FUNKCIONÁLIS PROGRAMOZÁS

Tartalom

Az első rész tartalma

- Absztrakció függvényekkel (eljárásokkal)
  - Programelemek
  - Függvények (eljárások) és az általuk létrehozott folyamatok
  - Magasabb rendű függvények (eljárások)
- Absztrakció adatokkal
  - Az adatabsztrakció fogalma
  - Hierarchikus adatok
  - Absztrakt adatok többszörös ábrázolása
  - Polimorfizmus, generikus műveletek

Irodalom: [SICP] Abelson, Sussman & Sussman: *Structure and Interpretation of Computer Programs,* The MIT Press, 1996

Történeti áttekintés

A λ-kalkulus rövid története

- 1910: Russel & Whitehead: Principia Mathematica
- 1935: Gödel: nem létezhet erős axióma- és tételrendszer, amely teljes
- 1930-40-es évek: *Alonzo Church: λ-kalkulus*

A Russel paradoxon

- R: az önmagukat nem tartalmazó halmazok halmaza
- R tartalmazza-e önmagát? $\Rightarrow$ ellentmondás

A Russel paradoxon a λ-kalkulusban

- $R = \lambda x.\neg(x\ x)$
- $R\ R = \neg(R\ R) \Rightarrow$ ellentmondás

Típusos λ-kalkulus

- minden kifejezésnek van típusa
- f csak akkor alkalmazható e-re, ha a típusaik „passzolnak”
- R nem írható le

Történeti áttekintés (folyt.)

Funkcionális programozási nyelvek

- LISP (LISt Processing), 1950-es évek vége, MIT, US, John McCarthy; típus nélküli
  - bizonyos logikai kifejezések (ún. rekurzív egyenletek) igazolására
  - szimbolikus kifejezések kezelésére
- ML (Meta Language), Edinborough, GB, 1970-es évek közepe; típusos
- Scheme, 1975, MIT, US; típus nélküli
- SML (Standard Meta Language), 1980-as évek vége; típusos
- Miranda, 1980-as évek; típusos
- Haskell, 1990-es évek, US; típusos
- Common LISP, 1994, ANSI standard; típus nélküli
- Clean, 1990-es évek közepe, Nijmegen, NL; típusos
- Alice, 2003, Saarbrücken, DE; típusos

Funkcionális programozás

Ami közös a funkcionális nyelvekben

- Rekurzív eljárások (függvények)
- Rekurzív adatstruktúrák
- Eljárások (függvények) kezelése adatként

A következő hetekben

- *számítási folyamatokkal* és az általuk kezelt *adatokkal* foglalkozunk,
- programjainkat – a folyamatokat vezérlő szabályrendszert – az SML funkcionális nyelven írjuk,
- *eszközül* és az MOSML vagy a PolyML értelmezőt/fordítót használjuk,
- az absztrakcióról, modellezésről, programstruktúráról tanulandók más programozási nyelvek használatakor is hasznosak lesznek.

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Programelemek

Minden valamirevaló programozási nyelvben háromféle mechanizmus van a számítási folyamatok és az adatok leírására:

- primitív kifejezések,
- összetételi eszközök,
- absztrakciós eszközök.

Kifejezések

- állandók (jelölések), pl. `486`, `2.0`, `"text"`, `#"A"`
- összetett kifejezések, pl. `482+4`, `2.3-0.3`, `"te"^"xt"`, `op+(482,4)`, `#"A"< #"a"`

  Megjegyzések: `<` és `#` ún. tapadó írásjelek, ezért közéjük szóközt kell rakni a példában. Az `op` kulcsszó egy infix helyzetű operátort átmenetileg prefix helyzetűvé tesz.

Összetett kifejezések elemei

- operátor (műveleti jel, függvényjel)
- operandus (formális paraméter)
- argumentum (aktuális paraméter)

Példák az MOSML használatára

Az SML értelmező is ún. *read-eval-print* ciklusban dolgozik. A kiértékelés (*eval*) a `;`, majd az `enter` leütésére kezdődik el.

```
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
- 486;
> val it = 486 : int
- 2.3-0.3;
> val it = 2.0 : real
- "te"^"xt";
> val it = "text" : string
- op+(482,4);
> val it = 486 : int
- #"A"< #"a";
> val it = true : bool
- val it = 486;
> val it = 486 : int
```

Minden kifejezés kiértékelése valójában *értékdeklaráció:* ha nem adunk meg más nevet, az SML az `it` nevet köti az adott kifejezés értékéhez.

Névadás, (globális) környezet

Egy *értékdeklarációval* egy nevet kötünk egy értékhez:

```
- val size = 2;
> val size = 2 : int
- 5*size;
> val it = 10 : int
- val ||| = 3;
> val ||| = 3 : int
- ||| * size;
> val it = 6 : int
```

Megjegyzés: `|` és `*` ún. tapadó írásjelek, ezért közéjük szóközt kell rakni a példában.

Egy név lehet:

- alfanumerikus (az angol ábécé kis- és nagybetűiből, az _ és a `'` jelekből állhat, betűvel kell kezdődnie),
- írásjelekből álló (húszféle írásjel használható).

A *névadás* a legegyszerűbb absztrakciós eszköz (a programozási nyelvekben is).
A *név–érték* párt az SML a „memóriájában”, az ún. globális *környezetben* tárolja. Később látni fogjuk, hogy vannak ún. lokális környezetek is.

Nevek képzési szabályai

- Alfanumerikus név: kis- és nagybetűk, számjegyek, percjelek (`'`) és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik
  - Példák: `tothGyorgy  Toth_3_Gyorgy  toth'gyorgy  'gyurika`
  - Percjellel kezdődő név csak típusváltozót jelölhet.
- Írásjelekből álló név: az alábbi 20 *tapadó* írásjel tetszőleges, nem üres sorozata

  `! % & $ # + - / : < = > ? @ \ ~ ' ^ | *`
  - Példák: `++  <->  |||  ##  |=|`
- Speciális a szerepe az alábbi fenntartott jeleknek

  `( ) [ ] { } , ; . ...`
- Más jelentés nem rendelhető az alábbi fenntartott nevekhez és jelekhez

```
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 : :: :> _ | = => -> #
```

Egyszerű adattípusok

Típusnév	*Megnevezés*	*Könyvtár*
`int`	előjeles egész	`Int`
`real`	racionális (valós)	`Real`
`char`	karakter	`Char`
`bool`	logikai	`Bool`
`string`	füzér	`String`
`word`	előjel nélküli egész	`Word`
`word8`	8 bites előjel nélküli egész	`Word8`

A beépített operátorok és precedenciájuk

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

div $-\infty$, quot 0 felé kerekít. div és quot, ill. mod és rem eredménye csak akkor azonos, ha két operandusuk azonos előjelű (mindkettő pozitív, vagy mindkettő negatív).

Különleges állandók

- Előjeles egész állandó

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

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

- Racionális (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`

- Karakterállandó: a # jelet közvetlenül követő, egykarakteres füzérállandó (l. a következő lapon).

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

  Ellenpéldák: `# "a" #c #"" #'a'`

- Logikai állandó: csupán kétféle lehet.

  Példák: `true false`

  Ellenpéldák: `TRUE False 0 1`

Különleges állandók, escape-szekvenciák

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

Összetett kifejezés kiértékelése

Egy összetett kifejezést az SML két lépésben értékel ki:

1. először kiértékeli az operátort (műveleti jelet, függvényjelet) és argumentumait (aktuális paramétereit),
2. majd alkalmazza az operátort az argumentumokra.

Vegyük észre, hogy ez a kiértekelési szabály azért ilyen egyszerű, mert rekurzív.

Az egyszerű kifejezések kiértékelési szabályai:

1. az állandók (jelölések) értéke az, amit jelölnek,
2. a belső (beépített) műveletek a megfelelő gépi utasításokat aktivizálják,
3. a nevek értéke az, amihez a környezet köti az adott nevet.

Megjegyzés: a 2. pont a 3. pont speciális esetének is tekinthető.

Példa:

```
(2+4*6)*(3+5+7) = op*(op+(2,op*(4,6)),op+(op+(3,5),7))
```

A kifejezéseket ún. kifejezésfával ábrázolhatjuk (lásd SICP-ben). A levelek operátorok vagy primitív kifejezések (állandók, nevek). A kiértékelés során az operandusok alulról fölfelé „terjednek”.

Névtelen függvény, függvény definiálása

Névtelen függvény λ-jelöléssel:

```
(fn x => x*x) 2
```

- Az `fn`-t *lambdának* olvassuk.
- Az `x` a függvény formális paramétere (lokális érvényű név!).
- Az `x*x` a függvény törzse.
- A `2` a függvény aktuális paramétere.

Névadás függvényértéknek:

```
val square = fn x => x * x
```

Függvény alkalmazása:

```
val sumOfSquares = fn (x, y) => square x + square y

val f = fn a => sumOfSquares(a+1, a*2)
```

A felhasználó által definiált függvények ugyanúgy használhatók, mint a belső (beépített) függvények.

További példák SML-függvények definiálására

Egyszeres Hamming-távolságú ciklikus kód előállítása

- A függvényt pl. *táblázattal* adhatjuk meg:

00	01
01	11
11	10
10	00

```
fn 00 => 01
 | 01 => 11
 | 11 => 10
 | 10 => 00
```

- Változatok („klózok”): minden lehetséges esetre egy változat.
- Az `fn` (olvasd: *lambda*) névtelen függvényt, *függvénykifejezést* vezet be.
- A függvény néhány alkalmazása:
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 10`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 11`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 111`
- Mintaillesztés, (egyirányú) egyesítés
- Érthető, de nem robosztus (ui. ez a függvény *parciális* függvény).

További példák SML-függvények definiálására (folyt.)

Modulo n alapú inkrementálás (pl. $n = 5$)

- A függvényt általában *algoritmussal* adjuk meg (nem táblázattal), egyébként
  - az argumentum nem lehetne változó,
  - túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod 5`
  - az `i` ún. *kötött változó,* a névtelen függvény argumentuma
- A függvény néhány alkalmazása:
  - `(fn i => (i + 1) mod 5) 2`
  - `(fn i => (i + 1) mod 5) 4`
  - `(fn i => (i + 1) mod 5) 3.0` – Hiba!
- Ez a függvény definiálható két klózzal is:
  `fn 4 => 0 | i => i + 1`
- Az SML (a Prologgal ellentétben) mindig csak az *első* illeszthető klózt használja!

Függvényérték névhez kötése (függvényérték deklarálása)

- Láttuk, hogy nevet függvényértékhez ugyanúgy köthetünk, mint bármely más értékhez.
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
  - `val incMod = fn i => 4 => 0 | i => i + 1`
- Szintaktikus édesítőszerrel (`fun`)
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
  - ```
    fun incMod 4 = 0
      | incMod i = i + 1
    ```
- Alkalmazásuk argumentumra
  - `kovKod 01`
  - `incMod 4`

Fejkomment

Írjunk *deklaratív fejkommentet* minden (függvény)érték-deklarációhoz!

- ```
  (* kovKod cc = az egyszeres Hamming-távolságú, kétbites,
                  ciklikus kódkészlet cc-t követő eleme
     PRE: cc ∈ {00, 01, 11, 10}
  *)
  fun kovKod 00 = 01
    | kovKod 01 = 11
    | kovKod 11 = 10
    | kovKod 10 = 00
  ```
- `PRE` = *pre*condition, előfeltétel
- `PRE: cc ∈ {00, 01, 11, 10}` jelentése: a `kovKod` függvény `cc` argumentumának a `{00, 01, 11, 10}` halmazbeli értéknek kell lennie, ellenkező esetben a függvény eredménye nincs definiálva.
- ```
  (* incMod i = (i+1) modulo 5 szerint
     PRE: 5 > i >= 0
  *)
  fun incMod i = (i+1) mod 5
  ```

A függvény mint érték

- A függvény „teljes jogú”*(first-class)* érték a funkcionális programozási nyelvekben
  - A függvény típusa általában: $\alpha \to \beta$, ahol az α az argumentum, a β az eredmény típusát jelöli
  - A függvény maga is: érték. *Függvényérték.*
  - Fontos: a függvényérték *nem* a függvény egy *alkalmazásának* az eredménye!
  - Példák függvényértékre
    - $\sin$ (a típusa: *valós* $\to$ *valós*)
    - round (a típusa: *valós* $\to$ *egész*)
    - $\circ$ (függvénykompozíció; a típusa: $((\beta \to \gamma) * (\alpha \to \beta)) \to (\alpha \to \gamma)$)
  - Példák függvényalkalmazásra
    - $\mathrm{round}\ 5.4 = 5$, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
    - $\mathrm{round} \circ \sin$ (a típusa: *valós* $\to$ *egész*)
    - $(\mathrm{round} \circ \sin)1.0 = 1$ (a típusa: *egész*)

Két- vagy többargumentumú függvények

- Függvény alkalmazása két vagy több argumentumra
  1. Az argumentumokat *összetett adatnak* – párnak, rekordnak, listának stb. – tekintjük
     - pl. $f(1,2)$ az f függvény alkalmazását jelenti az $(1,2)$ *párra,*
     - pl. $f[1,2,3]$ az f függvény alkalmazását jelenti az $[1,2,3]$ *listára.*
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f\ 1\ 2 \equiv (f\ 1)\ 2$ azt jelenti, hogy
     - az első lépésben az f függvény alkalmazzuk az 1 értékre, ami egy *függvényt ad eredményül,*
     - a második lépésben az első lépésben kapott függvényt alkalmazzuk a 2 értékre, így kapjuk meg az $f\ 1\ 2$ függvényalkalmazás (vég)eredményét.
- Az $f\ 1\ 2$ esetben az f függvényt *részlegesen alkalmazható* függvénynek nevezzük.
- A programozó szabadon dönthet, hogy a függvényt részlegesen alkalmazható vagy pl. egy párra alkalmazható formában írja meg. A különbség *csak* a szintaxisban van. A részlegesen alkalmazható változat, mint látni fogjuk, rugalmasabban használható.
- Infix jelölés: $x \oplus y \equiv$ a $\oplus$ függvény alkalmazása az (x,y) párra mint argumentumra. Az infix operátor balra vagy jobbra köt.