Magasabbrendű funkcionális programozás

Scheme 1

Altrichter Márta

grimma@freemail.hu

Scheme történelmi áttekintése

- Első LISP (LISt Processing) fordítót McCarthy készítette az „MIT”-n.
  - Akkori nyelvekkel ellentétben új objektumokat tartalmazott: atomok, és listák.
  - Nem létezett centralizált fejlesztése a nyelvnek
  - LISP használók tradicionálisan ellennezték a nyelv „szabványosítási” kísérleteit.

  –> Számos dialektus fejlődött ki

- Az egyik ilyen dialektus a **Scheme**

A Scheme típusai

Alaptípusok - Boolean

- Értékei: `#t` - igaz, `#f` - hamis (A LISPek jellemzően nem rendelkeznek külön igazságértékkel, hanem a `nil` jelenti a hamis, és minden más az igaz értéket.)
- Predikátum: `boolean?`
- Negálás: `not`

Alaptípusok - Számok

- Négy féle számtípus:
  - egész számok (pl. 42)
  - racionális számok (pl.: 3/7)
  - valós számok (pl.: 3.14)
  - komplex számok (pl.: $6 + 3i$)
- Predikátumok: `number?`, `integer?`, `rational?`, `real?`, `complex?`
- Relációk: `eqv?` (általánosan nem csak számokra értelmezve), $=, <, >, <=, \ldots$
- Operátorok: `+`,`-`,`/`,`expt` (működik negatív illetve nem egész számokra is)
- Műveletek: `max`, `min`, `abs` ...

Alaptípusok - Szimbólumok

- Az egyszerű típusú értékek ön-kiértékelők (self-evaluating), azaz az interpreterbe beírva önmagukat adják vissza.
- A szimbólumokkal más a helyzet, mivel azokat általában azonosítóként vagy változóként kezeli a nyelv.
- Két féle szimbólum típus:
  - „interned": ha két „interned" szimbólum nevei `string=?` egyenlőek, akkor ugyanazt az objektumot jelölik. A hasonlítás miatt az „interned" szimbólum készítésekor a Scheme interpreter azonnal kisbetűvé konvertálja a nevet.
  - „uninterned": a fenti tulajdonság nem áll fent. Jóval ritkábban és kevesebb módszerrel készíthető szimbólum.
- Predikátum: `symbol?`
- Műveletek például:
  - `symbol->name` *symbol* - visszaadja a szimbólum nevét string-ként.
  - `intern` *string* - „intern" szimbólumot készít stringben megadott névvel

Alaptípusok - Karakterek

- Általános forma: <#*karakter*> vagy <#*karakter-név*> (pl.: #\a,#\A,#\tab stb.)
- Néhány nem nyomtatható karakternek van neve: #\newline, #\ ami egyenlő a #\space - szel.
- Predikátum: `char?`
- Relációk: `char=?`, `char<?`, `char>=?` .... Ezek megkülönböztetik a kis és nagy betüket, míg `char-ci=?`, `char-ci<?` ...nem.
- Kis és nagybetük közötti konverzió a `char-downcase` és `char-upcase` függvényekkel valósítható meg.

Összetett típusok - Karakterfűzérek

- A karakterfüzérek karakterek sorozatából állnak (nem összetévesztendő a szimbólumokkal, amik karakterek sorozata jelöl).
- Konstruktor:
  - `(string #\a#\b)` ami egyenlő a rövid jelölés: "ab" - vel
  - `(make-string 3)` ami egyenlő "\0\0\0" - vel
- Predikátum: `string?`
- Relációk: `string=?`, `string<?`, `string>=?`, . . . , illetve `string-ci=?` . . . .
- Műveletek például:
  - `string->length` *string* - nem negatív egész szám, a karakterfüzér hossza
  - `string->ref` *string k* - visszadja a k. karakterét a karakterfüzérnek
  - `string->set!` *string k char* - k. karakterét a stringnek átállítja az új karakterre, és „unspecified” értékkel visszatér

Összetett típusok - Vektorok

- A vektorok a karakterfüzérekhez hasonló sorozatok, csak az elemeik bármik lehetnek (akár vektorok is)
- Konstruktor:
  - `(vector 0 "Anna" 3)` => #(0 "Anna" 3) jelölésben (általános jelölés: `#(object ...)`)
  - `(make-vector` *k [object]*`)` - újonnan allokál k-méretű vektort, ha object definiálva van az összes elemet arra állítja, egyébként „unspecified" lesz az értékük.
- A `#(object ...)` jelölés csak külső reprezentációja a vektornak, nem egy kifejezés ami vektorként értékelődik ki. Mint a listáknál is látjuk majd a vektorokat is '-vel kell ellátni.

  (Pl.: '#(0 "Anna" 3) – > #(0 "Anna" 3) -re értékelődik ki).
- Predikátum: `vector?`
- Műveletek: `vector-ref`, `vector-set!`, `vector-length`

Összetett típusok - Párok és Listák

Párok

- A pár az egy adatstruktúra két mezővel: `car` és `cdr`.
- Jelölésmódok: <`(c1 . c2)`>, `(cons c1 c2)`.

Listák

- Listákat rekurzív módszerrel definiálhatjuk, azaz vagy az üres lista `()`, vagy egy pár aminek a `cdr`-je egy lista.
- Jelölések: `(a b c d e)`, `(a . (b . (c . (d . (e . ()))))))`, `(list` *obj1 obj2 ...* `)`, `(make-list` *k [element]* `)`
- Helytelen listák („improper lists”): ahol nem az üres lista a záróelem: `(a b c .d)`.
- Hasonlóan a vektorokhoz, itt is `(a b c d)` jelölés csak külső reprezentációja a listának, nem egy kifejezés ami vektorként értékelődik ki. A '-val kell ellátni.

Összetett típusok - Párok és Listák

(Pl.: `(cons '(a b) c))` => `((a b) . c)`

- Predikátum: `pair?`, `list?`
- Műveletek:
  - `car` *pair* - a pár `car` mezőjét visszadja. Üres listánál hiba. (Pl.: `(car '(a b c d))` => `a`
  - `cdr` *pair* - a pár `cdr` mezőjét visszadja. Üres listánál hiba.
  - `car` és `cdr` kompozíciói. (Pl.: `(cddar '('(a '(b c d)) e f))` => `(c d)`
  - `set-car!` *pair object* - objectet eltárolja car-ban. Visszatérési érték „unspecified"
  - `set-cdr!` *pair object* - objectet eltárolja cdr-ben. Visszatérési érték „unspecified"
  - `length-list` *list* - lista hossza, üres lista 0
  - `list-ref` *list k* - lista k. elemét adja vissza, 0-tól indexel

Egyéb típusok - Bit stringek, Rekordok

Bit stringek

- Bit karakterfűzérek bitek sorozata. Jobbról balra van számozva, a legjobboldali bit a nulladik bit. Nagyon tömören kódolt a memóriában.
- Jelölés: #*0001, #*01010, #*
- A bit string függvények a MIT-Scheme kiterjesztései:
  - `(make-bit-string` *k initialization* `)` - (Pl.: `(make-bit-string 5 #f)` => #*00000
  - Vannak rajta bitenkénti operációk (pl.: `bit-string=?, bit-string-and` . . .)

Rekordok

- `make-record-type` *type-name field-name* - rekord-tipus leírót készít. A mező nevek szimbólumok listája (azonos név duplán nem szerepelhet).
- `record-constructor` *record-type [field names]* - eljárást ad vissza amivel rekordokat lehet felvenni.
- `record?`

Egyéb típusok - ígéret (Promise)

- A Scheme alapvetően imperatív, mohó kiértékelésű nyelv funkcionális elemekkel. Az ígéret adattípus létrehozásával kívánták a lusta kiértékelés lehetőségét fenntartani.

- `delay` *expression* - létrehoz egy ígéret objektumot, amellyel a jövőben kiértékeltethetjük az „expression”-t.

- `force` *promise* - ha még nem értékelődött ki az ígéret akkor kiszámolja és visszaadja azt. Az érték „memorizálódik”, tehát ha újra lekérdezzük értékét akkor számítás nélkül visszaadja a memorizált értéket.

- Pl.: `(force (delay (+ 1 2)))` => 3

- További műveletek:
  - `promise?` *object*
  - `promised-forced?` *promise* - #t ha már memorizált egyébként #f
  - `promise-value` *promise* - ha már kiértékelődött a jövő, akkor azt visszaadja, egyébként hibát jelez.

Egyéb típusok - Lusta listák (Streams)

- Lusta listák hasonlítanak a listákhoz, azonban itt a lista farka addig nem értékelődik ki, amíg nem hivatkozunk rá. Íly módon létrehozhatunk végtelen hosszú listákat.

- `cons-stream` *object expression* = (`cons` *object* (`delay` *expression*))

- `stream-pair?` *object* - igazat ad vissza, ha az objektum egy pár és `cdr`-je egy ígéret.

Konverzió a típusok között

- A különböző adattípusok közötti konverzióra a Scheme rendelkezik a szükséges eljárásokkal:
  - `char->integer`
  - `integer->char`
  - `string->list`
  - `list->vector`
  - `vector->list`
  - `symbol->string`
  - `string->symbol`
  - `list->stream`
  - `stream->list`
  - `...`

Egyéb típusok

Input/Output

- Input/Output : a Scheme portokon keresztül valósítja meg. Ezek a portok is külön típusnak számítanak.

Eljárás típus

- A LISP nyelvek különlegessége: maguk az eljárásokat is adatként tudjuk reprezentálni és manipulálni.

Kifejezések

Kifejezések

Scheme kifejezés olyan konstrukció ami értéket ad vissza.

Literálok

- Az eddig definiált egyszerű önkiértékelő adatok: szám konstansok, karakterek, karakterfüzérek és booleanok.
- Illetve a '-t szükséglő adatok: listák, párok, szimbólumok, vektorok.

Változó referenciák

- `(define x 10)`
  x => 10

Speciális szintaxisú kifejezések

- *keyword component* . . .

Kifejezések

- Pl.: access, and, begin, case, cond, cons-stream, declare, default-object?, define, define-integrable, define-structure, define-syntax, delay, do, er-macro-transformer, fluid-let, if, lambda, let, let*, let-syntax, let*-syntax, letrec, letrec-syntax, local-declare, named-lambda, . . .

Eljárás hívás

- *(operator operand . . .)*
- LISP néhány egyéb dialalektusaival szemben a Scheme mindig ugyanazzal a kiértékelő szabályrendszerrel értékeli ki az operátort és a operandusokat, a kiértékelés sorrendje definiálatlan.
- Pl.: `(+ 3 4)` $\Rightarrow$ 7
- Pl.: `(if (#f = *) 3 4)` $\Rightarrow$ 12

A programozás elemei

Változó kötés, Környezetek fogalma, Static scope

Változó megkötés

- Egy változó megnevez egy helyet ahol az értékét tároljuk.
- „unbound” - a változó nincs hozzákötve egy ilyen helyhez
- „unassigned” - a változó megkötött, de nincs értéke

Környezet

- Környezet = megkötött változók halmaza. Ha egy változó nincs az adott környezetben megkötve akkor „unbound”.
- Új környezetet teremtünk a régi kiterjesztésével, ha változók egy halmazát megkötjük. Ilyenkor az új környezet elfedi a régit.
- Eljárás hívások (`let`, `let*`, `letrec`), belső változó megkötések (`define`) mind a környezet kiterjesztését eredményezik

Változó kötés, Környezetek fogalma, Static scope

- Létezik egy kezdeti környezet amit a Scheme automatikus biztosít.

Static scope

- Static scope azt jelenti, hogy a környezet amit kiterjesztünk, es ami az aktuális környezetté válik egy eljárás hívásakor nem az aktuális környezet, hanem az amiben az eljárást létrehoztuk.

- Static scope példa:

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y))
f 5 => 3
```

- Példa: scope.scm és scope2.scm

Feltételes szerkezetek

- Egy feltételes kifejezés viselkedését az dönti el, hogy értéke hamis-e. Csak #f számít hamisnak, minden egyéb #t, párok, vektorok, szimbólumok . . . helyesek.

- (`cond` *clause clause* . . .), ahol *clause* formája:
  - (*predicate expression* . . .) vagy
  - (else *expression expression* . . .)

  Pl.: `(cond ((3 > 3) 'nagyobb)`
  `((3 < 3) 'kisebb)`
  `(else 'egyenlő))` $\Rightarrow$ `egyenlő`

- (`if` *predicate consequent [alternative]*)

  Pl.: `(if (3 > 2) 'yes 'no)` $\Rightarrow$ `'yes`

- (`case` *key clause clause* . . .), ahol *clause* formája:
  - ((*object* . . .) *expression* . . .) vagy
  - (else *expression expression* . . .)

Feltételes szerkezetek

Pl.: `(case (* 2 4)`
`((1 2 3 5 7 11) 'prim)`
`((4 6 8 9 10) 'összetett) => összetett`

- (`and` *expression*) - kiértékel balról jobbra, és az első helytelennek az értékét visszaadja. Egyébként #t-val tér vissza.

Pl.: `(and (= 2 2) (< 3 4)) => #t`
`(and 1 'a '()) => '()`

- (`or` *expression*) - első igaz értéknek kiértékelődöttet adja vissza

Belső definícók és Blokk struktúra

- Abban az esetben ha egy adott eljárás segédeljárásokat használ, amelyek a felhasználó szempontjából lényegtelenek, akkor azokat elrejthetjük a felhasználó által hívott eljárás belső definícióiba.

- Általában ilyenkor az operandust nem szükséges továbbadni a belső eljárásoknak, hiszen azok a főeljárás révén az operandus megkötött környezetében dolgoznak.

- Lásd squareroot.scm példaprogram

Rekurzió és Iteráció

A Schemeben a rekurzió az iteráció egyetlen módja, nem rendelkezik ciklus szervező utasításokkal (azonban ezek makrók segítségével könnyen létrehozhatóak, ld. később)

Lineáris Rekurzió	Iteráció

`(define (factorial n)`
`  (if (= n 1)`
`  1`
`  (* n (factorial (- n 1)))))`
	`(define (factorial n)`
	`  (fact-iter 1 1 n))`
	`(define (fact-iter product counter max-count)`
	`  (if (> counter max-count)`
	`  product`
	`  (fact-iter (* counter product)`
	`  (+ counter 1) max-count)))`

`(factorial 5)`
`(* 5 (factorial 4))`
`(* 5 (* 4 (factorial 3)))`
`(* 5 (* 4 (* 3 (factorial 2))))`
`(* 5 (* 4 (* 3 (* 2 (factorial 1)))))`
`(* 5 (* 4 (* 3(* 2 1))))`
`(* 5 (* 4 (* 3 2)))`
`(* 5 (* 4 6))`
`(* 5 24)`
	`(factorial 5)`
	`(fact-iter 1 1 5)`
	`(fact-iter 1 2 5)`
	`(fact-iter 2 3 5)`
	`(fact-iter 6 4 5)`
	`(fact-iter 24 5 5)`
	`(fact-iter 120 6 5)`
	`120`

Eljárások mint argumentumok

- A Scheme eljárásai a λ-kalkulusra utalva a `lambda` kulcsszóval definiálhatók.
- Általános alak: (`lambda` *formals expression expression* . . .) - formals: paraméter lista
- Név nélküli eljárás példa: `(lambda (x) (+ 1 x))`, `define` segítségével nevesíthető:
  `(define plusz_egy (lambda (x) (+ 1 x)))` = `(define (plusz_egy x) (+ 1 x))`
- Az eljárásnak tetszőleges számú paramétere lehet.
- Definiálhatóak változó számú paramétert elfogadó eljárások is:
  - Scheme-ben: '.' - minden maradék paraméter bekerül egy listába
  - MIT Scheme-ben csak: #!optional - a szükséges argumentumok után ezekkel probál párosítani
  - MIT Scheme-ben csak: #!rest - minden maradék paraméter bekerül egy listába
- Lásd lambda_example.scm

Lokális változó definiálása „let”-tel vagy „lambda”-val

- Lokális változó definiálása `let`-tel:
  (`let` ((*variable init*) . . .) *expressions expresions* . . .)
  ⇕
  ((`lambda` (*variable* . . .) *expressions expresions* . . .) *init* . . .)

- (`let*` ((*variable init*) . . .) *expressions expresions* . . .) - hasonlít a `let`-re, azonban, itt az értékadások balról jobbra történnek (`let`-nél ismeretlen a sorrend). Így a második értékadás olyan környezetben történik, ahol az első már megtörtént.

- Lásd let_example.scm példán.

Rekurzió lokális változóknál

- (`letrec` ((*variable init*) ...) *expressions expresions* ...) - létrehoz egy új környezetet ahol az összes változó „unbound”, majd mindegyiket megköti az „init” ebben a környezetben adott eredményével. Így egy olyan környezethez jutunk, amelyben az összes megkötés létrejött, és ebben hajtódnak végre az kifejezések. Példák lásd let_example.scm-ban.

```
(letrec
   (
      (even?  (lambda (n)
         (if (zero?  n) #t (odd?  (- n 1)))))
      (odd?  (lambda (n)
         (if (zero?  n) #f (even?  (- n 1))))))
   (even?  88)) => #t
```

Szimbólumok használata

- Szimbólumok segítségével absztrakt adattípus hozható létre.
- Például megvalósítható a szimbolikus deriváltszámítás.
- Lásd a symbolic_diff.scm

Szekvencia bevezetése

- (`begin` *expression expression* ...) - a változók szekvenciálisan balról jobbra értékelődnek ki.

  ```
  (define x 0)
  (begin (set!  x 5) (+ x 1)) => 6
  ```

- Sokszor szükségtelen a használata, mert azt sok speciális hívás amúgy is elősegítí már: `case, cond, define (csak eljárás def.)`
  `,do fluid-let, lambda, let*, named-lambda ...`

Ígéret és Lusta Lista elemzése

- Lusta kiértékelésre példa:
- (`cons-stream` <*a*> <*b*>) <=> (`cons` <*a*> (`delay` <*b*>))
- Ezért például:
```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```
- Lusta lista használatára példa: lazy_list.scm