ett konsekvent sätt) utan att påverka programmets funktion alls.

(Detta gäller förstås inte namn som är definierade utanför själva programmet – t.ex. `Math.pi` som är definierat i språket ML.)

Uttryck och deklarationer

Uttryck är delar av språket ML som beräknar ett värde.

Deklarationer är delar av språket ML som inte beräknar något värde, men som *påverkar* hur uttryck beräknas (tolkas).

```
- xyz;
! Toplevel input:
! xyz;
! ^^^
! Unbound value identifier: xyz
- val xyz = 4711;
> val xyz = 4711 : int
- xyz;
> val it = 4711 : int
```

Uttrycket kan inte beräknas

Deklarationen binder `xyz`

Nu kan samma uttryck beräknas

En deklaration kan dock innehålla ett uttryck som del (t.ex. `val`-deklarationer innehåller uttryck som beräknar värdet som binds). Likaså kan uttryck innehålla deklarationer som delar. (Mera senare.)

Syntax för identifierare i ML

- *Alfanumeriska identifierare* – en följd av bokstäver, siffror, understrykning och apostrof som börjar med en bokstav:

  Ex: `x`, `X`, `x2`, `x'`, `size`, `total_sum`

  Obs att stora och små bokstäver skiljer sig åt.
  Endast engelska bokstäver är tillåtna – inte t.ex. `åäöÅÄÖ`.

- *Symboliska identifierare* – en följd av tecknen

  `! % & $ # + - / : < = > ? @ \ ~ ` ^ | *`

  Ex: `+`, `*`, `:`, `++`, `!?`, `>>`

- *Reserverade ord* eller *nyckelord* (ord som kännetecknar eller delar upp konstruktioner i språket) får inte användas som identifierare.
  Ex: `val`, `if`, `then`, `:`, `=` (se kursbok eller ML-dokumentation)

Formattering

ML *tillåter* att blanka och radbrytningar används var som helst i program mellan olika symboler.

ML *kräver inte* blanka och radbrytningar *utom* för att undvika att namn läses ihop (`size x` kan inte skrivas `sizex`).

Varning! Även namn bestående av specialtecken kan läsas ihop! `3+~2` fungerar inte. ML tolkar `+~` som *en* identifierare som den inte känner till. Skriv `3+ ~2`.

I ML-program kan man lägga in *kommentarer* – förklarande text som inte ingår i beräkningen. Kommentarer inleds med `(*` och avslutas med `*)`. T.ex.: `1+(* Längden på strängen *) size "abc"`

Villkorsuttryck

Program kan välja mellan två olika uttryck beroende på ett *villkor*.

`if` b `then` e1 `else` e2

Om villkoret b är uppfyllt beräknar villkorsuttrycket e1, annars e2.

b skall beräkna ett värde av typ `bool` – ett *sanningsvärde*.

`10>20` ⟶ `false`, `1<2` ⟶ `true`.

```
- 10 > 20;
> val it = false : bool
- if 10 > 20 then 10 else 20;
> val it = 20 : int
```

Obs att e1 och e2 måste ha samma typ! (Varför?)

Villkorsuttryck (forts.)

`if b then e1 else e2`

b beräknas *först* och bara *ett* av e1 och e2 beräknas

```
- val x = 0.0;
> val x = 0.0 : real
- if x < 0.01 then 100.0 else 1.0/x;
> val it = 100.0 : real
```

`1.0/x` beräknas inte och det uppstår därför inget exekveringsfel.

```
if x < 0.01 then 100.0 else 1.0/x
--> if 0.0 < 0.01 then 100.0 else 1.0/x
--> if true then 100.0 else 1.0/x
--> 100.0
```

Andra typer

Andra grundläggande typer i ML:

`char`

Enstaka *tecken*. Syntax: `#"1"` för t.ex. tecknet `1`.

`unit`

Trivial typ. `unit` har *ett enda* värde som skrivs `()`.

`()` används när syntaxen för språket ML kräver ett uttryck, men man inte är intresserad av något.

Typen int (heltal)

Syntax: en eller flera siffor. Negativa tal föregås av ~ (”tilde”).
Exempel: `1`, `~25`

Funktioner	Typ	
`+`, `-`, `*`, `div`	`int*int->int`	(de fyra räknesätten, heltalsdivision – `7 div 3`—> `2`)
`mod`	`int*int->int`	(rest – `7 mod 3`—> `1`)
`=`, `<>`	`int*int->bool`	(lika med, inte lika med)
`<`, `<=`, `>`, `>=`	`int*int->bool`	(storleksjämförelser)
`~`	`int->int`	(negation – `~(2-3)`—> `1`)
`abs`	`int->int`	(absolutbelopp `abs ~3`—> `3`)

(alla tvåställiga funktioner på denna sida är infixa)

Typen real (flyttal)

Syntax: en eller flera siffor med decimalpunkt och/eller följd av tiopotens (bokstaven `E` följd av siffror) Negativa tal föregås av `~`. Termen flyttal (floating point) kommer sig av att decimalpunkten inte står på ett bestämt ställe i talet.

Exempel: `1.0`, `3E4` (=`3000.0`), `3.4E~2` (=`0.034`)

Grundläggande operationer på flyttal:

Funktioner	Typ	
`+`, `-`, `*`, `/`	`real*real->real`	(de fyra räknesätten)
`<`, `<=`, `>`, `>=`	`real*real->bool`	(storleksjämförelser)
`~`	`real->real`	(negation)
`abs`	`real->real`	(absolutbelopp)

(alla tvåställiga funktioner på denna sida är infixa)

Likhetstyper

Värden av de flesta typer kan jämföras med = och <>.
Detta gäller *inte* flyttal i Standard ML.

Anledningen är att flyttal är approximationer och strikta likhetsjämförelser i allmänhet inte är rätt sak att göra.

`1.0+10000000000000000.0-10000000000000000.0=1.0` ?!?

Det ML-system som vi använder (Moscow ML) tillåter likhets-jämförelser mellan flyttal, men det är ett (riskabelt!) *tillägg* till ML.

Här finns alltså dessutom:

Funktioner	Typ
=, <>	`real*real->bool` (lika med, inte lika med)

Typer som kan likhetsjämföras kallas *likhetstyper*.

Biblioteket Math

Det finns *programbibliotek* med färdigskrivna funktioner som man kan ha användning för. Ett sådant bibliotek är `Math`.

Ett urval funktioner:

Funktion	Typ	
`Math.sqrt`	`real->real`	(kvadratrot)
`Math.sin`	`real->real`	(sinus – vinkeln i radianer)
`Math.cos`	`real->real`	(cosinus – vinkeln i radianer)
`Math.tan`	`real->real`	(tangens – vinkeln i radianer)
`Math.ln`	`real->real`	(naturliga logaritmen)
`Math.pow`	`real*real->real`	(exponentiering)

```
Math.pow(10.0,3.0) ---> 1000.0
```

Laddning av bibliotek

De flesta bibliotek finns inte tillgängliga från början utan ligger på filer som måste läsas in (*laddas*) innan de kan användas.

Bibliotek laddas med funktionen `load`.

```
- Math.sqrt(4.0);
! Toplevel input:
! Math.sqrt(4.0);
! ^^^^^^^^^^
! Cannot access unit Math before it has been
loaded.
- load "Math";
> val it = () : unit
- Math.sqrt(4.0);
> val it = 2.0 : real
```

Typen string (strängar)

Syntax: En teckenföljd med citationstecken (") omkring.

Funktioner	Typ	
`=, <>`	`string*string->bool`	(lika med, inte lika med)
`<, <=, >, >=`	`string*string->bool`	(storleksjämförelser)
`^`	`string*string->string`	(sammansättning)
`size`	`string->int`	(längd)
`String.substring`	`string*int*int->string`	(delsträng)

(alla tvåställiga funktioner är infixa)

```
"Hej, " ^ "hopp" ---> "Hej, hopp"
String.substring("abcd",1,2) ---> "bc"
```

Citrationstecken *i* strängen skrivs `\"`. Bakvänt snedstreck skrivs `\\`.

Exempel: `"ab\"cd\\"` är en sträng med 6 tecken: `a`,`b`,`"`,`c`,`d` och `\`.

Typen char (tecken)

Syntax: Tecknet # följt av en sträng med exakt ett tecken.

Grundläggande operationer på tecken:

Funktioner	Typ	
`=, <>`	`char*char->bool`	(lika med, inte lika med)
`<, <=, >, >=`	`char*char->bool`	(storleksjämförelser)
`String.sub`	`string*int->char`	(tecken ur sträng)

```
String.sub("abcd",1) ---> #"b"
```

Typen bool (sanningsvärden)

Syntax: `true` respektive `false`

Observera: `true` och `false` är *data* i typen `bool`.

De är *konstanta värden* – ungefär som 1, 3.0 och `"Hej"`

Funktioner med värdetyp `bool` kallas ibland *predikat*.

Funktioner	Typ	
`=, <>`	`bool*bool->bool`	(lika med, inte lika med)
`not`	`bool->bool`	(logisk negation – `true` om argumentet är `false` och tvärtom.)
`orelse`	`bool*bool->bool`	(logisk eller – `true` om endera argument är `true`, `false` annars.)
`andalso`	`bool*bool->bool`	(logisk och – `true` om båda argument är `true`, `false` annars.)

`orelse` och `andalso` igen

```
1 = 2 orelse not 3 = 4
--> false orelse not 3 = 4
--> false orelse not false
--> false orelse true
--> true
```

`orelse` och `andalso` är inte vanliga funktioner – deras andra argument beräknas *inte* om det inte behövs. Man säger att de är *lata* i sitt andra argument. (Motsatsen till lat beräkning – alltså det normala att alltid beräkna argument – kallas för *ivrig* beräkning.)

`3>2 orelse 3.0<1.0/0.0` ger *inte* exekveringsfel trots divisionen med noll.

```
3>2 orelse 3.0<1.0/0.0
--> true orelse 3.0<1.0/0.0
--> true
```

Typomvandlingar

Det finns funktioner som vid behov kan byta typ på data. Exempel:

Funktioner	Typ	
`real`	`int->real`	(från heltal till flyttal)
`round`	`real->int`	(från flyttal till heltal, avrundat)
`trunc`	`real->int`	(samma, men decimaler kastas)
`str`	`char->string`	(från tecken till sträng)
`ord`	`char->int`	(från tecken till teckenkod)
`chr`	`int->char`	(från teckenkod till tecken)
`Int.toString`	`int->string`	(från heltal till sträng)

```
trunc 4.7 ==> 4
ord #"1" ==> 49
Int.toString 42 ==> "42"
```

Teckenkodning

Det finns ingen bestämd teckenkodning för ML utan det beror på vilken dator och operativsystem man använder. Funktionerna `ord` och `chr` kan alltså fungera olika på olika datorer/operativsystem.

Vid användning av `ord` och `chr` måste man alltså vara försiktig om programmet skall vara *portabelt* (dvs gå att flytta till annan dator).

En av de vanligaste teckenkodningarna är en internationell standard som kallas ISO 8859-1 och är avsedd för västeuropeiska språk. Den används av de flesta Unix-system samt är standardkodning på World Wide Web. ISO 8859-1 är en utökning av den vanliga äldre koden ASCII (American Standard Code for Information Interchange)

Apple MacOS och Microsoft Windows har egna, ej standardiserade, teckenkodningar som också bygger på ASCII.