Irodalom

- Robert Harper: Programming in Standard ML
  http://www-2.cs.cmu.edu/~rwh/smlbook/
- Riccardo Pucella: Notes on Programming Standard ML of New Jersey
  http://www.cs.cornell.edu/riccardo/smlnj.html
- Standard ML: http://www.standardml.org
- SML of New Jersey: http://www.smlnj.org

Lexikai elemek

- Komment: `(* ... *)`
- Azonosító:
  - Alfanumerikus: betűk, számok, aposztrófok és aláhúzásjelek
  - Szimbólumsorozat: `! % & $ # + - / * : < = > ? @ \ ~ ` ^ |`
- Hosszú azonosítók: `Modul.Modul.Azon`
- Kulcsszavak, pl. `if, let, type, =>, #`

Alaptípusok

- `unit:` egyetlen értéke a `()`
- `int:` egész számok: `0, 1, 2, 0xbc`
  - Negatív számok: `~1, ~2`
  - Aritmetikai műveletek: `+, -, *, div, mod`
  - Logikai műveletek: `=, <>, <, >, <=, >=`
- `word:` pozitív egészek
  - Konstansok: `0w12, 0wx1f`
  - Műveletek: ua., mint az `int`

További alaptípusok

- `bool:` logikai típus: `true, false`
  - `if ... then ... else`
  - `andalso, orelse, not`
- `real:` lebegőpontos számok: `1.0, 2.1E~4`
  - Műveletek: `+, -, *, /, <, >, <=, >=`
- `char:` karakter típus: `#"a", #"\n"`
- `string:` szöveg: `"", "alma\n"`
  - Műveletek: `=, <>, <, >, <=, >=, ^`

Deklarációk

- Változók: értéket lehet rendelni egy névhez, amit viszont később nem lehet megváltoztatni

```
val (i: int) = 3
val j = 2*i
and s = ″hat″
```

- Típusnév bevezetése:

```
type float = real
and count = int
```


Függvények

- SML-ben a függvények is egyszerű, futási időben is létrehozható értékek. Az `fn` kulcsszóval lehet leírni egy függvényt λ-kifejezésként, a függvénydeklarációkat pedig rövidíteni is lehet:

```
val f = (fn x => x+1)
fun f x = x+1
```

- Függvény alkalmazásához a függvény neve után kell írni az aktuális paraméterét:

```
f 2
```


Típuslevezetés

- Az előző `f` függvény típusa: `int -> int`. Ezt a típust a függvény törzse alapján egyértelműen meg lehet állapítani, és ezt az SML értelmező meg is teszi minden függvény esetében.
- Az SML szigorúan típusos nyelv, ha pl. egy `int` típusú formális paraméter helyére `real` típusú aktuális paramétert írunk, az fordítási hiba.
- Felhasználói túlterhelés nincs, mert aláássa a típuslevezetést.

Eljárások

- SML-ben léteznek mellékhatások, és nem teljesül a hivatkozási átlátszóság. Lehet például eljárásokat is készíteni, ezek `unit` típusú eredménye van. Ilyen például a `print` eljárás, aminek `string->unit` a típusa.
- Eljárásokhoz szükséges a *szekvencia* konstrukció. Az `(E1;E2)` kifejezés kiértékeli `E1`-et, majd `E2`-t, és `E2` értéke lesz az eredmény.
- `(E1 before E2)` kif.: `E1` értéke az eredmény

Polimorf függvények

- Lehet olyan függvényt készíteni, aminek a törzse alapján bármilyen típusú paramétere lehet, pl.:

```
fun ID x = x
```

- Ebben az esetben a paraméter levezetett típusát egy *típusváltozó* fogja jelölni. A függvény alkalmazásakor automatikusan a megfelelő típust helyettesíti be a fordító. Az `ID` függvény levezetett típusa `'a->'a` (ejtsd: α), az `(ID 1)` kifejezésben pedig már az `int->int` típust alkalmazza a fordító.

Egyenlőségvizsgálat

- A polimorf függvények teljes mértékben függetlenek a típusváltozóik értékétől, tehát a függvény törzse semmilyen típusműveletet nem használhat
- Kiemelt művelet az egyenlőségvizsgálat, ami sok típusra értelmezett, de nem mindre. *Egyenlőségjeles típusnak* hívjuk ezeket, és ha egy függvénynek ilyen típusra van szüksége (tehát használja az `=` operátort), azt speciális típusváltozó jelöli: `''a, ''b, ...`

Magasabb rendű függvények 1.

- Egy függvényt magasabb rendűnek nevezünk (többek között) akkor, ha eredménye egy újabb függvény. Például:

```
fun adder a =
   (fn x => a+x)
```

- Az `adder` típusa `int -> (int -> int)`, minden `a` egész számhoz egy `int -> int` típusú függvényt ad eredményül, ami a paraméteréhez hozzáadja `a`-t.

Függvények több paraméterrel

- Az adder függvény tekinthető két paraméteres függvénynek is, amit így kell alkalmazni:

```
(adder x) y  (* x és y összege *)
```

Egy függvény ilyen átalakítása a *currying*.

- Rövidített jelölés:

```
fun adder a b = a+b
adder x y
```

Ez ugyanazt jelenti, mint az előző jelölés.

Direkt szorzat típus

- Rendezett *n*-esek típusa: `(T1 * T2 * T3)`
- Rendezett *n*-es létrehozása: `(v1, v2, v3)`
- Szelekciós függvény: `#1 tup, #2 (1,2,3)`
- Több függvényparaméter ill. eredmény:

```
fun addsub (a, b) = (a+b, a-b)
val (sum, dif) = addsub (x, y)
```

- Az `addsum` típusa: `int*int -> int*int`

Mintaillesztés

- A mintaillesztés segítségével összetett adatokat bonthatunk szét a konstrukciós műveleteik mentén:

```
val v = (addsub (1,2), addsub (2,1))
val ((s1,d1),(s2,d2)) = v
```

- A `case` kifejezéssel több minta illesztését lehet megpróbálni (_ a joker):

```
case addsub (x,y)
  of (0,_) => "ellentett"
   | (_,0) => "azonos"
   | _ => "különböző"
```


Mintaillesztés 2.

- Függvényparaméterek mintaillesztésére egyszerűsített szintaxis:

```
fun f (0,y) = y
  | f (x,_) = x
```

- Egy részmintának önálló nevet is lehet adni:

```
fun f (v as (0,y), _) = v
  | f (_, v) = v
```

- Ha egyik mintára sem illeszthető a paraméter, az futási idejű hibát okoz

Alaptípusok további műveletei

- `size: string->int`
  `substring: string*int*int->string`
  `str: char->string`
- `ord: char->int, chr: int->char`
- `Int.toString: int->string`
  `Int.fromString: string->int`
- `Real.posInf, Real.negInf`
  `Real.Math.pi, Real.Math.e`
  `Real.Math.sin, Real.Math.cos,...`
  `Real.toString, Real.fromString`
  `Real.toInt, Real.fromInt`

Infix műveletek

- Az infix műveletek típusa mindig `('a*'b) -> 'c` alakú
- Tetszőleges azonosítót lehet infix operátorként használni: `infix N oper` utasítás után balról, `infixr N oper` után jobbról zárójelezett műveletet definiálhatunk, pl. `infix 7 div` vagy `infixr 5 ::`
- Infix operátor, mint függvény: pl. `op +`
- Infix státusz megszüntetése: `nonfix +`

Deklarációk hatóköre

- A változók értéke nem változik, ugyanazt a nevet viszont lehet több deklarációban használni, ilyenkor az új deklaráció *eltakarja* a régit.
- Minden deklaráció hatóköre *statikus*, a kiértékelési sorrendtől független.

```
val x = 1;
fun f (x:int) = x+x;
fun g (y:int) = x+y;
val (x,y) = (2,3);
(* Mi x, y, f 5, g 6 értéke? *)
```


Lokális deklarációk

- Deklarációk hatókörét korlátozni lehet egy másik deklarációra, ezzel lényegében lokális függvényeket (típusokat, változókat stb.) lehet létrehozni:

```
local
  fun fact' (0,p) = p
    | fact' (n,p) = fact' (n-1,n*p)
in
  fun fact n = fact' (n,1)
end
(* fact' itt már nincs *)
```


Lokális deklarációk 2.

- Kifejezésre nézve lokális deklaráció is létezik:

```
fun gyokok (a,b,c) =
let
  val D = Math.sqrt (b*b - 4.0*a*c)
in
  if D >= 0 then
    ((-b-D)/(2.0*a),(-b+D)/(2.0*a))
  else
    (Real.posInf, Real.posInf)
end
(* A ”D” itt már nem érhető el *)
```

Rekurzió

- Rekurzív hivatkozás `val rec` deklarációval:

```
val rec fact = fn 0 => 1
  | n => n*fact (n-1)
```

- A `fun` valójában a `val rec` rövidítése

```
fun fact 0 = 1
  | fact n = n*fact (n-1)
```

- Érdemes végrekurziót alkalmazni

```
fun fact n = let
  fun fact' (0,p) = p
    | fact' (n,p) = fact' (n-1,n*p)
in fact' (n,1) end
```


Rekord

- A rekord a direkt szorzat általánosítása: a komponensekhez nevet lehet rendelni, így több adat is könnyen kezelhető
- Rekord érték létrehozása:

```
{nev="Kis Pál",cim="Bp.", szul=1980}
```

- Ennek a rekordnak a típusát így írhatjuk le:

```
type szemely =
 {nev:string, cim:string, szul:int}
```

- Szelekciós függvény: `#nev, #cim`

Rekord minták

- Teljes minta:

```
fun nevcim {nev=n, cim=c, szul=_}
  = n ^ ", " ^ c
```

- Rövidítési lehetőség a változók nevére:

```
fun nevcim {nev, cim, szul}
  = nev ^ ", " ^ cim
```

- A nem használt mezők elhagyhatók, ha ismert a pontos típus:

```
fun nevcim ({nev, cim, ...}:szemely)
  = nev ^ ", " ^ cim
```


Algebrai típus

- Az algebrai adattípus az egyetlen eszköz új típus létrehozására. Minden ilyen deklaráció létrehoz egy típuskonstruktort és (általában több) adatkonstruktort.
- A legegyszerűbb példa a felsorolási típus:

```
datatype szin =
  makk | tok | zold | piros
```

- A mintaillesztés működik adatkonstruktorokkal:

```
fun duplaz piros = true
  | duplaz _ = false
```

Paraméteres adattípus

- Az adatkonstruktoroknak lehet paraméterük:

```
datatype int_real =
  INT of int | REAL of real
val n = INT 1
fun toStr (INT i) = Int.toString i
  | toStr (REAL r) = Real.toString r
```

- A paraméteres adatkonstruktor függvényként is használható, például az `INT` típusa függvényként `int -> int_real`.

Polimorf adattípus

- A típuskonstruktornak is lehet paramétere, egy vagy több típusváltozó:

```
datatype 'a option = NONE | SOME of 'a
```

- Ahol szükséges, ott a fordító automatikusan behelyettesíti a konkrét típust, például a `SOME 1` kifejezés típusa `int option`.
- Több típusparaméter is megadható:

```
datatype ('a,'b) two_opt =
  ZERO | ONE of 'a | TWO of 'a*'b
```


Rekurzív adattípus

- Az adatkonstruktorok paraméterei használhatják az éppen definiált típust:

```
datatype 'a tree = LEAF
  | NODE of 'a * 'a tree * 'a tree
```

- Az ilyen típusokat természetesen rekurzív függvényekkel dolgozzuk fel általában:

```
fun travel _ LEAF = ()
  | travel f (NODE (v, l, r)) =
      (f v; travel f l; travel f r)
```


Lista típus

- Algebrai adattípusként a következőképpen lehetne definiálni:

```
datatype datatype 'a list =
  nil | :: of 'a * 'a list
```

- Speciális jelölések:
  - Üres lista: `[]` (ua. mint `nil`)
  - Egy elemű lista: `[1]` (ua.mint `1::nil`)
  - Több elemű lista: `[1,2,3]` (ua. mint `1::2::3::nil`)

Egyszerű listaműveletek

```
fun null nil = true
  | null _ = false
fun hd nil = raise Empty
  | hd (h::_) = h
fun tl nil = raise Empty
  | tl (_::t) = t
fun length nil = 0
  | length (_::t) = (length t)+1
fun @ (nil, l) = l
  | @ (h::t, l) = h::(t @ l)
fun rev nil = nil
  | rev (h::t) = (rev t) @ [h]
```


Általánosítható függvények

```
fun sum nil = 0
  | sum (h::t) = h + (sum t)
fun prod nil = 1
  | prod (h::t) = h * (prod t)
fun intReal nil = nil
  | intReal (h::t) =
    (Real.fromInt h) :: intReal t
fun mul2 nil = nil
  | mul2 (h::t) = (2*h)::mul2 t
fun printAll nil = ()
  | printAll (h::t) =
     (print h; printAll t)
```


Magasabb rendű függvények 2.

- Magasabb rendűnek hívunk egy függvényt akkor is, ha egy paramétere függvény, például:
  ```
  fun map fv [] = []
    | map fv (h::t) =(fv h)::(map fv t)
  ```
  Itt az `fv` egy `'a->'b` típusú függvény, a második argumentum egy `'a list`, az eredmény pedig egy `'b list`.
- Az előző oldal függvényei:
  ```
  val intReal = map (Real.fromInt)
  val mul2 = map (fn x => 2*x)
  ```


Általános listafüggvények

```
fun app _ nil = ()
  | app fv (h::t) = (fv h; printAll t)
val printAll = app print

fun foldr _ egys nil = egys
  | foldr fv egys (h::t) =
     fv h (foldr fv egys t)
val sum = foldr (op +) 0
val prod = foldr (op *) 1
val concat = foldr (op ^) ""
```

Kivételek

- A kivételek `exn` típusú elemek, lehet paraméterük is. Deklarációjuk formája:

```
exception Empty
exception Error of string
```

- Az így deklarált `Error` név `string->exn` típusú konstrukciós függvényként használható.
- A `raise` kulcsszóval lehet kivételt kiváltani:

```
fun hd nil = raise Error "Üres"
  | hd (h::t) = h
```


Kivételek kezelése

- Minden kifejezés végére lehet tenni egy kivételkezelőt a `handle` kulcsszóval. A kezelő feladata, hogy megadja a kifejezés értékét a kivételes esetben (típushelyesen).
- A különböző kivételeket ill. paramétereiket mintaillesztés segítségével lehet kezelni.

```
( … (* int típusú kifejezés *) … )
  handle Empty => 0
    | Error msg => (print msg; 0)
```


Változó értékek tárolása

- SML-ben lehetőség van változtatható értékű memóriacellák használatára, típusos referenciák segítségével.
- Referencia típus műveletei:
  - Konstrukció: `ref: 'a -> 'a ref`
  - Értékadás: `:= : ('a ref * 'a) -> unit`
  - Értéklekérdezés: `!: 'a ref -> 'a`
- Az értékadás infix művelet

Referenciák lehetőségei

- Létezik referencia minta, lekérdező műveletekben kényelmesen helyettesíti a `!`-t:

```
fun += (_, ref 0) = ()
  | += (a, ref b) = a := !a + b
```

- A `while felt do kif` kifejezéssel a hagyományos ciklust írhatjuk le:

```
val i = ref 1
while !i < 10 do
 (print (Int.toString (!i));
  i := !i + 1)
```

Rejtett állapot

- Belső állapottal rendelkező objektum létrehozására mutat példát ez a függvény:

```
fun counter () =
  let
    val c = ref 0
    fun t () = (c := !c + 1; c)
    fun r () = (c := 0)
  in
    { tick = t, reset = r }
  end
```


Rejtett állapot 2.

- Az objektum így használható:

```
val cnt1 = counter ()
val cnt2 = counter ()
#tick cnt1 ()     (*   = 1   *)
#tick cnt2 ()     (*   = 1   *)
#reset cnt1 ()
#tick cnt2 ()     (*   = 2   *)
#tick cnt1 ()     (*   = 1   *)
```

- Fontos, hogy minden példánynak saját állapota van, nem pedig közös az összesnek.

Module language

- Szignatúra: egy modul interfészének leírása, típusinformációk minden elemről
- Struktúra: egy modul implementációja
- Minden struktúrának van egy elsődleges szignatúrája, de ennek a láthatóságát lehet korlátozni egy szigorúbb szignatúrával
- Funktor: struktúrák közötti függvény, generikus struktúrák megvalósításának eszköze

Szignatúrák

- A szignatúra specifikációk sorozata: típusnevek, algebrai típusok, kivételek és értékek leírása.
- Típus specifikációja: `type name` vagy `type name=`*`leírás`* (típusparaméter lehet) `eqtype`: egyenlőségjeles típus jelölése
- Algebrai típus: `datatype` (ld. deklaráció)
- Kivételek: `exception` (ld. deklaráció)
- Értékek: `val id : típus` (függvények is!)

Szignatúra definíció

- Polimorf verem típus szignatúrája:

```
signature STACK = sig
  type 'a stack
  exception Empty
  val stack: 'a stack
  val empty: 'a stack -> bool
  val push: 'a stack * 'a -> 'a stack
  val pop: 'a stack -> 'a * 'a stack
end
```


Származtatott szignatúrák

- Szignatúra kiterjesztése:

```
signature STACK_TOP = sig
  include STACK
  val top: 'a stack -> 'a
end
```

- Specializáció: hiányzó típusinformációt lehet megadni utólag

```
signature STACK_LST =
  STACK where type
        'a stack = 'a list
```


Struktúrák

- A struktúra deklarációk sorozata. A struktúra típusát egy szignatúra írja le. Típuskonstruktorok, adattípusok, kivételek és értékek deklarációit tartalmazhatja.
- Struktúra definíciója:

```
structure Stack = struct
  type 'a stack = 'a list
  exception Empty
  fun pop nil = raise Empty
    | pop (h::t) = (h,t)
end
```


Struktúrák elemeinek elérése

- Lehet használni minősített neveket, pl. `Stack.pop`
- Az `open Stack` direktíva minősítés nélkül elérhetővé teszi a struktúra elemeit. A mellékhatások elkerülése érdekében célszerű csak lokálisan használni:

```
let
  open Stack
  ...
in pop(s) end
```

Szignatúrák megfeleltetése

- Egy szignatúra *megfelel* egy másiknak, ha az
  - kevesebb komponenst tartalmaz,
  - egy érték típusában típusváltozó helyett konkrét típust követel meg,
  - elhagyja egy típus leírását, vagy adattípus helyett csak egy típuskonstruktort tartalmaz,
  - vagy más sorrendben tartalmaza a komponenseit.
- Egy struktúra *elsődleges szignatúrája* a struktúra komponenseinek a pontos típusát írja le.

Szignatúrák struktúrához rendelése

- Egy struktúra alapértelmezett interfésze az elsődleges szignatúrája.
- Ezt az interfészt le lehet szűkíteni egy olyan szignatúrával, aminek megfelel az elsődleges szignatúra. Ezt hívjuk szignatúra hozzárendelésnek (ascription).
- Két típusa van: *átlátszó,* ami csak a műveleteket rejti el, és *átlátszatlan,* ami a típusokat is átlátszatlanná teszi.

A hozzárendelés alakja

- Az átlátszó hozzárendelést kettőspont jelzi:

```
structure Stack : STACK = struct
  type 'a stack = 'a list
  ...
end
```

- Az átlátszatlannak `:>` a jelölése:

```
structure Stack :> STACK = struct
  type 'a stack = 'a list
  ...
end
```


Belső struktúrák

- Egy struktúra az eddig felsorolt deklarációkon kívül tartalmazhat struktúrákat is.
- A belső struktúrák elemeit többszörösen minősített névvel lehet elérni: `Str1.Str2.elem`
- A belső struktúrák a szignatúrában is megjelennek, a specifikációjuk alakja:
  `structure StrucName : SIG_NAME`
- A belső struktúrákat gyakran arra használjuk, hogy a szignatúrák ne függjenek külső elemektől

Példa a szignatúrák használatára

- Definiáljuk egy asszociatív tömb interfészét!

```
signature MAP = sig
  type (''a,'b) map
  val empty: (''a,'b) map
  val insert:
    (''a,'b)map*''a*'b -> (''a,'b)map
  val lookup:(''a,'b)map*''a -> 'b
end
```

- Hiányosság: nem tudjuk hatékonyan reprezentálni, ahhoz kellene egy művelet a kulcshoz

Példa - folytatás

- Tudunk adni olyan szignatúrát, ami lehetővé tesz hatékonyabb implementációkat:

```
signature MAP = sig
  type key
  type 'a map
  val lookup:'a map*key -> 'a
end
signature MAP_STR =
  MAP where type key = string
```

- Hiányosság: a szignatúra nem írja le teljesen az interpretációt, mert hiányzik belőle a rendezés

Példa - folytatás

- A rendezési műveletet egy struktúra írhatja le:

```
signature ORD = sig
  eqtype ty
  val lt: ty*ty -> bool
end
signature ORD_STR =
  ORD where type ty = string
structure OrdStr : ORD = struct
  type ty = string
  val lt = (op<):ty*ty -> bool
end
```


Példa - folytatás

- Hogy használhatjuk fel az `OrdStr` struktúrát?

```
signature MAP = sig
  structure Key: ORD
  type 'a map
  val lookup: 'a map*Key.ty -> 'a
end
signature MAP_STR =
  MAP where type Key.ty = string

structure MapStr :> MAP_STR = struct
  structure Key = OrdStr
end
```

Funktorok

- A legutóbbi megoldásnak szép interfésze van, de még mindig újra kell implementálni minden különböző kulcstípusra és rendezésre.
- A megoldás a *funktor* nyelvi elem, ami generikus programkód írását teszi lehetővé
- A funktor lényegében egy struktúrákkal paraméterezett struktúra, amely a példányosítás során egy "sima" struktúrát hoz létre. Tekinthető struktúrák közötti függvénynek is.

Példa funktorra

- A következő funktorral ugyanazt a struktúrát lehet létrehozni, ami a legutóbbi példában volt:

```
functor MapFun(structure K: ORD) :>
  MAP where type Key.ty = K.ty = struct
  structure Key = K
  ...
end
structure MapStr = MapFun(OrdStr)
```

- A nagy különbség az, hogy az `OrdStr` struktúra lecserélésekor nem kell lemásolni a kódot

Basis library

- Az SML szabványos könyvtár tartalma:
  - Alaptípusok műveletei: `int`, `word`, `char`, `string`, `bool`, `real`, `option`, `list`
  - Tárolók: array, vector
  - I/O könyvtár
  - Dátum, idő lekérdezése
  - Operációs rendszer elérése
- http://www.standardml.org/Basis

Alaptípusok struktúrái

- `Bool :> BOOL`
- `Int :> INTEGER, LargeInt`
- `Word :> WORD, Word8, LargeWord`
- `Real :> REAL, LargeReal`
- `Char :> CHAR, WideChar`
- `String :> STRING, WideString`
- `Option :> OPTION`
- `List :> LIST`

Vector típus

- Polimorf vektor: `Vector :> VECTOR`
- A `VECTOR` szignatúra néhány eleme:

```
eqtype 'a vector
fromList : 'a list -> 'a vector
length : 'a vector -> int
sub : 'a vector * int -> 'a
update : 'a vector*int*'a -> 'a vector
```

- Monomorf vektor: csak adott típusú elemeket tárol, pl. `CharVector` vagy `IntVector`, a szignatúrájuk: `MONO_VECTOR`

Array típus

- Változtatható tartalmú tömb: `Array:>ARRAY`
- Az `ARRAY` szignatúra néhány eleme:

```
eqtype 'a array = 'a array
val array : int * 'a -> 'a array
val fromList : 'a list -> 'a array
val length : 'a array -> int
val sub : 'a array * int -> 'a
val update : 'a array*int*'a -> unit
val vector : 'a array -> 'a vector
```

- Monomorf eset: `CharArray:>MONO_ARRAY`

I/O lehetőségek

- Három szintű I/O könyvtár:
  - Primitív I/O: pufferelés nélküli írás/olvasás
  - Stream I/O: pufferek használata, funkcionális olvasás, imperatív írás
  - Imperatív I/O: imperatív, átirányítható streamek
- Konkrét, használható struktúrák: `TextIO` és `BinIO`, ezeken keresztül lehet fájlokat nyitni és elérni a különböző szintű interfészeket
- A `TextIO`-ban van az `stdIn` és az `stdOut`

A STREAM_IO szignatúra

- Típusok: `instream`, `outstream`, `elem`, `vector`
- Input műveletek:
  - `input1:instream->(elem*instream)option`
    Egy elemet olvas be, NONE: stream vége
  - `inputAll:instream -> vector*instream`
    Minden elérhető elemet beolvas, üres vektor: vége
  - `canInput:instream*int -> int option`
    Megnézi, ven-e elég adat. NONE: blokkolni fog az input
  - `closeIn:instream -> unit`
    Lezárja a streamet

STREAM_IO (folyt.)

- Output műveletek:
  - `output: outstream*vector -> unit`
    Kiírja a vektor tartalmát
  - `output1: outstream*elem -> unit`
    Kiír egy elemet
  - `flushOut: outstream -> unit`
    Kiüríti a puffert
  - `closeOut: outstream -> unit`
    Lezárja a streamet

Az IMPERATIVE_IO szignatúra

- Tartalmaz egy `StreamIO` struktúrát, így lehet elérni a funkcionális IO műveleteit
- Típusok: `instream`, `outstream`, `elem`, `vector`
- Az output műveletek ugyanazok, mint az előző esetben, a különbség csak az átirányíthatóságban van
- Az input műveletek kicsit változnak:
  `input1: instream -> elem option`
  `inputAll: instream -> vector`

Átirányítás

- Input átirányító műveletek:
  - `mkInstream:`
    `StreamIO.instream -> instream`
    Létrehoz egy imperatív streamet egy funkcionálisból
  - `getInstream:`
    `instream -> StreamIO.instream`
    Lekérdezi a stream mögött álló funkcionális streamet
  - `setInstream:`
    `instream*StreamIO.instream -> unit`
    Átállítja a stream mögött álló funkcionális streamet

A TextIO struktúra

- Szöveges módú streamek létrehozása:
  - `openIn: string -> instream`
  - `openOut: string -> outstream`
  - `openAppend: string -> outstream`
  - `openString: string -> instream`
- Előre definiált streamek: `stdIn, stdOut, stdErr`
- Sort beolvasó művelet:
  `inputLine: instream -> string option`

String konverziók

- A StringCvt struktúra tartalmazza a szükséges típusokat és néhány függvényt
  - `datatype radix = BIN|OCT|HEX|DEC`
  - `datatype realfmt = SCI of int option | FIX of ... | GEN of ... | EXACT`
  - `type('a,'b)reader='b->('a,'b)option` (a `STREAM_IO`-beli `input1` függvény ilyen)
  - `padLeft: char->int->string->string`
  - `skipWS: (char,'a)reader -> 'a -> 'a`

Alaptípusok felismerése

- Az alaptípusokhoz van scanner függvény:
  - `Int.scan: radix -> (char, 'a)reader -> (int,'a)reader`
  - `Real.scan: (char,'a)reader -> (real,'a)reader`
- A scanner függvények alkalmazása stringre:
  - `StringCvt.scanString: ((char,cs) reader -> ('a,cs) reader) -> string -> 'a option`
  - Pl.: `scanString (Int.scan DEC) "123"`

Konverzió streamekről

- Közvetlenül a scanner fügvénnyel:

```
Int.scan StringCvt.DEC
  TextIO.StreamIO.input1
  (TextIO.getInstream stdIn)
```

- Imperatív streamek scanner függvénye:
  - `TextIO.scanStream: ( (char, TextIO.instream) reader ->('a, TextIO.instream) reader ) -> TextIO.instream -> 'a option`
  - Pl.: `scanStream (Int.scan DEC) stdIn`

Alaptípusok megjelenítése

- Adatformázó függvények:
  - `Int.fmt: StringCvt.radix-> int -> string`
  - `Real.fmt: StringCvt.realfmt -> real -> string`
- String konverziók rövidítése:
  - `Int.toString: int -> string`
  - `Int.fromString: string -> int option`