

Bevezetés

SML = Standard Meta Language

deklaratív programozás = logikai, ill. funkcionális (applikatív) programozás

funkcionális programozás = programozás *függvények alkalmazásával*

applikatív programozás = programozás *függvények alkalmazásával*

- imperatív nyelvek:
- *parancs* (utasítás) *végrehajtásán* alapulnak
 - „gépközei” programozás
- deklaratív (logikai és funkcionális) nyelvek:
 - *kifejezés kiértékelésén* alapulnak
 - logikai: minden kifejezés *reláció* (tkp. bool típusú)
 - funkcionális: minden kifejezés *tetszőleges*, de *meghatározott típusú*
 - „magasszintű” programozás

Rövid történeti áttekintés

- Leonhard Euler (1748): $\sin .x$ később $\sin x$ vagy $\sin(x)$
- Alfred N. Whitehead, Bertrand Russel (1910) ...
Alonzo Church: *lambda-jelölés*: $\lambda x.x + x$
- Church, 1936: a (funkcionális) lambda-kalkulus és a(z imperatív) Turing-gép ekvivalensek \rightarrow a funkcionális és az imperatív programozás *ekvivalensek*
- Church-tétele - a kiszámítható függvények halmaza azonos a rekurzív függvények halmazával - a funkcionális programozás alapja
- 1945-től beszélhetünk gyakorlati programozásról: gépi kód, autokód, assembly nyelv (mind imperatív)

- 1956: FORTRAN (FORmula TRANslation), nagy lépés előre, néhány nagyhatású, ám részben *hibásnak bizonyult* koncepcióval
 - *szubrutin* (célja a modularizálás): olyan pszeudofüggvény, amely (1) módosíthatja az argumentumait, (2) lehetnek implicit argumentumai (= a globális változók)
 - * CALL FACTORIAL(I), vagy akár CALL FACTORIAL
 - * hatás, mellékhatás, állapotváltozás
 - * globális változókkal a rekurzió nem lehetséges
 - * a szubrutin a FORTRAN-ban nem lehet rekurzív
 - a rekurzió „pótlására” bevezették az *iterációt* (ciklust)
 - bevezették az értékadást $J = J + 1$ alakban
 - a szintaxist nem elég körültekintően tervezték meg, pl.


```
* DO 10 I = 1.5
  A(I) = X + B(I)
10 CONTINUE
```
 - * az első sor valójában $DO 10 I = 1.5$ és nem $DO 10 I = 1,5$

- 1959: COBOL (COmmon Business Oriented Language): kísérlet a természetes nyelvhez (az angolhoz) közel álló nyelven való programozásra
- 1960: ALGOL (ALGOrithmic Language)
 - eljárás és függvényeljárás
 - rekurzió
- 1960: LISP (LISt Processing language)
 - alapja a λ -kalkulus
 - eredeti célja: *szimbolikus differenciálás*
 - ma elterjedt változata a Common LISP, imperatív elemekkel
 - az emacs is egy LISP-dialektusban íródott (Emacs-LISP)
- 1962-től újabb funkcionális programozási nyelvek:
 - APL, ML, HOPE, ERLANG, Miranda,
 - SML, Haskell, gofer, clean
 - stb.

A Meta Language (ML) története és jelene

- ML: Edinborough, 1977
 - tételbizonyításra (kijelentések helyességének levezetésére)
- Definition of Standard ML, 1990
 - Core Language + Module Language
- Revised Definition of Standard ML, 1997
 - + SML Basis Library
- Értelmezők:
 - mosml (Moscow SML, RVAU/Koppenhága és RAS/Moszkva)
 - sml-nj (Lucent Technologies, Bell Lab., New Jersey, USA)
 - Caml Light, Objective Caml (INRIA, Franciaország)
 - professzionális megvalósítások: PoplogML, MLWorks

Kifejezések az SML-ben

Példák helyes kifejezésre SML-ben:

1. $3 * 4 \text{ div } ((6-3) \bmod 2)$
2. `if 3<6 then "alma" else "szilva"`
3. `"Ali baba és a " ^ Int.toString 40 ^ " rabló"`
4. `3.0 * Math.pi + Math.sin 0.57`
5. `(1, "hello", #"A", 3.14159, Math.e, List.filter, (), [])`
6. `1 :: [3,7,15,31,63]`

A kifejezéseket az SML-értelmező *kiértékeli*, és a lehető legegyszerűbb alakra hozza (azaz *egyszerűsíti*). Ez az ún. *kanonikus alak*.

Minden kifejezésnek van *értéke*. Minden értéknek van *típusa*.

Mi az *értéke* és a *típusa* a felsorolt kifejezéseknek?

1. - `3 * 4 div ((6-3) mod 2)`
`> val it = 12 : int`
2. - `if 3<6 then "alma" else "szilva"`
`> val it = "alma" : string`
3. - `"Ali baba és a " ~`
`Int.toString 40 ~ " rabló"`
`> val it =`
`"Ali baba és a 40 rabló" : string`
4. - `3.0 * Math.pi + Math.sin 0.57`
`> val it = 9.9644100095 : real`
5. - `(1, "hello", #"A", 3.14159, Math.e,`
`List.filter, (), [])`
`> val it =`
`(1, "hello", #"A", 3.14159,`
`2.71828182846,`
`fn, (), []) :`
`int * string * char * real * real *`
`(('a -> bool) -> 'a list -> 'a list)`
`*`
6. - `1 :: [3,7,15,31,63]`
`> val it = [1, 3, 7, 15, 31, 63] : int`
`list`

Kifejezések alkotóelemei

- állandó (nulla argumentumú függvényjel)
- operátor (műveleti jel, függvényjel): infix, prefix
- zárójel
- változó

A *név* (azonosító) jelölhet

- operandust (állandót, változót)
- operátort

A név lehet

- alfanumerikus név,
 - pl. `sin`, `Math`, `rezsoke`, `sm197`
- írásjelekből álló („szimbolikus”) név,
 - pl. `~`, `-`, `+`, `!!!`, `\`, `/`, `\`, `+++`

Típuskifejezések alkotóelemei

- típusállandó,
 - pl. `int`, `real`, `bool`, `char`, `string`
- típusoperátor (`infix`, `postfix`),
 - pl. `*`, `->`, `list`
- zárójel
- típusváltózo,
 - pl. `'a'`, `'b'`, `'gamma'`, `''d`

Szintaxis

Többféle szintaxis egy programozási nyelven belül (ld. például MOSML Language Overview [LO]):

- *kifejezések szintaxisa* (ld. LO, 4. o.)
- *típuskifejezések szintaxisa* (ld. LO, 5. o.)
- *mintaillesztés szintaxisa* (ld. LO, 6. o.)
- *deklarációk szintaxisa* (ld. LO, 5. o.)

Szabad és kötött változó, értékdeklaráció

- szabad változó: csak olyan, amely egy korábbi *deklarációban* már kapott értéket
 - `val pi = 3.14159`
 - `val r = 1.0`
 - `val area = r * r * pi`
- kötött változó: függvény argumentuma (formális paramétere)
 - `fn r => r * r * pi` (vö. $\lambda r. r \cdot r \cdot \pi$)
 - értéke: függvényérték
 - típusa: `real -> real`
 - neve: nincs, azaz *névtelen*

Függvényalkalmazás

- `e1 e2`, ahol `e1`-nek függvényértéket adó kifejezésnek kell lennie
- `e1 juxtapozícióban` áll `e2`-höz képest

Példák

- `sin 0.59`
- `(fn r => r * r * pi) 1.0`

Függvénydeklaráció

- `val area = fn r => r * r * pi`
- `fun area r = r * r * pi`

Példa az area függvény alkalmazására

- `area 1.0`