FUNKCIONÁLIS PROGRAMOZÁS

Az első rész tartalma, ajánlott irodalom

- Absztrakció függvényekkel (eljárásokkal)
  - Programelemek
  - Függvények (eljárások) és az általuk létrehozott folyamatok
  - Magasabbrendű függvények (eljárások)
- Absztrakció adatokkal
  - Az adatabsztrakció fogalma
  - Hierarchikus adatok
  - Polimorfizmus, generikus műveletek

Irodalom

- G. J. Michaelson: *Elementary Standard ML*. UCL Press, ISBN 1-85728-398-8, 1995, http://www.macs.hw.ac.uk/~greg/books.html
- Gert Smolka: *Programmierung. Eine Einführung in die Informatik.* Universität des Saarlandes, 2005, http://www.ps.uni-sb.de/~smolka/programmierung.html
- Abelson, Sussman & Sussman: *Structure and Interpretation of Computer Programs (SICP).* The MIT Press, 1996, http://mitpress.mit.edu/sicp

Történeti áttekintés

A λ-kalkulus rövid története (lásd: Wikipédia)

- 1910: Russel & Whitehead: Principia Mathematica
- 1935: Gödel: nem létezhet erős axióma- és tételrendszer, amely teljes
- 1930-40-es évek: *Alonzo Church: λ-kalkulus*

A Russel paradoxon

- R: az önmagukat nem tartalmazó halmazok halmaza ($R = \{A \mid A \notin A\}$)
- R tartalmazza-e önmagát? $\Rightarrow$ ellentmondás

A Russel paradoxon a λ-kalkulusban

- $R = \lambda x.\neg(x\ x)$
- $R\ R = \neg(R\ R) \Rightarrow$ ellentmondás

Típusos λ-kalkulus

- minden kifejezésnek van típusa
- f csak akkor alkalmazható e-re, ha a típusaik „passzolnak”
- R nem írható le

Történeti áttekintés (folyt.)

Funkcionális programozási nyelvek

- LISP (LISt Processing), 1950-es évek vége, MIT, US, John McCarthy; típusok nélkül
  - bizonyos logikai kifejezések (ún. rekurzív egyenletek) igazolására
  - szimbolikus kifejezések kezelésére
- ML (Meta Language), Edinborough, GB, 1970-es évek közepe; típusok
- Scheme, 1975, MIT, US; típusok nélkül, LISP-dialektus
- *SML* (Standard Meta Language), 1980-as évek vége; típusok
- *Erlang*, 1980-as évek; típusok nélkül, párhuzamos és elosztott – Ericsson, telefonközpontok
- *Haskell*, 1990-es évek, US; típusok, lusta kiértékelés
- Common LISP, 1994, ANSI standard; típus nélküli
- Clean, 1990-es évek közepe, Nijmegen, NL; típusok, lusta kiértékelés
- *Alice*, 2003, Saarbrücken, DE; típusok, jövők, párhuzamos, elosztott, korlátalapú
- *Hume*, 2003, Glasgow, UK; típusok, korlátos erőforrások

Funkcionális programozás

Ami közös a funkcionális nyelvekben

- Rekurzív eljárások (függvények)
- Rekurzív adatstruktúrák
- Eljárások (függvények) kezelése adatként

A következő hetekben

- *számítási folyamatokkal* és az általuk kezelt *adatokkal* foglalkozunk,
- programjainkat – a folyamatokat vezérlő szabályrendszert – az SML funkcionális nyelven írjuk,
- értelmező- és fordítóprogramok:
  - mosml
  - PolyML
  - Alice
  - smlnj

Az absztrakcióról, modellezésről, programstruktúráról tanulandók más programozási nyelvek használatakor is hasznosak lesznek.

Példák az MOSML használatára

Az SML értelmező is ún. *read-eval-print* ciklusban dolgozik. A kiértékelés (*eval*) a `;`, majd az *enter* leütésére kezdődik el.

```
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
- 486;
> val it = 486 : int
- 2.3-0.3;
> val it = 2.0 : real
- "te"^"xt";
> val it = "text" : string
- op+(482,4);
> val it = 486 : int
- #"A"< #"a";
> val it = true : bool
- val it = 486;
> val it = 486 : int
```

Minden kifejezés kiértékelése valójában *értékdeklaráció:* ha nem adunk meg más nevet, az SML az `it` nevet köti az adott kifejezés értékéhez.

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Névadás, (globális) környezet

Egy *értékdeklarációval* egy nevet kötünk egy értékhez:

```
- val size = 2;
> val size = 2 : int
- 5*size;
> val it = 10 : int
- val ||| = 3;
> val ||| = 3 : int
- ||| * size;
> val it = 6 : int
```

Megjegyzés: a `|` és a `*` ún. tapadó írásjelek, ezért közéjük szóközt kell rakni (mint a példában).
A *névadás* a legegyszerűbb absztrakciós eszköz (a programozási nyelvekben is).
A *név–érték* párt az SML a „memóriájában”, az ún. globális *környezetben* tárolja. Később látni fogjuk, hogy vannak ún. lokális környezetek is.
Minden értéknek, így minden névnek is van típusa. A függvény, az operátor is érték, ezért a függvénynek (a függvény nevének), az operátornak (az operátor jelének) is van típusa, pl.

```
- round;
> val it = fn : real -> int
- op+;
> val it = fn : int * int -> int
```

Nevek képzési szabályai

- Alfanumerikus név: kis- és nagybetűk, számjegyek, percjelek (’) és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik
  - Példák: `tothGyorgy  Toth_3_Gyorgy  toth’gyorgy  ’gyurika`
  - Percjellel kezdődő név csak ún. típusváltozót jelölhet.
- Írásjelekből álló név: az alábbi 20 *tapadó* írásjel tetszőleges, nem üres sorozata

  `! % & $ # + - / : < = > ? @ \ ~ ’ ^ | *`
  - Példák: `++ <-> ||| ## |=|`
- Speciális a szerepe az alábbi fenntartott jeleknek

  `( ) [ ] { } , ; . ...`
- Más jelentés nem rendelhető az alábbi fenntartott nevekhez és jelekhez

```
abstype and andalso as case do datatype else end eqtype exception
fn fun functor handle if in include infix infixr let local nonfix
of op open orelse raise rec sharing sig signature struct structure
then type val where with withtype while : :: :> _ | = => -> #
```

Egyszerű adattípusok

Típusnév	*Megnevezés*	*Könyvtár*
`int`	előjeles egész	`Int`
`real`	racionális (valós)	`Real`
`char`	karakter	`Char`
`bool`	logikai	`Bool`
`string`	füzér	`String`
`word`	előjel nélküli egész	`Word`
`word8`	8 bites előjel nélküli egész	`Word8`

A beépített operátorok és precedenciájuk

Az alábbi táblázatban *wordint*, *num* és *numtxt* az alábbi típusnevek helyett állnak.

wordint = int, word, word8
num = int, real, word, word8
numtxt = int, real, word, word8, char, string

Prec.	*Operátor*	*Típus*	*Eredmény*	*Kivétel*
7	*	*num * num -> num*	szorzat	Overflow
	/	real * real -> real	hányados	Div, Overflow
	div, mod	*wordint * wordint -> wordint*	hányados, maradék	Div, Overflow
	quot, rem	int * int -> int	hányados, maradék	Div, Overflow
6	+, -	*num * num -> num*	összeg, különbség	Overflow
	^	string * string -> string	egybeírt szöveg	Size
5	::	'a * 'a list -> 'a list	elemmel bővített lista (jobbra köt)	
	@	'a list * 'a list -> 'a list	összefűzött lista (jobbra köt)	
4	=, <>	''a * ''a -> bool	egyenlő, nem egyenlő	
	<, <=	*numtxt * numtxt* -> bool	kisebb, kisebb-egyenlő	
	>, >=	*numtxt * numtxt* -> bool	nagyobb, nagyobb-egyenlő	
3	:=	'a ref * 'a -> unit	értékadás	
	o	('b -> 'c) * ('a -> 'b)-> ('a -> 'c)	két függvény kompozíciója	
0	before	'a * 'b -> 'a	a bal oldali argumentum	

div $-\infty$, quot 0 felé kerekít. div és quot, ill. mod és rem eredménye csak akkor azonos, ha két operandusuk azonos előjelű (mindkettő pozitív, vagy mindkettő negatív).

Különleges állandók

- Előjeles egész állandó

  Példák: `0 ~0 4 ~04 999999 0xFFFF ~0x1ff`

  Ellenpéldák: `0.0 ~0.0 4.0 1E0 -317 0XFFFF -0x1ff`

- Racionális (valós) állandó

  Példák: `0.7 ~0.7 3.32E5 3E~7 ~3E~7 3e~7 ~3e~7`

  Ellenpéldák: `23 .3 4.E5 1E2.0 1E+7 1E-7`

- Előjel nélküli egész állandó

  Példák: `0w0 0w4 0w999999 0wxFFFF 0wx1ff`

  Ellenpéldák: `0w0.0 ~0w4 -0w4 0w1E0 0wXFFFF 0WxFFFF`

- Karakterállandó: a # jelet közvetlenül követő, egykarakteres füzérállandó (l. a következő lapon).

  Példák: `#"a" #"\n" #"\^Z" #"\255" #"\""`

  Ellenpéldák: `# "a" #c #""" #'a'`

- Logikai állandó: csupán kétféle lehet.

  Példák: `true false`

  Ellenpéldák: `TRUE False 0 1`

Különleges állandók, escape-szekvenciák

- Füzérállandó: idézőjelek (") között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot).
- Escape-szekvenciák

`\a`	Csengőjel (BEL, ASCII 7).
`\b`	Visszalépés (BS, ASCII 8).
`\t`	Vízszintes tabulátor (HT, ASCII 9).
`\n`	Újsor, soremelés (LF, ASCII 10).
`\v`	Függőleges tabulátor (VT, ASCII 11).
`\f`	Lapdobás (FF, ASCII 12).
`\r`	Kocsi-vissza (CR, ASCII 13).
`\^`*c*	Vezérlő karakter, ahol $64 \leq c \leq 95$ (@ . . . _), és `\^`*c* ASCII-kódja 64-gyel kevesebb *c* ASCII-kódjánál.
`\`*ddd*	A *ddd* kódú karakter (*d* decimális számjegy).
`\u`*xxxx*	Az *xxxx* kódú karakter (*x* hexadecimális számjegy).
`\"`	Idézőjel (").
`\\`	Hátratört-vonal (\).
`\`$f \cdots f$`\`	Figyelmen kívül hagyott sorozat. $f \cdots f$ nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

Összetett kifejezés kiértékelése

Egy összetett kifejezést az SML két lépésben értékel ki (ez az ún. mohó kiértékelés):

1. először kiértékeli az operátort (műveleti jelet, függvényjelet) és az argumentumait (aktuális paramétereit),
2. majd alkalmazza az operátort az argumentumokra.

Vegyük észre, hogy ez a kiértekelési szabály azért ilyen egyszerű, mert rekurzív.

Az egyszerű kifejezések kiértékelési szabályai:

1. az állandók (jelölések) értéke az, amit jelölnek,
2. a belső (beépített) műveletek a megfelelő gépi utasításokat aktivizálják,
3. a nevek értéke az, amihez a környezet köti az adott nevet.

Megjegyzés: a 2. pont a 3. pont speciális esetének is tekinthető.

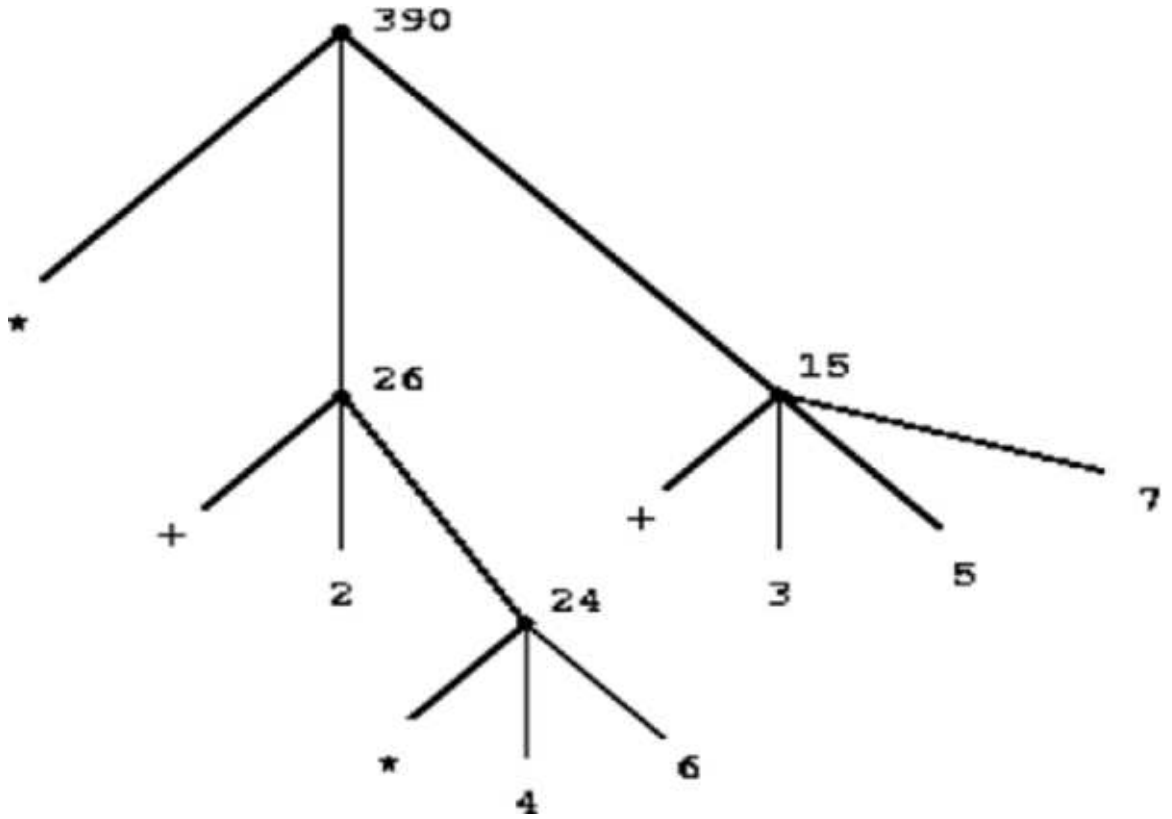
Példa:

```
(2+4*6)*(3+5+7) = op*(op+(2,op*(4,6)),op+(op+(3,5),7))
```

A kifejezéseket ún. kifejezésfával ábrázolhatjuk (ld. a következő dián).

Összetett kifejezés kifejezésfája

```
(2+4*6)*(3+5+7) = op*(op+(2,op*(4,6)),op+(op+(3,5),7))
```



A levelek operátorok vagy primitív kifejezések (állandók, nevek), a csomópontok részkifejezések eredményei. A kiértékelés során az operandusok alulról fölfelé „terjednek”.

Névtelen függvény, függvény definiálása

Névtelen függvény λ-jelöléssel: pl. `(fn x => x*x)`

Névtelen függvény alkalmazása: pl. `(fn x => x*x) 2`

- Az `fn`-t *lambdának* olvassuk.
- Az `x` a függvény formális paramétere (lokális [érvényű] név!).
- Az `x*x` a függvény törzse.
- A `2` a függvény aktuális paramétere.

Névadás függvényértéknek (azaz függvénynév deklarálása):

```
val square = fn x => x * x

val sumOfSquares = fn (x, y) => square x + square y

val f = fn a => sumOfSquares(a+1, a*2)
```

A felhasználó által definiált függvények ugyanúgy használhatók, mint a belső (beépített) függvények.

További példák SML-függvények definiálására

A feladat: egyszeres Hamming-távolságú ciklikus kód előállítása

- A függvényt pl. *táblázattal* adhatjuk meg:

00	01
01	11
11	10
10	00

```
fn 00 => 01
 | 01 => 11
 | 11 => 10
 | 10 => 00
```

- Klózok: minden lehetséges esetre egy-egy klóz.
- A függvény néhány alkalmazása:
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 10`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 11`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 111`
- Mintaillesztés: (egyirányú) egyesítés
- Érthető, de nem robosztus (ui. ez a függvény *parciális* függvény).

További példák SML-függvények definiálására (folyt.)

A feladat: modulo n alapú inkrementálás (pl. $n = 5$-re)

- A függvényt általában *algoritmussal* adjuk meg (nem táblázattal), egyébként
  - az argumentum nem lehetne változó,
  - túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod 5`
  - az `i` ún. *kötött változó*, a névtelen függvény argumentuma
- A függvény néhány alkalmazása:
  - `(fn i => (i + 1) mod 5) 2`
  - `(fn i => (i + 1) mod 5) 4`
  - `(fn i => (i + 1) mod 5) 3.0` – Hiba!
- Ez a függvény definiálható két klózzal is (a sorrend fontos!):
  `fn 4 => 0 | i => i + 1`
- Az SML (a Prologgal ellentétben) mindig csak az *első* illeszthető klózt használja!
- A függvény egyik változata sem robosztus. Melyik a szerencsésebb?

Függvényérték névhez kötése (függvényérték deklarálása)

- Láttuk, hogy nevet függvényértékhez ugyanúgy köthetünk, mint bármely más értékhez.
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
  - `val incMod = fn 4 => 0 | i => i + 1`
- Szintaktikus édesítőszerrel (`fun`)
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
  - ```
    fun incMod 4 = 0
      | incMod i = i + 1
    ```
- Alkalmazásuk argumentumra
  - `kovKod 01`
  - `incMod 4`

Fejkomment

Írjunk *deklaratív fejkommentet* minden (függvény)érték-deklarációhoz!

- ```
  (* kovKod cc = az egyszeres Hamming-távolságú, kétbites,
                  ciklikus kódkészlet cc-t követő eleme
     PRE: cc ∈ {00, 01, 11, 10}
  *)
  fun kovKod 00 = 01
    | kovKod 01 = 11
    | kovKod 11 = 10
    | kovKod 10 = 00
  ```
- `PRE` = *pre*condition, előfeltétel
- `PRE: cc ∈ {00, 01, 11, 10}` jelentése: a `kovKod` függvény `cc` argumentumának a `{00, 01, 11, 10}` halmazbeli értéknek kell lennie, ellenkező esetben a függvény eredménye nincs definiálva.
- ```
  (* incMod i = (i+1) modulo 5 szerint
     PRE: 5 > i >= 0
  *)
  fun incMod i = (i+1) mod 5
  ```

A függvény mint érték

- A függvény „teljes jogú” *(first-class)* érték a funkcionális programozási nyelvekben
  - A függvény típusa általában: $\alpha \to \beta$, ahol az α az argumentum, a β az eredmény típusát jelöli
  - A függvény maga is: érték. *Függvényérték.*
  - Fontos: a függvényérték *nem* a függvény egy *alkalmazásának* az eredménye!
  - Példák függvényértékre
    - $\sin$ (a típusa: *valós* $\to$ *valós*)
    - round (a típusa: *valós* $\to$ *egész*)
    - $\circ$ (függvénykompozíció; a típusa: $((\beta \to \gamma) * (\alpha \to \beta)) \to (\alpha \to \gamma)$)
  - Példák függvényalkalmazásra
    - $\mathrm{round}\ 5.4 = 5$, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
    - $\mathrm{round} \circ \sin$ (a típusa: *valós* $\to$ *egész*)
    - $(\mathrm{round} \circ \sin)1.0 = 1$ (a típusa: *egész*)

Két- vagy többargumentumú függvények

- Függvény alkalmazása két vagy több argumentumra
  1. Az argumentumokat *összetett adatnak* – párnak, rekordnak, listának stb. – tekintjük
     - pl. $f(1,2)$ az f függvény alkalmazását jelenti az $(1,2)$ *párra*,
     - pl. $f[1,2,3]$ az f függvény alkalmazását jelenti az $[1,2,3]$ *listára*.
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f\ 1\ 2 \equiv (f\ 1)\ 2$ azt jelenti, hogy
     - az első lépésben az f függvény alkalmazzuk az 1 értékre, ami egy *függvényt ad eredményül*,
     - a második lépésben az első lépésben kapott függvényt alkalmazzuk a 2 értékre, így kapjuk meg az $f\ 1\ 2$ függvényalkalmazás (vég)eredményét.
- Az $f\ 1\ 2$ esetben az f függvényt *részlegesen alkalmazható* függvénynek nevezzük.
- A programozó szabadon dönthet, hogy a függvényt részlegesen alkalmazható vagy pl. egy párra alkalmazható formában írja meg. A különbség *csak* a szintaxisban van. A részlegesen alkalmazható változat, mint látni fogjuk, rugalmasabban használható.
- Infix jelölés: $x \oplus y \equiv$ a $\oplus$ függvény alkalmazása az (x,y) párra mint argumentumra. Az infix operátor balra vagy jobbra köt.

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* – egyenlők helyettesítése egyenlőkkel, *equals replaced by equals* – 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 (mohó kiértékelés), normál sorrend (lusta kiértékelés

- 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* (applicative order) vagy *mohó* (eager) kiértékelésnek nevezzük.
- Van más lehetőség is: a kiértékelést addig halogatjuk, ameddig csak lehetséges. Ezt *normál sorrendű* (normal order), *szükség szerinti* (by need) vagy *lusta* (lazy) 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.
- Vegyük észre, hogy szükség szerinti kiértékelés mellett a példában egyes részkifejezéseket akár kétszer is ki kell értékelni.
- Ezen jobb értelmezők/fordítók (pl. Alice, Haskell) 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 szükségessége. Ma általában ezt nevezik *lusta* kiértékelésnek.

Feltételes kifejezések, logikai műveletek, predikátumok

- 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.
- 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.)

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 nevezzük az olyan függvényt, amelynek `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.)

- 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 ugyanis, 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. Vö. MIT és HOGYAN.)
- 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. A legalapvetőbb maga a függvény, és ilyen a „kifejezés lokális érvényű deklarációval” (rövidebben „kifejezés lokális deklarációval”) speciális nyelvi elem is, amit röviden `let`-kifejezésnek nevezünk.
- `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* (kontextustól) *függő* érvényességi és láthatósági szabályai hasonlóak a más programozási nyelvekben megszokottakhoz.
- Az `sqrt` függvény `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 kontextusból 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” *(separation of concerns)* 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 (processzek) 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 : n >= 0 *)
fun factorial 0 = 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
```

- `factIter` világosabb szerkezetű változatát kapjuk, ha a számlálót *lefelé* számláltatjuk.

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

Eljárások (függvények) és folyamatok (folyt.)

- 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: megismerkedünk a *paraméteres polimorfizmus* fogalmával, majd egy nagyon funkcionális adatszerkezettel, a listával foglalkozunk.

POLIMORFIZMUS

Polimorfizmus

- Nézzük az identitásfüggvényt: `fun id x = x`.
- Mi az `x` típusa? Tetszőleges, típusát *típusváltozó* jelöli: `val 'a id = fn : 'a -> 'a`.
- `id` *polimorf* függvényt jelöl, `x` és `id` *politípusú* nevek.
- A *percjellel* kezdődő típusnév (pl. `'a`, olvasd *alfa*): *típusváltozó*.
- Nézzük az egyenlőségfüggvényt: `fun eq (x, y) = x = y`.
- Mi az `x` és `y` típusa? Tetszőleges, típusát szintén *típusváltozó* jelöli:
  `val ''a eq = fn : ''a * ''a -> bool`.
- A *két* percjellel kezdődő típusnév (pl. `''a`, olvasd *alfa*) az ún. *egyenlőségi típus változója*.

Polimorfizmus (folyt.)

Polimorfizmus többféle változatban fordul elő a programozásban.

- Egy *polimorf név* **egyetlen** olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.
- Egy *többszörösen terhelt név* **több különböző** algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyifélét; ez az *ad-hoc* vagy *többszörös terheléses* polimorfizmus.
- A polimorfizmus harmadik változata az *öröklődéses polimorfizmus* (vö. pl. objektum-orientált programozás). Az ún. *genericitás* is tekinthető az öröklődéses polimorfizmus egy változatának.

LISTÁK

Lista: definíciók, adat- és típuskonstruktorok

- Definíciók
  1. A *lista* azonos típusú elemek véges (de nem korlátos!) sorozata.
  2. A lista olyan *rekurzív* lineáris adatszerkezet, amely azonos típusú elemekből áll, és
     - vagy üres,
     - vagy egy elemből és az elemet követő listából áll.
- Konstruktorok
  - Az üres lista jele a `nil` *adatkonstruktorállandó*, röviden *állandó*.
  - A `nil` helyett általában a `[]` jelet használjuk (szintaktikus édesítőszer).
  - A `nil` típusa: `'a list`.
  - Az `'a` *típusváltozót* jelöl, a `list`-et *típuskonstruktornak* nevezzük.
  - A `::` *adatkonstruktorfüggvény* új listát hoz létre egy elemből és egy (esetleg üres) listából.
  - A `::` típusa `'a * 'a list -> 'a list`, infix pozíciójú, 5-ös precedenciájú, jobbra köt. Infix pozíciója miatt *adatkonstruktoroperátornak* is nevezzük.
  - A `::`-ot *négyespontnak* vagy *cons*-nak olvassuk (vö. *constructor,* ami ennek az adatkonstruktorfüggvénynek a hagyományos neve a λ-kalkulusban és egyes funkcionális nyelvekben).

Lista: jelölések, minták

- Példák
  - Lista létrehozása adatkonstruktorokkal

```
[]            nil            #"" :: nil
3 :: 5 :: 9 :: nil     = 3 :: (5 :: (9 :: nil))
```

  - Szintaktikus édesítőszer lista jelölésére

```
[3, 5, 9]                  =  3 :: 5 :: 9 :: nil
```

  - Vigyázat! A Prolog listajelölése hasonló, de vannak lényeges különbségek:

SML	Prolog		SML	Prolog	
`[]`	`[]`	azonos	`(x::xs)`	`[X|Xs]`	különböző
`[1,2,3]`	`[1,2,3]`	azonos	`(x::y::ys)`	`[X,Y|Ys]`	különböző

- Minták

  A `[]`, `nil` adatkonstruktorállandóval és a `::` adatkonstruktoroperátorral felépített kifejezések, valamint a `[x1, x2, ..., xn]` listajelölés mintában is alkalmazhatók.

Lista: fej (hd), farok (tl)

- A nemüres lista első eleme a lista *feje*.

```
(* hd : 'a list -> 'a
   hd xs = a nemüres xs első eleme (az xs feje)
*)
fun hd (x :: _) = x;
```

- A nemüres lista első utáni elemeiből áll a lista *farka*.

```
(* tl : 'a list -> 'a list
   tl xs = a nemüres xs első utáni elemeinek az eredetivel
           azonos sorrendű listája (az xs farka)
*)
fun tl (_ :: xs) = xs;
```

- `hd` és `tl` *parciális* függvények. Ha könyvtárbeli megfelelőiket (`List.hd`, `List.tl`) üres listára alkalmazzuk, `Empty` néven *kivételt* jeleznek.
- Az _ (aláhúzás) az ún. *mindenesjel*, azaz a mindenre illeszkedő minta. Figyelem: a mindenesjel kifejezésben – pl. egyenlőségjel jobb oldalán – nem használható!

Lista: hossz (`length`), elemek összege (`isum`), szorzata (`rprod`)

- Egy lista hosszát adja eredményül a `length` függvény (vö. `List.length`).

```
(* length : 'a list -> int
   length zs = a zs lista elemeinek száma *)
fun length []        = 0
  | length (_ :: zs) = 1 + length zs
```

- Egy egész számokból álló lista elemeinek összegét adja eredményül `isum`.

```
(* isum : int list -> int
   isum ns = az ns egészlista elemeinek összege *)
fun isum []        = 0
  | isum (n :: ns) = n + isum ns
```

- Egy valós számokból álló lista elemeinek szorzatát adja eredményül `rprod`.

```
(* rprod : real list -> real
   rprod xs = az xs valós lista elemeinek szorzata *)
fun rprod []        = 1.0
  | rprod (x :: xs) = x * rprod xs;
```

Példák: hd, tl, length, isum, rprod

- hd, tl

A kifejezés	Az mosml válasza
`List.hd [1, 2, 3];`	`> val it = 1 : int`
`List.hd [];`	`! Uncaught exception:`
	`! Empty`
`List.tl [1, 2, 3];`	`> val it = [2, 3] : int list`
`List.tl [];`	`! Uncaught exception:`
	`! Empty`

- length, isum, rprod

A kifejezés	Az mosml válasza
`length [1, 2, 3, 4];`	`> val it = 4 : int`
`length [];`	`> val it = 0 : int`
`isum [1, 2, 3, 4];`	`> val it = 10 : int`
`isum [];`	`> val it = 0 : int`
`rprod [1.0, 2.0, 3.0, 4.0];`	`> val it = 24.0 : real`
`rprod [];`	`> val it = 1.0 : real`

`map`: adott függvény alkalmazása egy lista minden elemére

- Példa: vonjunk négyzetgyököt egy valós számokból álló lista minden eleméből!

```
load "Math";
map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0];
```

- Általában: `map f [`x_1`, `x_2`, ..., `x_n`] = [f `x_1`, f `x_2`, ..., f `x_n`]`
- `map` definíciója (`map` polimorf függvény!):

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   map f xs = az xs f-fel átalakított elemeiből álló lista
*)
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs;
```

- `map` típusa (mivel a `->` típusoperátor jobbra köt!):

  `('a -> 'b) -> 'a list -> 'b list` $\equiv$ `('a -> 'b) -> ('a list -> 'b list)`

- A `map` egy *részlegesen alkalmazható*, magasabbrendű függvény: ha egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor egy `'a list -> 'b list` típusú **függvényt** ad eredményül. A kapott függvényt egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.
- `map` – teljes nevén `List.map` – belső függvény az SML-ben.

A program helyességének (informális) igazolása a `map` példáján

- A rekurzív programról be kell látnunk, hogy
  - funkcionálisan helyes (azt kapjuk eredményül, amit várunk),
  - a kiértékelése biztosan befejeződik (nem „végtelen” a rekurzió).
- Bizonyítása hossz szerinti *strukturális indukcióval* lehetséges (visszavezethető a teljes indukcióra).

```
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs
```

  - Feltesszük, hogy a `map` jó eredményt ad az eggyel rövidebb listára (azaz a lista farkára).
  - Alkalmazzuk az `f`-et a lista első elemére (a fejére).
  - A fej transzformálásával kapott eredményt a farok transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
  - A kiértekelés véges számú lépésben befejeződik, mert
    - a lista véges,
    - a `map`-et a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk, és
    - a rekurziót leállítjuk (kezeljük az *alapesetet*, ui. van nemrekurzív ág).

Néhány belső, ill. könyvtári függvény

- `explode : string -> char list` – a füzér karaktereiből álló lista
  pl. `explode "abc" = [#"a", #"b", #"c"]`
- `implode : char list -> string` – a karakterlista elemeiből álló füzér
  pl. `implode [#"a", #"b", #"c"] = "abc"`
- `map`-nek más változatai is vannak, amelyek egyéb összetett adatokra alkalmazhatók. Például
  - `String.map : (char -> char) -> string -> string`
  - `Vector.map : ('a -> 'b) -> 'a vector -> 'b vector`
- A `Char` könyvtárban sok hasznos ún. *tesztelő* függvény található, például:
  - `Char.isLower : char -> bool` – igaz az angol ábécé kisbetűire
  - `Char.isSpace : char -> bool` – igaz a hat formázó karakterre
  - `Char.isAlpha : char -> bool` – igaz az angol ábécé betűire
  - `Char.isAlphaNum : char -> bool` – igaz az angol ábécé betűire és a számjegyekre
  - `Char.isAscii : char -> bool` – igaz a 128-nál kisebb ascii-kódú karakterekre

  pl. `Char.isSpace #"\t" = true; Char.isAlphaNum #"!" = false`

`filter`: adott predikátumot kielégítő elemek kiválogatása

- Példa: gyűjtsük ki a kisbetűket egy karakterlistából!

  ```
  List.filter Char.isLower (explode "VaLtOgAtVa") = [#"a",#"t",#"g",#"t",#"a"];
  ```

- Általában, ha `p` x_1 `= true,` `p` x_2 `= false,` `p` x_3 `= true, ..., p` x_{2k+1} `= true,` akkor `filter p [`x_1`,` x_2`,` x_3`, ...,` x_{2k+1}`] = [`x_1`,` x_3`, ...,` x_{2k+1}`]`.

- `filter` definíciója:

  ```
  (*  filter : ('a -> bool) -> 'a list -> 'a list
      filter p zs = a zs p-t kielégítő elemeiből álló lista
  *)
  fun filter _ [] = []
    | filter p (x :: xs) =
        if p x then x :: filter p xs else filter p xs;
  ```

- `filter` típusa, ha egyargumentumú függvénynek tekintjük (a `->` jobbra köt!):

  `filter : ('a -> bool) -> ('a list -> 'a list)`.

  Azaz ha `filter`-t egy `'a -> bool` típusú függvényre (predikátumra) alkalmazzuk, akkor egy `('a list -> 'a list)` típusú függvényt ad eredményül. A kapott függvényt egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.

Lista legnagyobb elemének megkeresése

- Üres listának nincs legnagyobb eleme,
- egyelemű lista egyetlen eleme a „legnagyobb”,
- legalább kételemű lista legnagyobb eleme
  - az első elem és a maradéklista elemeinek legnagyobbika közül a nagyobb:

```
load "Int";
(* maxl : int list -> int
   maxl ns = az ns egészlista
             legnagyobb eleme
*)
fun maxl [] = raise Empty
  | maxl [n] = n
  | maxl (n::ns) = Int.max(n, maxl ns)
```

`max` egy változata egészekre:

```
(* max: int * int -> int
   max (n,m) = n és m közül
               a nagyobb
*)
fun max (n,m) = if n>m
                then n
                else m
```

  - az első két elem legnagyobbika és a maradéklista elemei közül a legnagyobb:

```
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl'(Int.max(n,m)::ns)
```

    - `maxl`-lel szemben itt a klózok sorrendje közömbös (a minták diszjunktak).
    - `maxl'` *jobbrekurzív*, tárigénye konstans.

Lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető *polimorffá* a `maxl` függvény? Úgy, hogy ún. *generikus* függvényként definiáljuk: *aktuális paraméterként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max zs = a zs lista max szerint legnagyobb eleme
*)
fun maxl max [] = raise Empty
  | maxl max [z] = z
  | maxl max (z::zs) = max(z, maxl max zs)
```

- `max` mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javíthatja a hatékonyságot, ha *`let`-kifejezést* használunk.

```
fun maxl max zs = let fun mxl [] = raise Empty
                        | mxl [y] = y
                        | mxl (y::ys) = max(y, mxl ys)
                  in
                      mxl zs
                  end
```

Változatok `max`-ra

Változatok `max`-ra

- ```
  (* charMax : char * char -> char
     charMax (a, b) = a és b közül a nagyobbik
  *)
  fun charMax (a, b) = if ord a > ord b then a else b;
  ```

  vagy egyszerűen (`ord` nélkül)

  ```
  fun charMax (a : char, b) = if a > b then a else b;
  ```

- ```
  (* pairMax : ((int * real) * (int * real)) -> (int * real)
     pairMax (n, m) = n és m közül lexikografikusan a nagyobbik
  *)
  fun pairMax (n as (n1 : int, n2 : real), m as (m1, m2)) =
        if n1 > m1 orelse n1 = m1 andalso n2 >= m2 then n else m;
  ```

- ```
  (* stringMax : string * string -> string
     stringMax (s, t) = s és t közül a nagyobbik
  *)
  fun stringMax (s : string, t) = if s > t then s else t;
  ```
  ```

Listák összefűzése (append) és megfordítása (nrev)

- Két lista összefűzése (append, infix változatban @)

$$[x_1, \ldots, x_m]@[y_1, \ldots, y_n] = [x_1, \ldots, x_{m-1}]@(x_m::[y_1, \ldots, y_n]) = \ldots = [x_1, \ldots, x_m, y_1, \ldots, y_n]$$

Az xs-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az ys-hez, ugyanis a listákat csak elölről tudjuk építeni. A lépések száma $O(n)$.

```
(* append : 'a list * 'a list -> 'a list
   append(xs, ys) = xs összes eleme ys elé fűzve *)
fun append ([], ys) = ys
  | append (x::xs, ys) = x::append(xs, ys)
```

- Lista naív megfordítása (nrev)

$$\texttt{nrev}[x_1, x_2, \ldots, x_m] = \texttt{nrev}[x_2, \ldots, x_m]@[x_1] = \texttt{nrev}[\ldots, x_m]@[x_2]@[x_1] = \ldots = [x_m, \ldots, x_1]$$

A lista elejéről levett elemet egyelemű listaként fűzhetjük a végéhez. A lépések száma $O(n^2)$.

```
(* nrev : 'a list -> 'a list
   nrev xs = xs megfordítva *)
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]
```

Listák megfordítása: példa `nrev` alkalmazására

- Egy példa `nrev` egyszerűsítésére

  A `::` és a `@` jobbra kötnek, precedenciaszintjük 5.

```
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]

fun [] @ ys = ys
  | (x::xs) @ ys = x :: xs @ ys (* = (x :: xs) @ ys *)
```

`nrev([1,2,3,4])` → `nrev([2,3,4])@[1]` → `nrev([3,4])@[2]@[1]`
→ `nrev([4])@[3]@[2]@[1]` → `nrev([])@[4]@[3]@[2]@[1]`
→ `[]@[4]@[3]@[2]@[1]` → `[4]@[3]@[2]@[1]`
→ `4::[]@[3]@[2]@[1]` → `4::[3]@[2]@[1]`)
→ `[4,3]@[2]@[1])` → `4::([3]@[2])@[1])`
→ `[]@[4]@(3::[2,1]` → `[]@[4]@[3,2,1]` → ...

`nrev` rossz hatékonyságú: a lépések száma $O(n^2)$.

Listák összefűzése (`revApp`) és megfordítása (`rev`)

- Egy lista elemeinek egy másik lista elé fűzése fordított sorrendben (`revApp`)

```
(* revApp : 'a list * 'a list -> 'a list
   revApp(xs, ys) = xs elemei fordított sorrendben ys elé fűzve
*)
fun revApp ([], ys) = ys
  | revApp (x::xs, ys) = revApp(xs, x::ys)
```

`revApp` lépésszáma arányos a lista hosszával. Segítségével `rev` hatékonyan:

```
(* rev : 'a list -> 'a list
   rev xs = xs megfordítva
*)
fun rev xs = revApp (xs, [])
```

Egy 1000 elemű listát `rev` 1000 lépésben, `nrev` $\frac{1000 \cdot 1001}{2} = 500500$ lépésben fordít meg. Hatalmas a nyereség!

- `append` – @ néven, infix operátorként – és `rev` beépített függvények, `List.revApp` pedig `List.revAppend` néven könyvtári függvény az SML-ben.

ÖSSZETETT ADATTÍPUSOK

Rekord és ennes

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.

  `{x = 2, y = 1.0} : {x : int, y : real}` és `(2, 1.0) : (int * real)`

- A pár is csak szintaktikus édesítőszer. Pl.

  `(2, 1.0)` ≡ `{1 = 2, 2 = 1.0}` ≡ `{2 = 1.0, 1 = 2}` ≠ `{1 = 1.0, 2 = 2}`.

  Egy párban a tagok sorrendje meghatározó! Az `1` és a `2` is: *mezőnevek*.

- Rekordot kettőnél több értékből is összeállíthatunk. Pl.

  `{nev = "Bea", tel = 3192144, kor = 19} : {kor : int, nev : string, tel : int}`

  Egy hasonló rekord egészszám-mezőnevekkel:

  `{1 = "Bea", 3 = 3192144, 2 = 19}: {1 : string, 2 : int, 3 : int}`

  Az *utóbbi* azonos az alábbi *ennessel* (n-tuple):

  `("Bea", 19, 3192144) : (string * int * int)`

  azaz

  `(string * int * int)` ≡ `{1 = string, 2 = int, 3 = int}`

- Egy rekordban a tagok sorrendje közömbös, a tagokat a mezőnév azonosítja. Egy ennesben a tagok sorrendje nem közömbös, a tagokat a *pozícionális* mezőnév azonosítja.

GYENGE ÉS ERŐS ABSZTRAKCIÓ

Adattípusok: gyenge és erős absztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot) jelöl, az adatszerkezet részeihez csak korlátok között lehet hozzáférni.
- `type`: gyenge absztrakció; pl. `type rat = {num : int, den : int}`
  - Új nevet ad egy típuskifejezésnek (vö. értékdeklaráció).
  - Segíti a programszöveg megértését.
- `abstype`: erős absztrakció
  - Új típust hoz létre: név, műveletek, ábrázolás, jelölés.
  - Túlhaladott, van helyette jobb: `datatype` + modulok
- `datatype`: modulok nélkül gyenge, modulokkal erős absztrakció;
  pl. `datatype 'a esetleg = Semmi | Valami of 'a`
  Belső változata az SML-ben: `datatype 'a option = NONE | SOME of 'a`
  - Új entitást hoz létre.
  - Rekurzív és polimorf is lehet.

Adattípusok: felsorolásos és polimorf típusok `datatype` deklarációval

`datatype logi = Igen | Nem`	Felsorolásos típus.
`datatype logi3 = igen | nem | talan`	Felsorolásos típus.
`datatype 'a esetleg = Semmi | Valami of 'a`	Polimorf típus.

- *Adatkonstruktornak* nevezzük a létrehozott `Igen`, `Nem`, `igen`, `nem`, `talan`, `Semmi` és `Valami` értékeket. `Valami` ún. *adatkonstruktorfügvény*, az összes többi ún. *adatkonstruktorállandó*. Az adatkonstruktorok a többi értéknévvel azonos névtérben vannak.
- *Típuskonstruktornak* nevezzük a létrehozott `logi`, `logi3` és `esetleg` neveket; `esetleg` ún. postfix *típuskonstruktorfüggvény* (vagy típusoperátor), a másik kettő ún. *típuskonstruktorállandó* (röviden típusállandó). Típusnévként használható a típusállandó (pl. `logi`), valamint a típusállandóra vagy típusváltozóra alkalmazott típuskontruktorfüggvény (pl. `int list` vagy `'a esetleg`). A típuskonstruktorok más névtérben vannak, mint az értéknevek.
- Természetesen az adatkonstruktoroknak is van típusuk, pl.

`Igen : logi`	`Semmi : 'a esetleg`
`Nem : logi`	`Valami : 'a -> 'a esetleg`

- Példa `datatype` deklarációval létrehozott adattípust kezelő függvényre

```
fun inverz Nem = Igen | inverz Igen = Nem
```

FONTOS APRÓSÁGOK

Fontos apróságok az SML-ről

- A nullas és a `unit` típus

  A `()` vagy `{}` jelet *nullasnak* nevezzük, típusa: `unit`. A nullas a `unit` típus egyetlen eleme. A `unit` típusműveletek *egységeleme.*

- A `print` függvény

  Ha a `string -> unit` típusú `print` függvényt egy füzérre alkalmazzuk, eredménye a *nullas*, *mellékhatásként* pedig kiírja a a füzér értékét.

- Az `(e1; e2; e3)` szekvenciális kifejezés *eredménye* azonos az `e3` kifejezés eredményével. Ha az `e1` és `e2` kifejezéseknek van mellékhatásuk, az érvényesül. `(e1; e2; e3)` egyenértékű a következő `let`-kifejezéssel:

  ```
  let val _ = e1 val _ = e2 in e3 end
  ```

- Az `e1 before e2 before e3` kifejezés *eredménye* azonos az `e1` kifejezés eredményével. Ha az `e2` és `e3` kifejezésnek van mellékhatása, az érvényesül. `e1 before e2 before e3` egyenértékű a következő `let`-kifejezéssel:

  ```
  let val e = e1 val _ = e2 val _ = e3 in e end
  ```