Mit tanultunk eddig az SML-ről?

A következő néhány lapon összefoglaljuk, hogy az első két órán mit tanultunk az SML nyelvről és a funkcionális programozásról.

- Alapfogalmak
  (kifejezés, érték, típus; típuskifejezés; típusmegkötés)
- Értékdeklaráció, lokális kifejezés
- Egyszerű típusok
  (`int`, `real`, [`word`, `word8`],
  `string`, `char`, `bool`, `unit` stb.)
- Összetett adatstruktúrák és típusok
  (ennes; `'a option`, `'a list` stb.)

Alapfogalmak

Fontos alapfogalmak: *kifejezés, érték, típus*

- a *kifejezés* elemei:
  - *operátor* (függvényjel): infix, prefix; precedencia,
  - *operandus* (állandó, változó),
  - *zárójel*
- minden kifejezésnek meghatározott *értéke* és *típusa* van
- az értéket *értékdeklarációval* köthetjük tetszőleges *névhez*
- az SML-ben minden érték ún. *első osztályú érték,* azaz lehet
  - összetett adatstruktúra komponense,
  - művelet operandusa (függvény paramétere),
  - művelet (függvény) eredménye
- *függvényértéket* ad eredményül a *függvénykifejezés*
- egy kifejezés vagy érték típusát *típuskifejezéssel* adjuk meg
- a *típuskifejezés* elemei:
  - *típusoperátor:* infix, postfix; precedencia,
  - *típusoperandus* (típusállandó, típusváltozó),
  - *zárójel*
- a kifejezést az SML-értelmező *kiértékeli* [egyszerűsíti, redukálja] (ún. *read-eval-print* ciklusban)

Alapfogalmak (folyt.)

- a kiértékelés az SML-ben alapvetően *mohó*
- az egyszerűsítés során *egyenlőket helyettesítünk egyenlőkkel*
  – pl. a 3*4+5 kifejezést az értékével, 17-tel
- a kifejezés tovább nem egyszerűsíthető alakja a *kanonikus* kifejezés
  – az SML-értelmező válasza mindig kanonikus kifejezés

Az ún. *hivatkozási transzparencia* az olyan programozási nyelvek jellemzője, amelyben az egyenlők mindig helyettesíthetők egyenlőkkel. E tulajdonság megléte esetén egy kifejezés kiértékelésének eredménye *nem függ* részkifejezéseinek kiértékelési sorrendjétől.

A funkcionális program

- *mennyiségek közötti kapcsolatokat leíró egyenletek halmaza.*

Minden egyenletnek *kiszámítási szabálynak* kell lennie, amelyben a keresett mennyiség az egyenlőségjeltől balra és csak egyszer fordulhat elő.

- pl. `square x = x * x`

A programban olyan egyenlet *(állítás)* is használható, amelyet nem egy függvény definiálására, hanem valamilyen *tulajdonság* definiálására és ellenőrzésére használunk

- pl. `sqrt x * sqrt x = x`

Értékdeklaráció

Deklaráció: valamilyen *értéknek* (típusnak) *nevet* adunk, azaz egy nevet egy értékhez (típushoz) kötünk. Az SML-ben a kötés *statikus*, azaz a *fordítás során* jön létre. A név (azonosító) a továbbiakban ennek az értéknek (típusnak) a *szinonimája*.

Egy név *egyszerre csak egy* értéknek (típusnak) lehet a szinonimája, de a név jelentése új kötéssel megváltoztatható. Ugyanaz a név egyidejűleg jelenthet valmilyen értéket és típust (továbbá mezőt, struktúrát, szignatúrát és funktort is).

Néhány példa:

```
- val t = 3 * 4;       t itt a 12 szinonimája...
> val t = 12 :  int
- fun f1 () = t;       ...és itt is.
> val f1 = fn :  unit -> int
- val t = t + 4;       t itt már a 16 szinonimája...
> val t = 16 :  int
- fun f2 () = t;       ...azaz itt 16-ot jelent.
> val f2 = fn :  unit -> int
- f1();                f1 törzsébe a 12 épült be...
> val it = 12 :  int
```



```
- f2();                f2 törzsébe pedig a 16.
> val it = 16 :  int
```

Az előző sor az `f2()` *függvényalkalmazás eredményének* a típusát mutatja, a következő sor pedig magának az `f2` *függvénynek* a típusát.

```
- f2;
> val f2 = fn :  unit -> int
```

Jegyezzük meg: a `t` nem frissíthető változó, az értékdeklaráció nem értékadás.

Az értékdeklaráció egy változata a *függvénydeklaráció:*

- `val rec f = fn x => ...` helyett `fun f x = ...` használható (szintaktikus édesítőszer),
- a függvényjel alapértelmezésben *prefix* pozíciójú,
- deklarálható balra (`infix p ...`), ill. jobbra (`infixr p ...`) kötő, *infix* pozíciójú operátornak, `p` precedenciával,
- az *infix* pozícionálás átmenetileg kikapcsolható az op függvénnyel

Lokális kifejezés és lokális deklaráció

Lokális kifejezés: `let` *d* `in` *e* `end`, ahol

- d nem üres deklarációsorozat,
- e nem üres kifejezés

Lokális deklaráció: `local` *d1* `in d2 end`, ahol

- d1 és d2 nem üres deklarációsorozat

Belső és könyvtári függvények

- *belső* vagy *beépített* függvények (built-in functions)
- *előre definiált f*üggvények (predefined functions)
- *az SML-alapkönyvtárban* (SML Basis Library) definiált függvények

Egyszerű numerikus típusok

Megnevezés	*Hivatkozott egyszerű típusok*
realint	`int, real`
wordint	`int, word, word8`
num	`int, real, word, word8`

A `real` függvény egész számot alakít át valóssá.

A `floor` és a `ceil` függvények valós szám egész részét adják eredményül. `floor` a $-\infty$, `ceil` pedig a $+\infty$ felé kerekít, azaz `f = floor r` az a legnagyobb egész, amelyre `real f <= r`, `c = ceil r` pedig az a legkisebb egész, amelyre `real c >= r`.

Típusuk:

`real : int -> real`,

`floor` és `ceil : real -> int`.

A szemantikájuk *tulajdonságot definiáló egyenlőtlenséggel:*

```
real(floor r) <= r < real(floor r + 1)
real(ceil r)  >= r > real(ceil r - 1)
```

Egyszerű numerikus típusok (folyt.)

Az `abs` név *többszörös terhelésű,* mind `int`, mind `real` típusú értékre alkalmazható. Csak leírásokban alkalmazható típusspecifikációval:

abs : *realint* -> *realint*

Az SML-értelmező az `abs` név begépelésére az alábbi választ adja:

```
> val it = fn :  int -> int
```

Alapértelmezés szerint a többszörösen terhelhető nevek `int` típusú értéket várnak.

Szemantikája *kiszámítási szabályként* is használható *egyenlettel :*

abs x = max(x, ~x)

Kerekítésre használható a `round` és a `trunc` függvény. `round` a „legközelebbi" egész szám, `trunc` pedig a 0 felé kerekít.

`round` és `trunc :  real -> int`

Szemantikájuk *kiszámítási szabályként* is használható *egyenlettel :*

```
round r = floor(r + 0.5),
abs(real(trunc r)) <= abs r
```


Egyszerű numerikus típusok (folyt.)

Az *előjeles egész számokra* alkalmazható alapműveletek:

jele	*jelentése*	*jellege*	*pozíciója*
~	negatív előjel	monadikus	prefix
+	összeadás	diadikus	infix
-	kivonás	diadikus	infix
*	szorzás	diadikus	infix
div	osztás, $-\infty$ felé tart	diadikus	infix
mod	div maradéka	diadikus	infix
quot	osztás, 0 felé tart	diadikus	infix
rem	qout maradéka	diadikus	infix

Az egyoperandusú egészművelet típusa `int -> int`, a kétoperandusúaké `int * int -> int`.

> A `div` a `quot`-tal, a `mod` pedig a `rem`-mel azonos eredményt ad, ha két operandusuk *azonos* előjelű, egyébként az eredményük különböző. `a mod b` előjele `b` előjelével, `a rem b` előjele `a` előjelével azonos.

A `real` típusú számok *fixpontos* és *lebegőpontos* alakban is megadhatók, pl.

`0.01`   `~1.2E12`   `7E~5`

A kitevő tízes számrendszerbeli pozitív vagy negatív egész az `E` után.

Egyszerű numerikus típusok (folyt.)

A *valós számokra* alkalmazható alapműveletek:

jele	*jelentése*	*jellege*	*pozíciója*
`~`	negatív előjel	monadikus	prefix
`+`	összeadás	diadikus	infix
`-`	kivonás	diadikus	infix
`*`	szorzás	diadikus	infix
`/`	osztás	diadikus	infix

Az egyoperandusú valósművelet típusa `real -> real`, a kétoperandusúaké `real * real -> real`.

A `~`, `+`, `-` és `*` műveleti jelek tehát *többszörösen terhelt* nevek.

A `real` nem *egyenlőségi típus* (az smlnj-ben), azaz `real` típusú értékekre nem alkalmazhatók az = és a <> relációk.

Szemelvények a `Math` könyvtárból

név	*jelentés*	*típus*
`pi`	a π állandó	`real`
`e`	az e állandó	`real`
`sqrt`	sqrt x = x négyzetgyöke	`real -> real`
`sin`	sin z = z szinusza	`real -> real`
`cos`	cos z = z koszinusza	`real -> real`
`tan`	tan z = z tangense	`real -> real`
`exp`	exp x = e x-edik hatványa	`real -> real`
`pow`	pow(x, y) = x az y-odikon	`real * real -> real`
`ln`	ln x = x e alapú logaritmusa	`real -> real`
`log10`	log10 x = x 10 alapú log-a	`real -> real`

z-t radiánban kell megadni.

`ln` és `log10` esetén $x > 0.0$ a feltétel.

Az *infix* operátoroknál erősebben kötnek a *prefix* operátorok. Pl. exp $a + b =$ (exp $a) + b \neq$ exp $(a + b)$.

Infix operátorok precedenciája

Az *infix* operátorok precedenciáját 0 és 9 közötti egész számokkal adjuk meg (a 9-es szint a legmagasabb). A nagyobb precedenciájú operátor erősebben köt.

operátor	*típus*	
7-es szintű precedencia		
*	*num* * *num* -> *num*	
/	real * real -> real	
div, mod	*wordint* * *wordint* -> *wordint*	
quot, rem	int * int -> int	
6-os szintű precedencia		
+, -	*num* * *num* -> *num*	
4-es szintű precedencia		
=, <>	''a * ''a -> bool	
<, <=, >=, >	*numtxt* * *numtxt* -> bool	

Típusmegkötés

A *típuslevezetés* (type inference) a *többszörösen terhelhető nevű* belső függvények (operátorok) esetében nem lehetséges.

Alapértelmezés szerint a többszörösen terhelhető nevek `int` típusú értéket várnak. Ha az alapértelmezéstől el akarunk térni, *típusmegkötést* kell használni, például:

```
fun sq'r (x:  real) = x * x
```


Egyszerű típusok: füzér és karakter

A `string` típusú füzért idézőjelek között álló, esetleg üres karaktersorozattal jelöljük. Különleges karakterek ún. *escape-szekvenciákkal* írhatók le.

Példák: `"abraka"`, `""`, `"\r"`, `"\065"`.

jelölés	*escape-szekvencia neve*	*dec. ASCII-kódja*
\a	csengő (alert)	7
\b	visszalépés (backspace)	8
\t	vízszintes (horizontális) tab.	9
\n	újsor-jel (newline)	10
\v	függőleges (vertikális) tab.	11
\f	lapdobás (form feed)	12
\r	sorelejére-jel (return)	13
\^c	vezérlőkarakter	ord c - 64
\ddd	tetszőleges karakter	ddd
\"	idézőjel (double quote)	34
\\	hátratört-vonal (backslash)	92
\w...w\	figyelmen kívül hagyandó	-

Két füzért fűz össze az *infix* pozíciójú ^ operátor, típusa `string * string -> string`. Füzér hosszát adja eredményül a `size` függvény, típusa `string -> int`.

`string` típusú értékek is összehasonlítók a szokásos relációs operátorokkal (<, <=, =, >=, >, <>).

Egyszerű típusok: karakter (folyt.)

Az SML-ben a karaktertípust a `char` típusnévvel jelöljük.

jelölés	magyarázat	ASCII-kód
`#"a"`	a kis a betű	97
`#"Z"`	a nagy Z betű	90
`#"0"`	a 0 számjegy	48
`#"^G"`	a ^G-vel jelölt vezérlőkarakter	7
`#"\007"`	a 007 kódú karakter	7
`#"\a"`	a csengő jele	7

Karakter ASCII-kódját állítja elő az `ord : char -> int`, adott ASCII-kódú karaktert ad vissza a `chr : int -> char` függvény.

`char` típusú értékek is összehasonlítók a szokásos relációs operátorokkal (<, <=, =, >=, >, <>).

Egyszerű típusok: igazságérték (f.)

Csak kétféle `bool` típusú állandó van, jelölésük: `true` és `false` (ún. adatkonstruktorállandók).

Igazságértéket adnak eredményül a *relációs operátorok:*

- előjeles és előjel nélküli egészekre, valósakra, karakterekre, füzérekre: <, <=, >=, >,
- az ún. *egyenlőségi típusokra* (equality types): =, <>.

Feltételes és logikai operátorok

- Háromoperandusú feltételes operátor : `if ... then ... else ...`
- Kétoperandusú logikai operatorok: `... andalso ...` , `... orelse ...`
- Mindhárom *lusta* kiértékelésű.
- Egyoperandusú logikai operator: `not`

`orelse` precedenciája kisebb `andalso` precedenciájánál, és mindkettőé kisebb bármely más *infix* operátor precedenciájánál. A `not` precedenciája a lehető legnagyobb, mert prefix pozíciójú .

Tesztelő függvények

A `Char` könyvtárban több függvény található karakterek osztályozására. Ezek általában `char -> bool` típusúak.

A függvények szemantikáját a `Char` könyvtárbeli `contains : string -> char -> bool` függvény segítségével adjuk meg. `contains s c` akkor igaz, ha a `c` karakter benne van az `s` füzérben. Az alábbi táblázatban a tesztelő függvények egyetlen argumentumát `c`-vel jelöljük a jelentésük megadásakor.

függvénynév	*jelentés*
`isLower`	`contains "abcdefghijklmnopqrstuvwxyz" c`
`isUpper`	`contains "ABCDEFGHIJKLMNOPQRSTUVWXYZ" c`
`isDigit`	`contains "0123456789" c`
`isAlpha`	`isUpper c orelse isLower c`
`isHexDigit`	`isDigit c orelse contains "abcdefABCDEF" c`
`isAlphaNum`	`isAlpha c orelse isDigit c`
`isPrint`	`c` látható karakter vagy szóköz (`#" "`)
`isSpace`	`contains " \t\r\n\v\f" c`
`isPunct`	`isPrint c andalso not(isSpace c orelse isAlphaNum c)`
`isGraph`	`not(isSpace c) andalso isPrint c`
`isAscii`	`0 <= ord c <= 127`
`isCntrl`	`not(is Print c)`

Ennesek és rekordok

Az *ennes* (angolul: *tuple*)

- elemeit vessző választja el,
- elemeinek típusa tetszőleges lehet,
- elemei különböző típusúak lehetnek,
- elemeinek sorrendje fontos:
  - az ennes típusát elemeinek száma, típusa és sorrendje szabja meg.

Példák:

```
- ("Laca", 18);
> val it = ("Laca", 18) : string * int
- (18, "Laca");
> val it = (18, "Laca") : int * string
```

Egy ennes elemeit elegánsan *mintaillesztéssel* azonosíthatjuk. Példa:

```
- val (xc, yc) = scalevec (4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

A *nullas* olyan ennes, amelynek egyetlen eleme sincs:

- a nullast (angolul *zero tuple)* a `()` jellel jelöljük,
- ez a `unit` típus egyetlen eleme.

Ennesek és rekordok (folyt.)

A *rekord* olyan ennes, amelyben

- az egyes elemeket megcímkézzük, tehát a *sorrend nem fontos;*
- az elemekre nem a helyük szerint, hanem a *címkéjükkel* hivatkozunk;
- a rekord *típusát* elemeinek *száma, típusa* és *címkéje* szabja meg;
- az ennes olyan rekord, amelyben a címkék „láthatatlan” természetes számok.

Ennesek és rekordok (folyt.)

Példák:

```
- val empl = {name = "Jones", age = 25,
              salary = 15300};
> val empl = {age = 25, name = "Jones",
              salary = 15300} :
      {age : int, name : string,
       salary : int}
- val empl = {name = "Jones",
              salary = 15300, age = 25};
> val empl = {age = 25, name = "Jones",
              salary = 15300} :
      {age : int, name : string,
       salary : int}
- #name empl;
> val it = "Jones" : string
- #age empl;
> val it = 25 : int
```

Ennesek és rekordok (folyt.)

Példa: az ennes mint rekord

```
- val negyes =
      {1="a", 2="b", 3="c", 4="d"};
> val negyes = ("a", "b", "c", "d") :
    string * string * string * string
- val negyes =
      {3="c", 4="d", 2="b", 1="a"};
> val negyes = ("a", "b", "c", "d") :
    string * string * string * string
- val negyes = ("a", "b", "c", "d");
> val negyes = ("a", "b", "c", "d") :
    string * string * string * string
- #1 negyes;
> val it = "a" :  string
- #4 negyes;
> val it = "d" :  string
```

Címke helyett elegánsabb mintaillesztést használni:

```
- val (a, b, c, d) =
         (#"a", #"b", 3, false);
> val a = #"a" :  char
> val b = #"b" :  char
> val c = 3 :  int
> val d = false :  bool
```


Típusegyenletek

A `string * int` és `int * string` *típuskifejezések.*

A típuskifejezés elemei

- *típusállandók* (`int`, `real`, `string` stb.),
- *típusváltozók* (`'a`, `'b` stb.),
- *típusoperátorok* (`*`, `->`, `list` stb.).

A típusoperátoroknak is van precedenciájuk:

- a *postfix* típusoperátoroké a legnagyobb,
- a *keresztszorzaté* (Descartes-szorzat, jele a `*`) kisebb,
- a *leképzésé* (jele a `->`) a legkisebb.

Típuskifejezésben is használható *zárójel* a műveletek sorrendjének meghatározására.