SML-SZINTAXIS

SML-szintaxis: különleges állandók

- Előjeles egész állandó

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

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

- 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`

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

- Karakterállandó: # jelet közvetlenül követő, egykarakteres füzérállandó.

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

  Ellenpéldák: `# "a" #c #"""`

SML-szintaxis: escape-szekvenciák

- 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.

SML-szintaxis: név

- Alfanumerikus: kis- és nagybetűk, számjegyek, percjelek (`'`) és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik
  - ◇ Példák: `tothGyorgy` `Toth_3_Gyorgy` `toth'gyorgy`
- Szimbolikus: az alábbi jelek tetszőleges, nem üres sorozata

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

  - ◇ Példák: `++` `<->` `|||` `##` `|=|`
- Speciális a szerepe az alábbi fenntartott jeleknek

```
( ) [ ] { } , ; . ...
```

- Más jelentés nem rendelhető az ún. fenntartott nevekhez

```
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 SML-ben

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)`	a két függvény kompozíciója	
0	`before`	`'a * 'b -> 'a`	a bal oldali argumentum	

LISTÁK

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 tudjuk a végéhez fűzni. A lépések száma $O(n^2)$.

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


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.

Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy n db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható*, *prefix* pozíciójú *függvényt*) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:

```
foldr op* 1.0 [] = 1.0;                    foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0;                 foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0; foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Jelöljön $\oplus$ tetszőleges kétoperandusú infix operátort. Akkor

```
foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
foldr op⊕ e [] = e
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
foldl op⊕ e [] = e
```

- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

Példák `foldr` és `foldl` alkalmazására

- A $\oplus$ művelet `e` operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- `isum` egy egészlista elemeinek összegét, `rprod` egy valóslista elemeinek szorzatát adja eredményül.

```
val isum = foldr op+ 0;          val rprod = foldr op+ 1.0;
val isum = foldl op+ 0;          val rprod = foldl op+ 1.0;
```

- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;
(* lengthl, lengthr : 'a list -> int *)
val lengthl = fn ls => foldl inc 0 ls;
fun lengthr ls = foldr inc 0 ls;

lengthl (explode "tengertanc");
lengthr (explode "hajdu sogor");
```

Lista: `foldr` és `foldl` definíciója

- `foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))`
  `foldr op⊕ e [] = e`

```
(* foldr f e xs = az xs elemeire jobbról balra haladva
                  alkalmazott, kétoperandusú, e egységelemű
                  f művelet eredménye
   foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
  | foldr f e [] = e;
```

- `foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)`
  `foldl op⊕ e [] = e`

```
(* foldl f e xs = az xs elemeire balról jobbra haladva
                  alkalmazott, kétoperandusú, e egységelemű
                  f művelet eredménye
   foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
  | foldl f e [] = e;
```


Újabb példák `foldr` és `foldl` alkalmazására

- Egy lista elemeit egy másik lista elé fűzi `foldr` és `foldl`, ha kétoperandusú műveletként a *cons* konstruktorfüggvényt – azaz az `op::`-ot – alkalmazzuk.

  `foldr op:: ys [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))`
  `foldl op:: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: ys)))`

- A `::` nem asszociatív, ezért `foldl` és `foldr` eredménye különböző!

```
(* append : 'a list -> 'a list -> 'a list
   append xs ys = az xs ys elé fűzésével előálló lista *)
fun append xs ys = foldr op:: ys xs;

(* revApp : 'a list -> 'a list -> 'a list
   revApp xs ys = a megfordított xs ys elé fűzésével
                  előálló lista *)
fun revApp xs ys = foldl op:: ys xs;
```

`append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6];` (vö. Prolog: `append`)
`revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6];` (vö. Prolog: `revapp`)

Lista redukciója bal oldali egységelemű függvénnyel (`foldL`)

- A kivonás művelete balra köt: $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$.
- Nem feleltethető meg sem `foldr`-nek, sem `foldl`-nek.

```
foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
```

- Nevezzük `foldL`-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy $\oplus$ *bal oldali* egységelemet vár.

```
foldL op⊕ e [x1, x2, ..., xn] =
     ( ... ((e ⊕ x1) ⊕ x2) ⊕ ... ⊕ xn)
```

- `foldL` olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az *első* argumentuma: `f : 'a * 'b -> 'a`.

```
(* foldL : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
   foldL f e xs = az xs elemeire balról jobbra haladva
                  alkalmazott, kétoperandusú, e egységelemű
                  f művelet eredménye *)
fun foldL f e (x::xs) = foldL f (f(e, x)) xs
  | foldL f e [] = e;
```


Példák listaelemek különbségének és hányadosának képzésére

- Az `e` argumentum aktuális értéke a sorozat *első* eleme – a *kisebbítendő*, ill. az *osztandó*.

```
foldL op- 20 [] = 20;
foldL op- 20 [5, 6, 7] = (((20 - 5) - 6) - 7);
foldL (op div) 180  [] = 180;
foldL (op div) 180  [2, 3, 5] = (((180 div 2) div 3) div 5);
```

- Ha többször használjuk e műveleteket, érdemes nekik nevet adni. A kisebbítendő, ill. az osztandó speciális kezelését elrejtjük.

```
fun subtract ns = foldL op- (hd ns) (tl ns);
subtract [20, 5, 6, 7] = (((20 - 5) - 6) - 7);

fun divide ns = foldL op div (hd ns) (tl ns);
divide [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

Listaelemek különbsége és hányadosa `foldl`-lel és `foldr`-rel

- Igazság szerint `foldL` felesleges: a feladat jól megoldható `foldl`-lel vagy `foldr`-rel is.

```
fun subtract1 ns = hd ns - foldl op+ 0 (tl ns);
subtract1 [20, 5, 6, 7] = (((20 - 5) - 6) - 7);

fun divide1 ns = hd ns div foldl op* 1 (tl ns);
divide1 [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

- `foldr` és `foldl` típusa, ha egyparaméteres függvénynek tekintjük őket (a `->` jobbra köt!):

  `foldr, foldl : ('a * 'b -> 'b) -> ( 'b -> 'a list -> 'b)`

  Azaz ha `foldr`-t vagy `foldl`-t egy `'a -> * 'b -> 'b` típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'b` típusú egységelemre és egy `'a list` típusú listára alkalmazva `'b` típusú (redukált) értéket kapunk.