A Scheme programozási nyelv

Ez a vázlat nem része a Algoritmusok elmélete tantárgy hivatalos tananyagának. A hibákat és a megjegyzéseket a szerző a `terek@cs.bme.hu` címen várja.

1. Adattípusok

Beépített adattípusok

bool	`#t`, `#f`	`boolean?`
szám	`-128`, `1.085`, *stb.*	`number?`
string	`"hello"`	`string?`
vektor	`#(`o_1 o_2 ... `)`	`vector?`
azonosító	`hello`, `+`, `number->string`	`symbol?`
függvény	*`(lambda (x) (+ x x))`*	`procedure?`
pár	`("a" . 12)`	pair?
üres lista	()	null?

A lista vagy üres lista `()`, vagy egy pár, amelynek első eleme (`car`) tetszőleges, a második (`cdr`) pedig egy lista.

```
(1 2) = (1 . (2 . ()))
(1 "h" 10 13) = (1 . ("h" . (10 . (13 . ()))))
((1 2) 3 4) = ((1 . 2) . (3 . (4 . ())))
(lambda (x) (+ x x)) =
   (lambda . ((x . ()) . ((+ . (x . (x . ()))) . ())))
```

2. Kiértékelés

Az adat *kiértékelhető*. Egy program futtatása: a külső reprezentáció (szöveg) először adattá konvertálódik, majd kiértékelődik.

- A egyszerű adattípusok saját magukká értékelődnek ki (szám, bool, string).
- Ha valami előtt `'` áll, saját magává értékelődik ki.
- Azonosító vagy egy *szintaktikus formát* jelöl, vagy az általa jelölt névhez kötött objektummá értékelődik ki. (Kötés: változó név/érték összerendelés).
- Lista: az első elem kiértékelése alapján:
  - ha az egy függvény, meghívódik a lista többi eleméből kiértékelt paraméterekkel, és a visszatérési érték lesz a lista kiértékelésének eredménye;
  - ha egy szintakszist jelöl, akkor az a konkrét szintakszis definiálja, hogy mi történik a lista többi elemével, és mivé értékelődik ki a lista.

Néhány példa kiértékelésre:

```
(+ 1 (* 3 2))              ⟹ 7
(number->string 12.3)      ⟹ "12.3"
(lambda (x) (+ x 1))       ⟹  egy függvény
(write 10)                 ⟹  nemdefiniált
                  mellékhatásként ír a kimenetre
(cons 10 #t)               ⟹ (10 . #t)
```

3. Szintaktikus formák

Feltételes kifejezések esetén, ha a feltétel nem `#f`, a harmadik elem értékelődik ki, különben a negyedik:

`(if` *feltétel igaz-kif hamis-kif*`)`

A `define` az aktuális „névtartományhoz” (scope) hozzáad egy új név-objektum összerendelést:

```
(define a 4)               ⟹  nemdefiniált
(define b (lambda (x) (+ x 1)))
                           ⟹  nemdefiniált
(b a)                      ⟹ 5
```

Lokális névtartományok létrehozhatók a `let`, `let*` és `letrec` szintakszisokkal.

A `let` esetén a névhez rendelt értékek mind a külső névtartományban értékelődnek ki.

```
(define a 3)
(let ((a 2)
      (b a))
  (+ a b))                 ⟹ 5
a                          ⟹ 3
```

A `let*` esetén a névhez rendelt értékek mindig az előző lokális kötésekkel kiegészített névtartományban értékelődnek ki.

```
(define a 3)
(let* ((a 2)
       (b a))
  (+ a b))                 ⟹ 4
```

A `letrec` esetén az összes kötés azonnal él, de inicializálatlanul, nem szabad közvetlenül hivatkozni:

```
(letrec ((páros?
          (lambda (n)
            (if (zero? n)
                #t
                (páratlan? (- n 1)))))
         (páratlan?
          (lambda (n)
            (if (zero? n)
                #f
                (páros? (- n 1))))))
  (páros? 88))             ⟹ #t
```

A `lambda` kulcsszóval függvény (closure) hozható létre, amely a lista második elemében felsorolt nevekkel bővítve az aktuális névtartományt, a lista többi elemének kiértékelését végzi a meghívása során. A meghíváskor az átadott paraméterek kötődnek majd értékként a formálisan megadott nevekhez. Ha a paraméterek listája helyett egyetlen azonosító szerepel, az paraméterekből alkotott *listára* hivatkozhatunk az adott névvel.

```
((lambda (a b) (+ a b)) 3 4)⟹ 7
((lambda l (car l)) 8 9 10) ⟹ 8
```

Egy adott névhez kötött változó módosítható a `set!` szintakszissal:

```
(define a 3)
(set! a (+ a 1))
```

4. Példák

Globális (top-level) változó:

```
(define c 10)
(define a (lambda (b)
              (+ b c)))
```

Egyszerű lokális változó:

```
(define a (lambda (b)
            (let ((c 10))
              (+ b c))))
```

Rejtett, de globális változó:

```
(define a (let ((c 10))
            (lambda (b)
               (set! c (+ c 1))
               (+ b c))))
```

Magasabb rendű függvény:

```
(define hozzáadó
  (lambda (x)
    (lambda (y)
      (+ x y))))
(define három-hozzáadó (hozzáadó 3))
(három-hozzáadó 5)          ⟹ 8
```

Adat-rejtés:

```
(define make-counter
  (lambda ()
     (let ((counter 0))
        (lambda ()
          (set! counter (+ counter 1))
          counter))))
(define c1 (make-counter))
(c1)                        ⟹ 1
(c1)                        ⟹ 2
(define c2 (make-counter))
(c2)                        ⟹ 1
(c1)                        ⟹ 3
```

5. Jobbrekurzió

A ciklusszervezés az eddig bemutatott nyelvi eszközökkel is megvalósítható, rekurzív hívások segítségével.

```
(define ismétel
  (lambda (hányszor f)
    (if (> hányszor 0)
        (begin
          (f)
          (ismétel (- hányszor 1) f)))))
```

Egymásba ágyazott függvényhívások esetében bizonyos feltételek mellett lehetőség van a verem további mélyülésének megakadályozására. Pl. egy X függvény visszatérési értéke a törzs utolsó helyén álló Y kifejezés kiértékelt változata. Ha Y egy függvény, annak meghívása előtt – a paraméterek kiértékelését követően — X „verem-kerete” (stack-frame) felszabadítható.

```
(define a
  (lambda ()
     (write "hello")
     (a)))
```

Azokat a pozíciókat, ahol egy függvény hívása a verem mélyítése nélkül elvégezhető, *tail-context*-nek hívják. Ilyen többek között a `let` család tagjainak valamint a `lambda` törzsének utolsó helye, az `if` mindkét ága, stb.

6. Memóriaszervezés

A memóriaszervezés alapelve, hogy a programozó nem szűntethet meg explicit módon egy objektumot. A rendszer jogosult felszabadítani azokat a memóriacellákat, amelyekről bizonyítható, hogy többet soha nem használja őket a program.

Ezt általában *szemétgyűjtéssel* (garbage collection) oldják meg, ami azt jelenti, hogy időnként a futtató környezet megjelöli azokat a cellákat, amelyek a jelenlegi pontról közvetlenül láthatók változóneveken keresztül, továbbá rekurzívan azokat a cellákat, amelyek az ilyen cellákban lévő objektumokból elérhetők. (Ilyenek pl. egy lista elemei, függvény által használt változók, stb.)

7. Makrók

A Scheme *higiénikus makró* rendszerrel rendelkezik. Makrókkal a nyelv szintakszisát bővítjük egy ún. *átíró* (transformer), amely leírja, milyen átalakított kifejezésen folytatódjék a kiértékelés. Globális és lokális makró definíciók is adhatók (`define-syntax`, `let-syntax`, `letrec-syntax`). Terminológia: szintaktikus kulcsszót kötünk egy makró átíróhoz.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))


(let-syntax ((when (syntax-rules ()
                    ((when test stmt1 stmt2 ...)
                     (if test
                         (begin stmt1
                                stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                    ⟹ now
```

Példa a `let` szintakszis definiálására `lambda`-val:

```
(define-syntax let
  (syntax-rules ()
    ((let ((nam1 obj1) ...) stmt1 ...)
       ((lambda (nam1 ...) stmt1 ...) obj1 ...))))
```