FUNKCIONÁLIS PROGRAMOZÁS

Történeti áttekintés

A λ-kalkulus rövid története

- 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 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

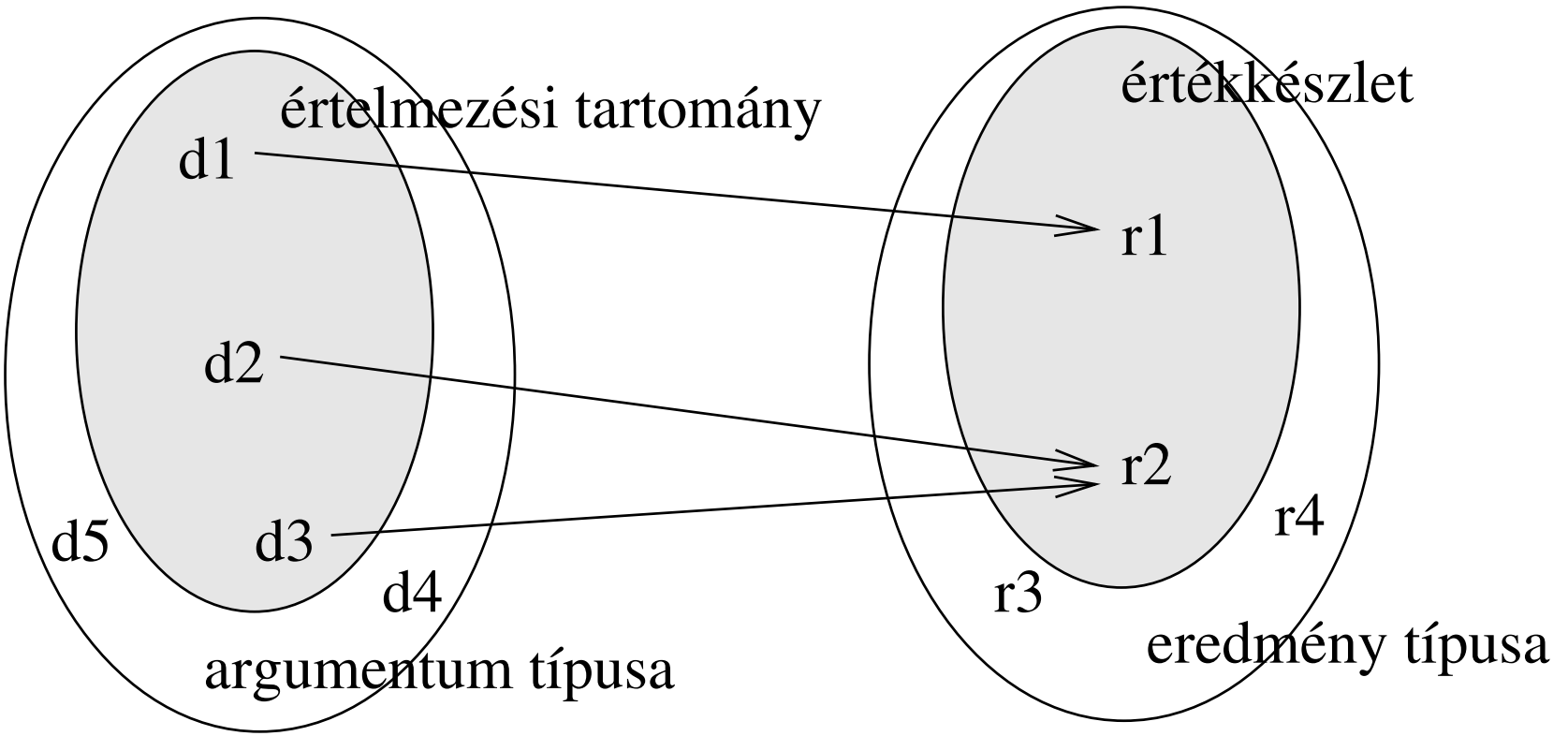
- minden kifejezésnek van egy típusa
- f csak akkor alkalmazható e-re, ha a típusaik „passzolnak”
- R nem írható le

TÍPUS ÉS FÜGGVÉNY

A típus és a függvény fogalma

- A típus fogalma
  - A típus értékek egy halmaza (pl. egész típus ~ az egész számok egy (rész)halmaza)
  - Tetszőleges típus jelölése az ún. *típuselméletben:* α, β ...
  - Típusállandó (azaz konkrét típus) jelölése az SML-ben: `int`, `real`, `char`, `string` ...
  - Típusváltozó jelölése az SML-ben α, β ... helyett: `'a`, `'b` ...
- A függvény fogalma
  - A függvény valamely D halmaznak valamely R halmazba való olyan *egyértelmű* leképzése, amelyet a (d, r) rendezett párok halmaza ad meg, ahol $d \in D$ és $r \in R$.
  - A d a függvény argumentuma (paramétere), az r az eredménye
  - A D a függvény értelmezési tartománya, az R az értékkészlete
  - A típusos nyelvekben d is, r is *meghatározott* típusú
  - A függvény értelmezési tartománya $\subseteq$ az argumentum típusa
  - A függvény értékkészlete $\subseteq$ az eredmény típusa

A függvény mint leképzés



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*)

Függvények osztályozása, ill. alkalmazása

- Függvények osztályozása
  - Parciális függvény: értelmezési tartomány $\subset$ argumentum típusa (figyelem: hibák forrása lehet!)
  - Teljes függvény: értelmezési tartomány $=$ argumentum típusa
  - Szürjektív függvény: értékkészlet $=$ eredmény típusa
  - Nemszürjektív függvény: értékkészlet $\subset$ eredmény típusa
  - Injektív függvény: a leképzés kölcsönösen egyértelmű
  - Az $f : \alpha \to \beta$ injektív függvény inverze: $f^{-1} : \beta \to \alpha$
  - Bijektív = injektív + szürjektív, azaz f bijektív, ha f^{-1} teljes függvény
- Függvények alkalmazása
  - *Függvényalkalmazást* jelöl az f és e jelek egymás mellé írása (*„juxtapozicionálása”*): $f\ e$ azt jelenti, hogy az f-et alkalmazzuk az e-re.
  - Általánosabban: az $f e$ kifejezésben az e tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.
  - Még általánosabban: az $f e$ kifejezésben az f függvényértéket adó tetszőleges kifejezés, e pedig tetszőleges olyan kifejezés, amelynek értéke az f értelmezési tartományába esik.

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

FÜGGVÉNYEK AZ SML-BEN

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. `terulet`, `/\`, `head`

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt pl. *táblázattal* adhatjuk meg:

00	01
01	11
11	10
10	00

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

- Változatok („klózok”): minden lehetséges esetre egy változat.
- Az `fn` (olvasd: *lambda*) névtelen függvényt, *függvénykifejezést* vezet be.
- 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, egyesítés
- Érthető, de nem robosztus (vö. parciális a függvény!).

SML-példa: modulo n alapú inkrementálás

- 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 n`
  - az `i` ún. *kötött változó,* a névtelen függvény argumentuma
  - az `n` ebben a kifejezésben *szabad változó,* és nincs értéke (!)
  - az `n`-et is *le kell kötni* mint a függvény argumentumát
- `fn n => fn i => (i + 1) mod n`
- A függvény néhány alkalmazása:
  - `(fn n => (fn i => (i + 1) mod n)) 128 111`
  - `(fn n => (fn i => (i + 1) mod n)) 4 ~7`
  - `(fn n => (fn i => (i + 1) mod n)) 128 6.0` – Hiba!

Értékdeklaráció SML-ben: függvényérték deklarálása

- Név kötése függvényértékhez
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
  - `val incMod = fn n => fn i => (i + 1) mod n`
- Szintaktikus édesítőszerrel (`fun`)
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
  - `fun incMod n i = (i + 1) mod n`
- Alkalmazásuk argumentumra
  - `kovKod 01`
  - `incMod 128 111`
    ```

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 n i = (i+1) modulo n szerint
     PRE: n > i >= 0
  *)
  fun incMod n i = (i+1) mod n
  ```

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*.
  - 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`

LOKÁLIS ÉRVÉNYŰ DEKLARÁCIÓK

Kifejezés lokális érvényű deklarációval: `let`-kifejezés

- Ún. `let`-kifejezést használunk, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket *el akarunk rejteni* a program többi része elől.
- Szintaxisa:

```
let d
in e
end
```

  ahol `d` egy nemüres deklarációsorozat,
  `e` egy nemüres kifejezés.

- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza
*)
fun length zs =
      let (* len : 'a list * int -> int
             len (zs, n) = az n és a zs lista hosszának összege
          *)
          fun len ([], n)    = n
            | len (_::zs, n) = len(zs, n+1)
      in
          len(zs, 0)
      end
```

EGYSZERŰ ADATTÍPUSOK

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`

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.

NEVEK, OPERÁTOROK

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 típusváltozót jelölhet.
- Írásjelekből álló név: az alábbi 20 jel 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 : :: :> _ | = => -> #
```

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).

LISTÁK

map: adott függvény alkalmazása egy lista minden elemére

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

```
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 ≡ ('a -> 'b) -> ('a list -> 'b list)
```

- A map egy ún. *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.

PROGRAMHELYESSÉG

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` függvényt a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk, és
    - gondoskodtunk a rekurzió leállításáról (a *triviális eset* kezeléséről, ui. van nem rekurzív ág).

NÉHÁNY BELSŐ, ILL. KÖNYVTÁRI FÜGGVÉNY

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`

LISTÁK

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 (-> 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) függvényt ad eredményül. A kapott függvényt egy 'a list típusú listára alkalmazva egy 'a list típusú listát kapunk.

Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az `Int.max` függvényre.
  - Üres listának nincs legnagyobb eleme,
  - egyelemű listában az egyetlen elem a legnagyobb,
  - legalább kételemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közül vesszük a nagyobbat.

```
(* 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
```

- `maxl` egy másik változata (az előzővel szemben itt a klózok sorrendje közömbös!):

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

- `maxl'` *jobbrekurzív,* ezért (talán) hatékonyabb, mint `maxl`. De `maxl` szebb, pontosan tükrözi a definíciót (és a rekurzív gondolkodásmódot), a helyességét is könnyebb belátni.

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 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. objektum-orientált programozás).

LOGIKAI MŰVELETEK

Logikai műveletek

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- *Lusta kiértékelésű* beépített operátorok
  - 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 értelmező, amikor szükség van rá.
- Mind három logikai operátor 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`!!!

Logikai műveletek (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 *operátorrá* alakítja.

LISTÁK

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;
  ```
  ```