

KIÍRÁS AZ SML-BEN



Kiírás

- `{TextIO.}print : string -> unit`

`print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.

- `{General.}makestring : numtxt-> string` (Csak mosml!)

`makestring v` = eredménye a `v` érték *ábrázolása*.

- `{Meta.}printVal : 'a -> 'a` (Csak mosml! Az Alice-ben: `'a -> unit`)

`printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. *Csak interaktív módban használható.*

1. megjegyzés. A `{` és `}` kapcsos zárójelek között opcionális modulnév áll. Pl. `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.

2. megjegyzés. `numtxt = int | real | word | word8 | char | string`

- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

`Bool.toString : bool -> string`

`String.toString : string -> string`

`Char.toString : char -> string`

`Time.toString : time -> string`

`Date.toString : date -> string`

`Word.toString : word -> string`

`Int.toString : int -> string`

`Word8.toString : word8 -> string`

`Real.toString : real -> string`

Kíírás (folyt.)

● Példák:

```
print("alma"^"Korte\n");
```

```
almaKorte
```

```
> val it = () : unit
```

```
makestring ~5.8e~3;
```

```
> val it = "~0.0058" : string
```

```
printVal("alma"^"Korte\n");
```

```
"almaKorte\n"> val it = "almaKorte\n" : string
```

```
makestring("alma"^"Korte\n");
```

```
> val it = "\"almaKorte\\n\"" : string
```

● printVal-lal tetszőleges típusú érték íratható ki, például ennes, rekord vagy lista:

```
printVal (3, 5.0);
```

```
(3, 5.0)> val it = (3, 5.0) : int * real
```

```
printVal {m2 = 3, m1 = 5.0};
```

```
{m1 = 5.0, m2 = 3}> val it = {m1 = 5.0, m2 = 3} : {m1 : real, m2 : int}
```

```
printVal [#"A",#"Z",#":"];
```

```
["#A", #"Z", #":"]> val it = ["#A", #"Z", #":"] : char list
```

Kiírás (folyt.)

- `printVal`-al `datatype`-deklarációval létrehozott típusú érték is kiíratható, pl.

```
datatype t = L | B of t * t;
```

```
> New type names: =t
```

```
datatype t = (t, con B : t * t -> t, con L : t)
```

```
con B = fn : t * t -> t
```

```
con L = L : t
```

```
val fa = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)));
```

```
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```

```
printVal fa;
```

```
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))> val it = B(...
```

- A kiírt sor túl hosszú, a folytatását `...`-tal helyettesítettük.
- Törjük el a sort a `> válaszjel` előtt *szekvenciális* kifejezés alkalmazásával:

```
(printVal fa; print "\n");
```

```
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
```

```
> val it = () : unit
```

- Sajnos, az eredményül kapott érték nem `fa` értéke!

Kiírás (folyt.)

- Hogyan írjunk ki egy újsor-jelet úgy, hogy az eredmény mégis fa értéke legyen? Pl. így:

```
let val res = printVal fa; val _ = print "\n"
in
  res
end;
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```

- Ez elég körülményes. A before operátort az ilyen esetek kezelésére vezették be:

```
printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```

- Szekvenciális kifejezés alkalmazásával további magyarázó szöveget írhatunk ki:

```
(print "`fa' értéke =\n"; printVal fa before print "\n");
`fa' értéke =
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```

Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az `mosml`-értelmező is) alapesetben csak az első 20 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszát a `printLength`, a szintek számát a `printDepth` *frissíthető változó* szabályozza. Mindkét érték felülírható.

```
printLength : int ref      printLength := 7; !printLength;
printDepth  : int ref      printDepth  := 3; !printDepth;
```

- Példák:

```
printLength := 6; printVal [1,2,3,4,5,6,7,8] before print "\n";
[1, 2, 3, 4, 5, 6, ...]
> val it = [1, 2, 3, 4, 5, 6, ...] : int list
```

```
printDepth := 4; printVal fa before print "\n";
B(B(#, #), B(#, #))
> val it = B(B(#, #), B(#, #)) : t
```

- Figyelem: a `printLength` és a `!printLength` kifejezések különböznek!

<code>printLength;</code>	<code>!printLength;</code>
<code>> val it = ref 7 : int ref</code>	<code>> val it = 7 : int</code>

NYOMKÖVETÉS KIÍRÁSSAL: LISTÁK

Nyomkövetés kiírással: length (nem iteratív)

- Az mosml-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
- Példa: a length függvény két változatának kiértékelése
- A length „naív” változata

```
fun length (_::xs) = 1 + length xs
  | length []      = 0
```

- A length „naív” változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun length ((_ : int) :: xs) =
    printVal(1 + (print " & "; printVal(length(printVal xs))
              before print " $ "
            )
          )
    before print " #\n"
  | length []      = (print " * "; printVal 0
                    before print " %\n")
```


Nyomkövetés kiírással: lengthi (iteratív)

- A length iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
                    | len (i, [])      = i
                  in len(0, xs)
                  end
```

- A length iteratív változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun lengthi xs =
  let fun len (i, (_ : int) :: xs) =
    len((print " "; printVal((printVal i
      before print " $ ") + 1)),
    (print " & "; printVal xs)
  )
  before print "#\n"
  | len (i, []) = (print " * "; printVal i
    before print " %\n")
in len(0, xs)
end
```

Nyomkövetés kiírással: length egy alkalmazása

- length egy alkalmazása

```

fun length ((_ : int) :: xs) =
    printVal(1 + (print " & "; printVal(length(printVal xs))
                before print " $ "
                )
            )
    before print " #\n"
| length [] = (print " * "; printVal 0
               before print " %\n");

```

```

length [1,2,3];
& [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #

```

Nyomkövetés kiírással: lengthi egy alkalmazása

- lengthi egy alkalmazása

```

fun lengthi xs =
  let fun len (i, (_ : int) :: xs) =
    len((print " "; printVal((printVal i
      before print " $ ") + 1)),
      (print " & "; printVal xs)
    )
    before print "#\n"
  | len (i, []) = (print " * "; printVal i
    before print " %\n")
  in len(0, xs)
end

```

```

lengthi [1,2,3];
0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#

```

Nyomkövetés kiírással: length és lengthi összehasonlítása

- length és lengthi kiértékelésének összehasonlítása

length [1,2,3];	lengthi [1,2,3];
& [2, 3] & [3] & [] * 0 %	0 \$ 1 & [2, 3] 1 \$ 2 & [3] 2 \$ 3 & [] * 3 %
0 \$ 1 #	#
1 \$ 2 #	#
2 \$ 3 #	#

HIBAKERESÉS ÉS NYOMKÖVETÉS POLY/ML-BEN

Hibakeresés és nyomkövetés Poly/ML-ben

- Töréspontot elhelyezni csak olyan segédfüggvényben lehet, amelyet egy másik függvény törzsében egy `let`-kifejezésben definálunk (példákat a következő diákon mutatunk).
- A hibakeresést és nyomkövetést *először* a `PolyML.Compiler.debug` kapcsoló `true` értékre állításával engedélyezni kell (ld. a következő diát), *majd* definiálni kell azokat a függvényeket, amelyek működését követni akarjuk, vagy amelyekben töréspontot akarunk elhelyezni. (Ellenkező esetben a hibakereséshez és nyomkövetéshez szükséges plusz kódot a Poly/ML értelmező nem írja bele a lefordított programrészbe.)
- A politípusúnak definiált nevek értékét nem tudják kiírni a hibakeresés kiíró függvényei (`PolyML.Debug.variables()`, `PolyML.Debug.dump()`, `PolyML.Debug.stack()` – lásd a következő diákon). Ahhoz, hogy egy név értékét ezek a függvények kiírják, a névnek *már a függvény definiálásakor* monotípusúnak kell lennie. (Egy nevet, ha a szöveggörnyezetből nem vezethető le a típusa, *típusmegkötéssel* tehetünk monotípusúvá.)
- A Poly/ML fontosabb hibakereső függvényeit lásd a következő dián (konkretizálva a `length` és a `len` függvényre utaló hivatkozásokkal).
- A `PolyML.profiling : int -> unit` függvénnyel egy kifejezés kiértékelési idejét, ill. egyes függvények futási idejét és helyfoglalását monitorozhatjuk (részletek a Poly/ML-leírásban olvashatók).

Hibakeresés és nyomkövetés Poly/ML-ben (folyt.)

```
PolyML.Compiler.debug := true;
open PolyML.Debug;
```

Hibakeresés engedélyezése; engedélyezett állapotban kell definiálni a vizsgálandó függvényt

```
breakIn "len";
breakIn "length()len";
continue();
down();
up();
dump();
stack();
variables();
clearIn "length()len";
```

Töréspont elhelyezése, rövid változat

Töréspont elhelyezése, teljes változat

Folytatás a töréspont következő előfordulásáig

Áttérés az előző hívási szintre a veremben

Áttérés a következő hívási szintre a veremben

A verem teljes tartalmának kiírása

A hívások kiírása a veremtartalom alapján

A nevek értékének kiírása

Töréspont törlése, teljes változat

```
trace true;
step();
stepOver();
stepOut();
```

Nyomkövetés bekapcsolása

Adott hívási szinten tovább vagy beljebb

Adott hívási szinten tovább

Előző hívási szintre vissza

`breakIn` és `clearIn` konkrétan a `length` és a `len` függvényre hivatkozik a fenti felsorolásban.

Használatukról részletesebben itt lehet olvasni: <http://www.polymml.org/docs/Debugging.html>.

Első példa a Poly/ML debugger használatára

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
  let (* len : 'a list -> int
       len xs = az xs elemeinek száma *)
      fun len [] = 0
        | len (_ :: xs) = 1 + len xs
    in
      len zs
    end;
```

```
breakIn "length()len";
val it = () : unit

length(explode "abcde");
line:7 function:length()len
debug>
```

```
variables();
val xs = [?, ?, ?, ?] val zs = [?, ?, ?, ?, ...]
val it = () : unit
debug>
```

- Hmm. Politípusúnak definiált értéket a Poly/ML hibakeresője nem tud kiírni?

Első példa a Poly/ML debugger használatára (folyt.)

- Nem bizony! A nevek típusát pl. *típusmegkötéssel* meg kell adni ahhoz, hogy az értékek kiírásához szükséges kód fordítási időben beépüljön a lefordított programba.

```
(* length : char list -> int *)
fun length (zs : char list) =
  let (* len : char list -> int *)
      fun len []          = 0
        | len (_ :: xs : char list) = 1 + len xs
    in
      len zs
    end;

breakIn "len";
length(explode "abcde");
variables();
...
val xs = [#"b", #"c", #"d", #"e"] val zs = [#"a", #"b", #"c", #"d", ...]
...

continue(); variables();
...
line:5 function:length()len
debug> val xs = [#"c", #"d", #"e"] val zs = [#"a", #"b", #"c", #"d", ...]
...
```

Első példa a Poly/ML debugger használatára (folyt.)

- length egy hibás változata a hibakeresés kipróbálásához

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
  let (* len : 'a list * int -> int
       len xs = n + az xs elemeinek száma -- HIBÁS! *)
      fun len ([], n)      = n
        | len (_ :: xs, n) = len(xs, n)
    in len(zs,0)
  end;
```

- length egy másik hibás változata a hibakeresés kipróbálásához

```
(* length : char list -> int
   length zs = a zs elemeinek száma *)
fun length (zs : char list) =
  let (* len : char list * int -> int
       len xs = n + az xs elemeinek száma -- HIBÁS! *)
      fun len ([], n)      = n
        | len (_ :: xs : char list, n : int) = len(xs, n)
    in len(zs,0)
  end;
```

Második példa a Poly/ML debugger használatára

```
(* maxl : (string * string -> string) -> string list -> string
   maxl max zs = a zs lista max szerint legnagyobb eleme *)
fun maxl max zs = let fun mxl [] = NONE
                      | mxl [n] = SOME n
                      | mxl (n::m::ns) = mxl(max(n:string, m)::ns)
                    in
                      mxl zs
                    end;
```

```
(* 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;
```

● Hibakeresés töréspont elhelyezésével

```
breakIn "mxl";
maxl stringMax ["ec", "pec", "kimehetsz", "holnaputan", "bejohetsz"];
line:1 function:
debug>

continue();
val it = () : unit
line:6 function:maxl()mxl
debug>
```

Második példa a Poly/ML debugger használatára (folyt.)

```
step();  
val it = () : unit  
  line:4 function:stringMax  
debug>
```

```
step();  
val it = () : unit  
  line:6 function:max1()mx1  
debug>
```

```
step();  
val it = () : unit  
  line:4 function:stringMax  
debug>
```

```
variables();  
val t = "kimehetsz" val s = "pec"  
val it = () : unit  
debug>
```

```
step();  
val it = () : unit  
  line:6 function:max1()mx1  
debug>
```

Második példa a Poly/ML debugger használatára (folyt.)

```
stepOver();
val it = () : unit
  line:6 function:maxl()mxl
debug>

variables();
val n = "pec" val m = "bejohetsz" val ns = []
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
...

continue();
val it = () : unit
  line:5 function:maxl()mxl
debug>

stack();
  line:5 function:maxl()mxl
  line:6 function:maxl()mxl
  line:6 function:maxl()mxl
  line:6 function:maxl()mxl
  line:6 function:maxl()mxl
  line:8 function:maxl
...
```

Második példa a Poly/ML debugger használatára (folyt.)

```

variables();
val n = "pec" val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...]
val max = fn
...

dump();
Function maxl()mxl: val n = "pec" val zs =
  ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "bejohetsz" val ns = []
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "holnaputan" val ns = ["bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "kimehetsz"
val ns = ["holnaputan", "bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "ec" val m = "pec"
val ns = ["kimehetsz", "holnaputan", "bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl: val mxl = fn
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn

val it = () : unit
debug>

```

Második példa a Poly/ML debugger használatára (folyt.)

● Nyomkövetés

```
clearIn "mx1";  
trace true;  
mx1 stringMax ["ec", "pec", "kimehetsz", "holnaputan", "bejohetsz"];  
mx1 entered  
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...]  
val max = fn
```

```
mx1()mx1 entered  
val n = "ec"  
val m = "pec"  
val ns = ["kimehetsz", "holnaputan", "bejohetsz"]
```

```
stringMax entered val t = "pec" val s = "ec"  
stringMax returned "pec"
```

```
mx1()mx1 entered  
val n = "pec"  
val m = "kimehetsz"  
val ns = ["holnaputan", "bejohetsz"]
```

Második példa a Poly/ML debugger használatára (folyt.)

```
stringMax entered val t = "kimehetsz" val s = "pec"
stringMax returned "pec"
```

```
maxl()mxl entered
val n = "pec"
val m = "holnaputan"
val ns = ["bejohetsz"]
```

```
stringMax entered val t = "holnaputan" val s = "pec"
stringMax returned "pec"
maxl()mxl entered val n = "pec" val m = "bejohetsz" val ns = []
  stringMax entered val t = "bejohetsz" val s = "pec"
  stringMax returned "pec"
  maxl()mxl entered val n = "pec"
  maxl()mxl returned SOME "pec"
  maxl()mxl returned SOME "pec"
  maxl()mxl returned SOME "pec"
  maxl()mxl returned SOME "pec"
  maxl()mxl returned SOME "pec"
  maxl returned SOME "pec"
val it = SOME "pec" : string option
```