

Az SML-programozás alapjai

Az SML-programozás alapjait példákon mutatjuk be.

Minden függvényt *fejkomment* specifikáljon:

- az eredmény *deklaratív* leírása az argumentum(ok) felhasználásával
- a függvény *típusának* specifikálása

A megjegyzések (* és *) párok közé kell tenni

- a megjegyzések egymásba skatulyázhatók

Két szám maximuma

- az aritmetikai műveletek és a relációk argumentumának típusa alapértelmezés szerint: int
- az alapértelmezés ún. *típusmegkötéssel* ($\text{(: } \langle \text{típusnév} \rangle \text{ megvaltoztatható, pl. rmax (a, b : real), rmax (a, b) : real}$)
- az SML *szigorúan típusos* nyelv, ezért az SML-értelmező az értékek típusát általában *típus/vezetéssel* meg tudja határozni
- az alábbi két deklaratív specifikáció közül a második precízebb, és azt is kifejezi, hogy a = b esetén megengedhető a nondeterminizmus
- a *nondeterminizmus* (megfelelő fordítóprogram esetén) javíthatja a hatékonysságot (a fordítóprogram a kedvezőbb változatot választhatja)

- (* 1. $\text{imax(a, b) = a és b maximuma}$
 2. $(\text{imax(a, b)= a } \wedge \text{ a}>=b) \vee /$
 $(\text{imax(a, b)}=\text{b } \wedge \text{ a}<=b)$
- imax : int * int -> int
- *)

```
fun imax (a, b) = if a >= b then a else b
```

Két szám maximum (folyt.)

Ugynéz, valós számokra

- típusmegkötést kell alkalmazni

(* rmax (a, b) = a és b maximuma

rmax : real * real -> real

```
*) fun rmax (a, b : real) =
    if a >= b then a else b

  • egyes formális logikai specifikációk nem írhatók fel
  közvetlenül SML-nyelven

  • az alábbi függvénydefiníció rossz, ui. rmax
    függvényérték:
    (* (max(a, b)=a /\ a>=b) \vee (max(a, b)=b
    /\ a<=b)
    max : real * real -> real
  *)
  fun rmax (a : real, b) =
    rmax=a andalso a>=b orelse
    rmax=b andalso a<=b
```

Két szám maximuma (folyt.)

- hasonló specifikációjú függvény három argumentummal írható, de csak az argumentumok között fennálló kapcsolatellenőrzésére

(* (max(a, b, r) = igaz, ha

(r=a /\ a>=b) \vee (r=b /\ a<=b)

max : real * real -> real

```
*)
fun rmax (a : real, b, r) =
  r=a andalso a>=b orelse
  r=b andalso a<=b
```

- először a logikai definíció nem írható fel

- az alábbi függvénydefiníció rossz, ui. rmax

függvényérték:

(* (max(a, b)=a /\ a>=b) \vee (max(a, b)=b

/\ a<=b)

max : real * real -> real

*)

fun rmax (a : real, b) =

rmax=a andalso a>=b orelse

rmax=b andalso a<=b

Logikai műveletek

- andalso és orelse / lista kiértékelésű infix operátorok
 - not prefix pozíciójú monadikus függvényel $\&\&$, ill. $\|$ néven infix pozícióban használható operátorokat írunk
 - az infix-deklaráció alakja: infix $<prec>$ $<név>$
 - az így deklarált, $<prec>$ precedenciájú $<név>$ operátor *balra* köt
 - $\&\&$ és $\|$ mindenkiértékelésű, mert az SML-programozó csak ilyen függvényt definiálhat
 - ($* \&\& (a, b) = a \wedge b$
 $\&\& : \text{bool} * \text{bool} \rightarrow \text{bool}$
 $*$)
 $\text{fun } \&\& (a, b) = \text{if } a \text{ then } b \text{ else false}$
 $\text{infix } 3 \&\&$
- fun $\&\& (a, b) = \text{if } a \text{ then true else b}$
 $\text{infix } 2 \|$

- a függvényeket mintaillesztéssel is definiálhatjuk
 - egy vagy több esetet lefedő változatokat (klózokat) kell írni
 - minden lehetséges esetet le kell fedni
 - a lefedetlen esetekre az SML-értelmező figyelmeztet
 - a változatok (klózok) kiértékelése felülről lefelé, balról jobbra halad
 - diszjunktív változatok esetén a sorrend elvileg közömbös
 - pl. az $\&\&$ operátort így definiálhatjuk mintaillesztéssel
 - fun $\&\& (\text{true}, \text{b}) = \text{b}$
 - $| \&\& (\text{false}, _) = \text{false}$
 - ha $\&\&$ már definálva volt, a fenti definíció szintaktikailag hibás, ui. az SML-értelmező az $\&\&$ operátorjelét infix pozícióban várja
 - az op függvény infix pozíójú nevet átmeneti legprefixszé alakítja
 - fun $\text{op}\&\& (\text{true}, \text{b}) = \text{b}$
 - $| \text{op}\&\& (\text{false}, _) = \text{false}$

Függvény definíálása mintaillesztéssel (folyt.)

- nem kapunk hibajelzést, ha az infix operátorjel a definícióban is infix pozícióban alkalmazzuk
- ```
infix 3 &&
fun true && b = b
| false && _ = false
|| definíálása mintaillesztéssel
(* || (a, b) = a \vee b
| | : bool * bool -> bool
*)
```

• λ-jelöléssel minden /ustá kiértékelésű függvényt definiálunk

```
(* a andalso b*)
(fn true => b | false => false) a
(* a orelse b *)
(fn true => true | false => b) a
(* if b then e1 else e2 *)
(fn true => e1 | false => e2) b
```

- példák a λ-jelölésű függvények alkalmazására

```
(fn true => 3.0 > 5.0
| false => false) (3 < 5)
(fn true => true
| false => #("A" < #"Z") (3.0 < 5.0)
(fn true => "alma" | false => "szilva")
((3, #"A") = (4-1, Char.toUpper
#"a")))
```

## Függvény definíálása mintaillesztéssel (folyt.)

- λ-jelöléssel minden /ustá kiértékelésű függvényt definiálunk

```
(* a andalso b*)
(fn true => b | false => false) a
(* a orelse b *)
(fn true => true | false => b) a
(* if b then e1 else e2 *)
(fn true => e1 | false => e2) b
```

- példák a λ-jelölésű függvények alkalmazására

```
(fn true => 3.0 > 5.0
| false => false) (3 < 5)
(fn true => true
| false => #("A" < #"Z") (3.0 < 5.0)
(fn true => "alma" | false => "szilva")
((3, #"A") = (4-1, Char.toUpper
#"a")))
```

## Iteráció megvalósítása rekurzióval

- ciklusváltozó megvalósítása SML-ben
    - funkcionális stílusban nem használható frissíthető változó
    - argumentum használható ciklusváltozóként
  - példa: „jó számok”: keressük azokat a számokat, amelyek négyzete háromjegyű, és a szám fordítottjával kezdődik (vö. Prolog-példa)
    - (\* jószam ?? = olyan szám, melynek a négyzete háromjegyű, és a szám fordítottjával kezdődik
    - joszam : ???? -> int
    - \*)
  - mi legyen a jószam függvény argumentuma?
    - a () (nullas), ami unit típusú (C-beli megfelelője void)

- `joszam1` int helyett `int * bool` típusú értéket ad eredményül
  - a pár 2. tagja a keresés sikérével vagy sikertelenségét jelzi
  - a `jsz` segédfüggvény közli, hogy az átadott szám jó-e
  - az `infix` \* típusoperátor precedenciája nagyobb, mint az ugyancsak `infix` -> típusoperátor
  - a \* típusoperátor *balra*, a -> típusoperátor *jobbra* köt
  - a *postfix* típusoperátorok (pl. a `list`) precedenciája nagyobb, mint az *infix* típusoperátoroké
  - (\* `jsz (a, b) = pár, melynek 1. tagja  $10*a+b$ , 2. tagja pedig akkor igaz, ha az 1. tag jó szám`
  - `jsz : int * int -> int * bool`
  - \*)  

```
fun jsz (a, b) =
 if ($10*a+b$) * ($10*a+b$) div 10 = $10*b+a$
 then ($10*a+b$, true)
 else ($10*a+b$, false)
```

  - *lokális kifejezéseket csak egyszer számítsuk ki részkifejezésekkel* használható arra, hogy az ismétlődő
  - a lokális kifejezés általános alakja: let D in E end

## joszam 1. változata (folyt.)

```

fun jsz (a, b) =
 let val sz = 10*a+b
 in if sz*sz div 10 = 10*b+a
 then (sz, true)
 else (sz, false)
 end
 • joszam10 néven segédfüggvényt írunk, rekurzív
 hívásakor az argumentumát növeljük
 • amíg a segédfüggvény sikertelen eredményt ad,
 ismételten hívogatjuk
 (* joszam10 (a, b) = pár, melynek 1.
 tagja 10*a+b, 2. tagja pedig akkor
 igaz,
 ha az 1. tag jó szám
 joszam1 : unit -> int * bool
 *)
 fun joszam1 () =
 let
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
 fun jsz (a, b) = ...
..
 *)
 fun joszam10 (10, 0) = (0, false)
 | joszam10 (a, 10) = joszam10(a+1, 0)
 | joszam10 (a, b) =
 let val (v, i) = jsz(a, b)
 in if i then (v, i)
 else joszam10(a, b+1)
 end
 • példa joszam1 alkalmazására
 - joszam1();
 > val it = (27, true) : int * bool

```

## joszam 2. változata

- joszam2 int helyett int option típusú értéket ad eredményül
- egy 'a option típusú érték vagy NONE, vagy SOME v lehet, ahol v 'a típusú

(\* joszam2 () = vagy NONE, vagy SOME v,  
ahol v olyan szám, melynek a négyzete  
háromjegyű, és a szám fordítottjával

kezdődik

joszam2 : unit -> int option

\*)

fun joszam2 () =

let fun jsz (a, b) =

let val sz = 10\*a+b

in

if sz\*sz div 10 = 10\*b+a  
then SOME sz else NONE

end

## joszam 2. változata (folyt.)

```
fun joszam20 (10, 0) = NONE
| joszam20 (a, 10) =
 joszam20 (a+1, 0)
| joszam20 (a, b) =
 case jsz(a, b) of
 NONE => joszam20(a, b+1)
 | SOME v => SOME v
 in
 joszam20(1, 0)
 end
```

Példa joszam2 alkalmazására

```
- joszam2();
> val it = SOME 27 : int option
```

### joszam 3. változata

- joszam3 int típusú értéket ad eredményül, vagy hibát (exception) jelez, ha nem talál jó számon

```
exception Joszam
```

```
(* > exn Joszam = Joszam : exn *)
```

```
(* joszam3 () = olyan szám, melynek a
négyzete háromjegyű, és a szám
fordított jával kezdődik
joszam3 : unit -> int
*)
```

```
fun joszam3 () =
```

```
let fun jsz (a, b) =
```

```
let val sz = 10*a+b
```

```
in
```

```
if sz*sz div 10 = 10*b+a
```

```
then sz else raise Joszam
```

```
end
```