ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Függvények alkalmazása az SML-ben

- Az SML-ben az `f` és az `e` tetszőleges *név* lehet, amelyeket megfelelően *szeparálni* kell egymástól: `f e`, vagy `f(e)`, vagy `(f)e`
- Szeparátor: nulla, egy vagy több *formázó* karakter (␣, \t, \n stb.). Nulla db formázó karakter elegendő pl. a (előtt és a) után.
- FONTOS! A szeparátor a legerősebb balra kötő infix operátor (!) az SML-ben.
- Példák:

```
Math.sin 1.00  (Math.cos)Math.pi   round(3.17)
2 + 3          (real) (3 + 2 * 5)
```

- Függvények egy csoportosítása az SML-ben
  - Beépített függvények, pl. `+`, `*` (mindkettő infix), `real`, `round` (mindkettő prefix)
  - Könyvtári függvények, pl. `Math.sin`, `Math.cos`, `Math.pi` (0 argumentumú!)
  - Felhasználó által definiálható függvények, pl. `square`, `/\`, `head`

A függvényalkalmazás kiértékelése

Egy felhasználói függvényeket tartalmazó kifejezést az összetett kifejezéshez hasonlóan értékel ki az SML. Ott hallgatólagosan feltettük, hogy az SML tudja, hogyan alkalmazza a belső függvényeket az aktuális paramétereikre.
Ezt most azzal egészítjük ki, hogy az SML egy függvény alkalmazásakor

- a függvény törzsében a formális paraméterek összes előfordulását lecseréli a megfelelő aktuális paraméterre, majd kiértékeli a függvény törzsét.

Nézzük az `f 5` kifejezés kiértékelését! Minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk.

```
f 5 → sumOfSquares(5+1, 5*2) → sumOfSquares(6, 5*2) →
sumOfSquares(6, 10) → square 6 + square 10 → 6*6 + square 10 →
36 + square 10 → 36 + 10*10 → 36 + 100 → 136
```

(`val sumOfSquares = fn (x, y) => square x + square y; val square = fn x => x * x`)
A függvényalkalmazás itt bemutatott *helyettesítési modellje* segíti a függvényalkalmazás *jelentésének* megértését. (Olyan esetekben alkalmazható, amikor egy függvény jelentése független a környezetétől.)
Az értelmezők/fordítók rendszerint más, bonyolultabb modell szerint működnek.

Applikatív sorrend, normál sorrend

Az összetett kifejezés kiértékelésénél leírtak szerint az SML először kiértékeli az operátort és argumentumait, majd alkalmazza az operátort az argumentumokra. Ezt a kiértékelési sorrendet *applikatív sorrendű* vagy *mohó* kiértékelésnek nevezzük.
Van más lehetőség is: a kiértékelést addig halogatjuk, ameddig csak lehetséges. Ezt *normál sorrendű*, *szükség szerinti* vagy *lusta* kiértékelésnek nevezzük.
Nézzük `f 5` kiértékelését, ha az SML lusta kiértékelést alkalmazna:

```
f 5 → sumOfSquares(5+1, 5*2) → square(5+1) + square(5*2) →
(5+1)*(5+1) + (5*2)*(5*2) → 6*(5+1) + (5*2)*(5*2) → 6*6 +
(5*2)*(5*2) → 36 + (5*2)*(5*2) → 36 + 10*(5*2) → 36 + 10*10 → 36
+ 100 → 136
```

Igazolható, hogy olyan függvények esetén, amelyek jelentésének megértésére a helyettesítési modell alkalmas, a kétféle kiértékelési sorrend azonos eredményt ad.
De vegyük észre, hogy szükség szerinti kiértékelés mellett a példában egyes részkifejezéseket kétszer kellett kiértékelni.
Ezen jobb értelmezők/fordítók úgy segítenek, hogy az azonos részkifejezéseket megjelölik, és amikor egy részkifejezést először kiértékelnek, az eredményét megjegyzik, a többi előfordulásakor pedig ezt az eredményt veszik elő. E módszer hátránya a nyilvántartás bonyolultsága.

Feltételes kifejezések, logikai műveletek, predikátumok

Először lássunk néhány példát!

```
val absolute = fn x => if      x < 0 then ~x
                       else if x > 0 then x
                       else               0

val absolute = fn x => if      x < 0 then ~x
                       else               x

use "sumOfSquares.sml";
val sumOfSquaresOfTwoLarger =
      fn (x,y,z) =>
           if      x < y andalso x < z then sumOfSquares(y, z)
           else if y < x andalso y < z then sumOfSquares(x, z)
           else                             sumOfSquares(x, y);
```

Predikátumnak az olyan függvényt nevezzük, amelynek logikai (`bool` típusú) érték az eredménye, pl.

```
val isAlphaNum = fn c =>
      #"A" <= c andalso c <= #"Z" orelse
      #"a" <= c andalso c <= #"z" orelse
      #"0" <= c andalso c <= #"9"
```

Feltételes kifejezések, logikai műveletek, predikátumok (folyt.)

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- *Lusta kiértékelésű* beépített műveletek (speciális nyelvi elemek!)
  - Három argumentumú: `if b then e1 else e2`.
    Nem értékeli ki az `e2`-t, ha a `b` igaz, ill. az `e1`-et, ha a `b` hamis.
  - Két argumentumúak:
    `e1 andalso e2` : nem értékeli ki az `e2`-t, ha az `e1` hamis.
    `e1 orelse e2` : nem értékeli ki az `e2`-t, ha az `e1` igaz.
- *Lusta kiértékelés* = kiértékelés szükség szerint *(call-by-need)*: csak akkor értékeli ki az SML, amikor szükség van rá.
- Mind a három logikai művelet csupán szintaktikus édesítőszer!
  - `if b then e1 else e2` ≡ `(fn true => e1 | false => e2) b`
  - `e1 andalso e2` ≡ `(fn true => e2 | false => false) e1`
  - `e1 orelse e2` ≡ `(fn true => true | false => e2) e1`
- Tipikus hiba: `if exp then true else false` !!!

Feltételes kifejezések, logikai műveletek, predikátumok (folyt.)

- Nyilvánvaló: `andalso` és `orelse` kifejezhető `if-then-else`-szel is.
  - `if e1 then e2 else false` ≡ `e1 andalso e2`
  - `if e1 then true else e2` ≡ `e1 orelse e2`
- Használjuk az `andalso`-t és az `orelse`-t az `if-then-else` helyett, ahol csak lehet: olvashatóbb lesz a program.
- Lusta kiértékelésű függvényt a programozó nem definiálhat az SML-ben. Az SML, mielőtt egy függvényt alkalmazna az (egyszerű vagy összetett) argumentumára, kiértékeli.
- Az `andalso` és az `orelse` *mohó kiértékelésű* megfelelői:

```
(* && (a, b) = a /\ b
   && : bool * bool -> bool
*)
fun op&& (a, b) = a andalso b;
infix 2 &&
```

```
(* || (a, b) = a \/ b
   || : bool * bool -> bool
*)
fun op|| (a, b) = a orelse b;
infix 1 ||
```

- `infix` *prec név1 név2* `...` : a *név1 név2* `...` függvényeket *prec* precedenciaszintű, *infix* helyzetű, balra kötő *operátorrá* alakítja.

Négyzetgyökvonás Newton-módszerrel

- A funkcionális nyelvi függvények sokban hasonlítanak a matematikai függvényekhez: egy vagy több argumentumtól függő értéket adnak eredményül. Egy dologban azonban mindenképpen különböznek: a funkcionális nyelvi függvényeknek hatékonyaknak is kell lenniük.
- Nézzük pl. a négyzetgyök következő definícióját: $\sqrt{x} = y$, ahol $y \geq 0$ és $y^2 = x$.
- Ez az egyenletrendszer alkalmas pl. annak ellenőrzésére, hogy egy szám egy másiknak a négyzetgyöke-e, de nem alkalmas a négyzetgyök előállítására.
- A matematikai függvénnyel egy bizonyos tulajdonságot *deklarálunk,* a funkcionális nyelvi függvénnyel (eljárással) azt is megmondjuk, *hogyan kell kiszámítani* az adott értéket. (A deklaratív programozás tehát csak az imperatív programozáshoz *képest* tekinthető deklaratívnak.)
- A négyzetgyökszámítás legismertebb módszere a *szukcesszív approximáció:* ha az y az x négyzetgyökének egy közelítése, akkor az y és az x/y átlaga a négyzetgyök egy jobb közelítése lesz. A lépéssorozat akkor fejeződik be, amikor a közelítő értéket már elég jónak tartjuk.
- Írjuk le ezt az algoritmust SML-nyelven (l. következő dia)!

Négyzetgyökvonás Newton-módszerrel (folyt.)

```
val rec sqrtIter =
      fn (guess, x) => if goodEnough(guess, x)
                       then guess
                       else sqrtIter(improved(guess, x), x)
```

- Itt a `rec` kulcsszó arra utal, hogy az értékdeklaráció *rekurzív*: a most deklarált nevet *használjuk* a deklaráció jobb oldalán, a függvény definíciójában.
- A megoldási stratégiánkat jól tükrözi a fenti programrészlet. Ezt a stílust *fölülről lefelé haladó* (top down) módszernek nevezik. Kezdetben nem foglalkozunk a részletekkel, feltesszük, hogy minden megvan, amire szükségünk van, legfeljebb később megírjuk.

Most tehát definiálnunk kell még néhány részletet.

```
val improved = fn (guess, x) => average(guess, x/guess)
val average = fn (x, y) => (x+y)/2.0
val goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
val square_r = fn (x : real) => x * x
```

Végül meg kell hívnunk az `sqrtIter` függvényt a négyzetgyök első közelítő értékével.

```
val sqrt = fn x => sqrtIter(1.0, x);
```

Négyzetgyökvonás Newton-módszerrel (folyt.)

- Sajnos, gond van a deklarációk sorrendjével, az SML ugyanis azt igényli, hogy egy deklaráció jobb oldalán minden kifejezésnek legyen értéke.
- Ha a deklarációk sorrendjét megfordítjuk, a programszöveg kevésbé hűen tükrözi a követett módszert.
- Megoldást az ún. *egyidejű deklaráció* jelent, amely előbb beolvassa, majd egyidejűleg dolgozza fel az összes deklarációt. Az egyidejűleg deklarálni kívánt értékeket az `and` kulcsszóval kell elválasztani egymástól.

```
val rec sqrtIter =
      fn (guess, x) => if goodEnough(guess, x)
                       then guess
                       else sqrtIter(improved(guess, x), x)
and improved = fn (guess, x) => average(guess, x/guess)
and average = fn (x, y) => (x+y)/2.0
and goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
and square_r = fn (x : real) => x * x
val sqrt = fn x => sqrtIter(1.0, x)
```

Négyzetgyökvonás Newton-módszerrel (folyt.)

- Eddigi absztrakciós eszközeink (a névadás, a függvény, ill. eljárás jók a dolgok *megnevezésére,* de alkalmatlanok bizonyos *részletek elrejtésére.*
- Elrejtésre az SML-ben több eszközünk is van, ilyen pl. a függvény, az eljárás is, és ilyen egyebek mellett a „kifejezés lokális érvényű deklarációval”, másnéven `let`-kifejezés speciális nyelvi elem is.
- `let`-kifejezést használunk akkor is, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani.
- Szintaxisa:

```
let d
in e
end
```

ahol *d* egy nemüres deklarációsorozat,

e egy nemüres kifejezés.

A továbbiakban a függvénydefiníciókat a `fun` „szintaktikai édesítőszerrel” tálaljuk.

Négyzetgyökvonás Newton-módszerrel (folyt.)

```
fun sqrt x =
  let fun sqrtIter (guess, x) = if goodEnough(guess, x)
                                then guess
                                else sqrtIter(improved(guess, x),x)
      and improved (guess, x) =  average(guess, x/guess)
      and average (x, y) = (x+y)/2.0
      and goodEnough (guess, x) = abs(square_r guess - x) < 0.001
      and square_r (x : real) = x * x
  in
       sqrtIter(1.0, x)
  end
```

- Az SML-ben a nevek *szövegkörnyezettől függő* érvényességi és láthatósági szabályai hasonlóak a más programozási nyelvekben megszokottakhoz.
- `sqrt x` formális paramétere például látható a függvény törzsében definiált függvényekben is, hacsak egy azonos nevű formális paraméter el nem fedi. Az `sqrt`-ben *lokális* `x` *globális érvényű névként* használható a lokális függvénydefiníciókban.
- A `:real` *típusmegkötés* elhagyható: az SML a szövegkörnyezetből kitalálja a paraméter típusát.

Négyzetgyökvonás Newton-módszerrel (folyt.)

A könnyített változat:

```
fun sqrt x =
  let fun sqrtIter guess = if goodEnough guess
                           then guess
                           else sqrtIter(improved guess)
      and improved guess = average(guess, x/guess)
      and average (x, y) = (x+y)/2.0
      and goodEnough guess = abs(square_r guess - x) < 0.001
      and square_r x = x * x
  in
      sqrtIter 1.0
  end;
```

Azzal, hogy értelmes nevet adtunk az egyes programelemeknek, könnyebbé, egyszerűbbé tettük

- „az ügyek szétválasztásával” a program kidolgozását,
- a jövőbeli olvasóknak a megértését,
- a javítását (ha a segédfüggvényeknek nincsen *mellékhatásuk*, a specifikáció megtartása mellett bármikor lecserélhetők).

Eljárások (függvények) és folyamatok

- Az eljárások (függvények) olyan *minták*, amelyek megszabják a számítási folyamatok menetét, *lokális* viselkedését.
- Egy számítási folyamat *globális* viselkedését (pl. a szükséges lépések számát, a végrehajtási időt) általában nehéz megbecsülni, de törekednünk kell rá.

Lineáris rekurzió és iteráció

- A faktoriális matematikai definíciójának hű tükörképe az alábbi SML-program:

$0! = 1$
$n! = n(n-1)!$

```
(* PRE : x > 0 *)
fun factorial 1 = 1
  | factorial n = n * factorial(n-1)
```

- Ha a helyettesítési modellünket alkalmazzuk, láthatjuk, hogy a program által létrehozott folyamat az összes tényezőt `n`-től 1-ig eltárolja, mielőtt az első szorzást végrehajtaná („késlelteti” a szorzásokat) – a folyamat *lineáris-rekurzív folyamat.*
- Ha ehelyett 1-et szoroznánk 2-vel, majd a részszorzatokat 3-mal, 4-gyel s.í.t., akkor az `n` érték meghaladásakor az utolsó részszorzat éppen `n` faktoriálisa lenne! Ehhez a programban szükségünk van egy olyan *formális paraméterre* (tkp. lokális változóra), amely tárolja a részszorzat aktuális értékét, és egy másikra, amely 1-től `n`-ig számlál. A létrehozott folyamat *lineáris-iteratív folyamat.*

Lineáris rekurzió és iteráció (folyt.)

```
fun factorial n =
      let fun factIter (product, counter) =
                  if counter > n
                      then product
                  else factIter(product*counter, counter+1)
      in
          factIter(1, 1)
      end
```

Eggyel kevesebb paramétere lehet `factIter`-nek, ha a számlálót *lefelé* számláltatjuk.

```
(* PRE : x > 0 *)
fun factorial n =
      let fun factIter (product, 1) =
                      product
            | factIter (product, counter) =
                      factIter(product*counter, counter-1)
      in
          factIter(1, n)
      end
```

Eljárások (függvények) és folyamatok

- Ne tévesszük össze egymással a rekurzív (számítási) folyamatot és a rekurzív függvényt (eljárást)!
- Egy rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény (eljárás) önmagára.
- A folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk.
- Ha egy függvény *jobbrekurzív (tail-recursive),* a megfelelő folyamat – az értelmező/fordító jóságától függően – még lehet iteratív.

Még visszatérünk az „absztrakció függvényekkel” témakörhöz, de most témát váltunk: egy nagyon funkcionális adatszerkezettel, a listákkal foglalkozunk.