Magasabbrendű funkcionális programozás

Hanák Péter

hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

OM Kutatás-Fejlesztési Helyettes Államtitkárság

Hanák Dávid

dhanak@inf.bme.hu

Számítástudományi és Információelméleti Tanszék

Csala Viktor

viktor@csala.net

AAM Technologies Kft.

BEVEZETÉS

Bevezetés

Az előadás felépítése

- 1. alkalom: A Scheme története
  A nyelv alapvető jellemzői
  Típusok
  Összetettebb kifejezések
- 2. alkalom: I/O
  Makrók
  Asszociációs listák
  Struktúrák
  Call-with-current-continuation
- 3. alkalom: Objektum-orientált programozás
  Nem-determinisztikus végrehajtás
  Érdekességek

Források

- Guy L. Steele Jr., Richard P. Gabriel: *The Evolution of Lisp*, The Second ACM SIGPLAN History of Programming Languages Conference, 1993.
- Dorai Sitaram: *Teach Yourself Scheme in Fixnum Days* (http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html). Ez képezi az előadás anyagának törzsét.
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi: *How to Design Programs - An Introduction to Computing and Programming* (http://www.htdp.org/).
- Guy L. Steele Jr.: RABBIT: A Compiler for Scheme (A Study in Compiler Optimization). MIT Artifical Intelligence Laboratory, 1978.
- David Andrew Kranz: ORBIT: An Optimizing Compiler For Scheme. Yale University, 1988.
- *Revised^5 Report on the Algorithmic Language Scheme*

Eszközök

- DrScheme – interaktív futtató környezet, amelyben a programozó tapasztaltságának megfelelően több nyelvi szint is beállítható. Szabadon letölthető dokumentációval együtt a www.drscheme.org site-ról. Ugyanott a tanulást segítő dokumentumok és linkek is találhatók.
- MIT Scheme- Emacs klónra épített interaktív futtató környezet. Letölthető a http://www.swiss.ai.mit.edu/projects/scheme/ site-ról. Ugyanott megtalálható a dokumentációja is (szintaxisa kicsit eltér a szabványostól).
- További implementációk:
  - GUILE
  - KAWA
  - Scheme 48
  - SCM
  - VSCM

A Scheme története

- Gerald J. Sussman és Guy L. Steele 1975-ben el kezdtek foglalkozni Carl Hewitt aktor elméletével. Mivel volt néhány részlet Hewitt munkáiban, ami tisztázásra szorult, így úgy döntöttek, hogy kidolgoznak egy segédnyelvet a hozzá tartozó interpreterrel együtt. Ennek megfelelően MacLispet használva írtak egy egyszerű Lisp interpretert az aktorok használatát támogató nyelvi elemekkel kiegészítve.
- A segédeszköz meglepően jól sikerült, elnevezték Schemer-nek (a Planner és Conniver MI eszközök mintájára), de mivel az ITS operációs rendszer csak 6 karakteres neveket engedett meg, így a Scheme-t használták, ami végül rajta maradt.
- 1978-ban Steele összegyűjtve addigi eredményeiket elkészítette a RABBIT-ot, az első Scheme fordítót, és ugyanebbe az évben kiadták a nyelv első leírását (Revised Report on the Algorithmic Language Scheme – a címmel utalva az Algolra)

A SCHEME BEMUTATÁSA

Alapvető jellemzők

- A Scheme imperatív, mohó kiértékelésű nyelv, funkcionális jellemzőkkel.
- A Scheme a Lisp család tagja, ez első pillantásra észrevehető:
  - Lista alapú működés (eredetileg a Lisp-ben minden lista volt, ami szép lassan átalakult, de jelei a mai napig láthatóak).
  - Zárójelezett, prefix jelölési mód.
- De vannak különbségek:
  - Minden S-kifejezés (ld. később).
  - Futásidejű, dinamikus típusrendszerrel rendelkezik.
- A Scheme, akárcsak a Common Lisp lexikai láthatóságot valósít meg (minden változó az őt befoglaló blokkban látható, ellenőrzése fordítási időben történik), szemben az eredeti Lisp-pel és az Emacs Lisp-pel (amelyek dinamikus láthatóságot használnak, minden változó miután létrejött mindenhol látható).

A SCHEME TÍPUSAI

Alaptípusok - Boolean

- Két érték: #t az igaz és #f a hamis jelölésére (a Lisp-ek jellemzően nem rendelkeznek külön igazságértékekkel, hanem a nil jelenti a hamis és minden más az igaz értéket, ezt a Scheme is így kezeli az esetek többségében).
- Predikátum: boolean?
- Negálás: not

Alaptípusok - Számok

- A Scheme négy féle szám típust különböztet meg:
  - egész számok (pl. 42)
  - racionális számok (pl. 22/7)
  - valós számok (pl. 3.1416)
  - komplex számok (pl. 3+6i)
- Predikátumok: number?, complex?, real?, rational?, integer?
- Relációk: eqv? (általános, nem csak számokra értelmezett), =, <, <=, stb.
- Operátorok: +, -, *, /, expt (jellemzően 2-től különböző argumentummal is használhatók)
- Műveletek: max, min, abs, stb.

Alaptípusok - Karakterek

- A karaktereket a #\ prefix-szel jelöli a Scheme, pl. #\a az a karakter.
- Néhány fontosabb, nem nyomtatható karakternek van saját neve, pl. #\newline, #\tab. A szóköz jelölhető #\ -vel is, de sokkal olvashatóbb a #\space.
- Predikátum: char?
- Relációk: char=?, char<?, char>=?, stb., ezek megkülönböztetik a kis és nagy betűket, illetve, char-ci=?, char-ci<?, stb., amik nem.
- A kis és nagy betűk közötti konverzió a char-downcase és char-upcase függvényekkel végezhető el.

Alaptípusok - Szimbólumok

- Az egyszerű típusú értékek a Scheme-ben ön-kiértékelőek (self-evaluating), vagyis az interpreterbe beírva önmagukat adják vissza.
- A szimbólumokkal más a helyzet, mivel azokat általában azonosítóként, vagy változóként kezeli a nyelv. Ennek ellenére a szimbólumok legális Scheme adattípusok.
- A szimbólumok kiértékelése a quoting használatával kerülhetők ki: (quote xyz) vagy röviden `xyz. A szimbólumok nevére az egyetlen kikötés, hogy nem ütközhetnek más adattípusok értékeivel.
- A szimbólumok neveiben nem különböztetjük meg a kis és nagy betűket.
- Predikátum: symbol?
- Az xyz szimbólum használható globális változóként is: (**define** xyz 9), ekkor xyz beírásakor 9-et kapunk eredményként.
- A változó értéke a set!-tel változtatható meg (a ! általában a nem funkcionális elemeket jelöli)

Összetett típusok – Karakterfüzérek

- A karakterfüzérek karakterek sorozatából *állnak* (nem összetévesztendő a szimbólumokkal, amiket karakterek sorozata *jelöl*), rövid jelölése: ”ABCabc123”.
- Konstruktor: (string #\a #\b #\c) $\Rightarrow$ ”abc”, illetve (make-string 3) $\Rightarrow$ ”\0\0\0”
- Predikátum: string?
- Relációk: string=?, string<?, string>=?, stb., ezek megkülönböztetik a kis és nagy betűket, illetve, string-ci=?, string-ci<?, stb., amik nem.
- Műveletek karakterfüzéreken: string-ref, string-append, stb.
- A string változók karakterei a string-set!-tel írhatók át.

Összetett típusok – Vektorok

- A vektorok a karakterfüzérekhez hasonló sorozatok, csak az elemei bármik lehetnek (akár vektorok is).
- Konstruktor: (vector 0 1 2 3 4) $\Rightarrow$ #4(0 1 2 3 4), illetve (make-vector 5) $\Rightarrow$ #5(0) vagy (make-vector 4 2) $\Rightarrow$ #5(2)
- Predikátum: vector?
- Vektor kezelő függvények: vector-ref, vector-set!, vector-length, stb.

Összetett típusok – Párok

- A pár 2 tetszőleges objektumból képzett összetett típus.
- Konstruktor: (cons 2 #t) $\Rightarrow$ (2 . #t), az első elemet a car, a másodikat a cdr operátorral kaphatjuk vissza.
- A pár nem ön-kiértékelő, így konstans párt aposztróffal kell leírni (ellenkező esetben függvényhívásnak tekinti): ’(1 . #t)
- A pár bármely eleme lehet összetett típusú is, akár egy másik pár: ’((2 . ”aaa”) . 4). Ekkor a (car (car ’((2 . ”aaa”) . 4))) hívás eredménye 2. A car és cdr többszöri alkalmazására rövidítések használhatók, a c....r forma (pl. caaaar, caddar, cadr, cdar, stb.), négy szintig minden szabványos Scheme implementációban garantáltan létezik.
- A (cons 1 (cons 2 (cons 3 (cons 4 5)))) konstrukció eredménye fontossága miatt saját jelöléssel rendelkezik: ’(1 2 3 4 . 5)
- Predikátum: pair?

Összetett típusok – Listák

- A lista a pár speciális esete, amit a (cons 1 (cons 2 (cons 3 ...))) konstrukcióval hozunk létre, ahol az utolsó pár második eleme a '() (nil, és nem ön-kiértékelő). Jelölése: '(1 2 3 4 5) (ez a '(1 2 3 4 5 . ()) rövidített formája).
- Konstruktor (a fenti forma is használható, de túl hosszú): (list 1 2 3 4)
- Predikátumok: list?, null? (és természetesen a pair? is használható).
- Eljárások: list-ref, list-tail, stb.

Konverzió a típusok között

- A különböző adattípusok közötti konverzióra a Scheme rendelkezik a szükséges eljárásokkal:
  - char-integer
  - integer-char
  - string-list
  - list-string
  - list-vector
  - vector-list
  - string-number (A második, opcionális argumentumával a számrendszer alapja is megadható)
  - number-string
  - symbol-string
  - string-symbol
  - ...

Egyéb típusok

- Két további adattípus létezik még a Scheme-ben:
  - Mivel minden értékkel bíró kifejezésnek számít, így létezik az eljárás típus is: cons ⇒ #<procedure:cons> (illetve bizonyos megvalósításokban #<primitive:cons>).
  - A másik típus a I/O-hoz kapcsolódik, amit a Scheme portokon keresztül valósít meg. Ezek a portok is külön típusnak számítanak.

S-kifejezés

- Az eddig felsorolt összes adattípus beletartozik az s-kifejezések (s a symbolic angol szó rövidítése) halmazába, gyakorlatilag a Scheme-ben minden s-kifejezésnek számít.
- Az s-kifejezések jellemzően zárójelezett, prefix jelölésmódú kifejezések.
- Az s-kifejezést eredetileg a Lisp első megvalósításaiban az olvashatóbb m-kifejezések (m, mint meta – eredetileg a Lisp szintaxisának szánták, de teljesen soha nem definiálták) belső, gépi reprezentációjának szánták, de mivel a programozók között nagyon gyorsan elterjedt, így az eredeti elképzelést elvetették.

A SCHEME ÖSSZETETT KIFEJEZÉSEI

Globális változók

- Globális változók a **define** paranccsal hozhatók létre, ezekre a definíciót követő programszövegben bárhol lehet hivatkozni (lexikai láthatóság).
- Változó értéke bármi lehet, így akár eljárás is.
  - (**define** x 2)
  - (**define** c cons)
  - (**define** a (cons 2 (cons 3))
  - ...

Eljárások

- A Scheme eljárásai a λ-kalkulusra utalva a **lambda** kulcsszóval definiálhatók.
- Név nélküli eljárás: (**lambda** x (+ x 2)) és ((**lambda** x (+ x 2)) 5) ⇒ 7
- A **define** parancs használatával az eljárásoknak név is adható (ez ebben a formában pongyola fogalmazás, mert nem adunk nevet az eljárásnak, csak eltároljuk a megadott nevű globális – vagy lokális – változóban).
  pl. (**define** f (**lambda** x (+ x 2))), és (f 5) ⇒ 7.
- Az eljárásnak tetszőleges számú paramétere lehet, pl. (**lambda** (x y) (* x y))
- Definiálhatók változó számú paramétert elfogadó eljárások is, a következő formában: (**lambda** (x y . z) (...)), ahol két paraméter megadása minimum kötelező (természetesen (v x y . z) esetében három paramétert kell kötelezően megadni), ha több van, akkor a harmadiktól kezdve listaként bekerülnek z-be (ha csak kettő van, akkor z értékel () lesz).
- Az apply eljárással valósítható meg a currying, pl. (apply + '(1 2 3)).

Szekvencia

- Ahogy már az alapvető jellemzőknél említettük, a Scheme igazából imperatív nyelv, így természetes módon támogatja az eljárások szekvenciális végrehajtását. Ennek megfelelően létezik a **begin** parancs:
  pl. (**lambda** (a b c)
  (**begin**
  (display a)
  (display ” ”)
  (display b)
  (display ” ”)
  (display c)
  (newline)))
- A **begin** parancsot nem kötelező kiírni.
- A szekvenciának, mint kifejezésnek az értéke, minden esetben az utolsó elemének értékével egyezik meg.

Feltételes kifejezések

- Alapforma: (**if** *teszt-kifejezés igaz-ág hamis-ág*), a hamis ág elhagyható, a *hamis-ág* elhagyható, ilyen esetben, ha a *teszt-kifejezés* mégis hamis, akkor a kifejezés értéke nem specifikált (jellemzően semmi).
- Az egy-ágú feltételes kifejezésekre néhány Scheme implementációban léteznek a **when** (csak *igaz-ág*) és az **unless** (csak *hamis-ág*) formák.
- Egymásba ágyazott **if** kifejezések rövidített leírására létezik a **cond** kifejezés: pl. (**lambda** x (**cond** ((< x 0) -1) ((= x 0) 0) (**else** 1))).
- A **cond** egyszerűsített formája a **case**, ami a C nyelv switch utasításához hasonlóan működik: pl. (**case** c ((#\a) 1) ((#\b) 2) (**else** 3))
- A boolean kifejezések kombinálására létezik az **and** és **or** forma (a not eljárás, ezek nem), hasonlóak az andalso és az orelse SML-es formákhoz (ha eljárások lennének, akkor a mohó kiértékelés miatt nem intuitív módon működnének).

Lokális értékadás

- Mint minden funkcionális nyelvben, itt is létezik a let forma, amely kvázi párhuzamosan végzi el az értékadásokat, ezért azok nem hivatkozhatnak egymásra, pl. (**let** ((x 1) (y 2) (z 3)) (list x y z)) ⇒ (1 2 3), de
  (**define** x 2)
  (**let** ((x 1) (y x)) (list x y)) ⇒ (1 2)
- A helyi értékadások szekvenciális végrehajtásához a let* formát kell használni: pl. (define x 2) (let* ((x 1) (y x)) (list x y)) ⇒ (1 1)
- A változók ideiglenes értékadására használható a fluid-let forma
  pl. (**fluid-let** ((counter 99))
  (display (next)) (newline)
  (display (next)) (newline)
  (display (next)) (newline))
- A fluid-let nem vezet be új változót, ahogy a let teszi.

Rekurzió - 1

- Egy eljárás törzse természetesen hivatkozhat más eljárásra, így perszer önmagára is:

```
pl. (define factorial
        (lambda (n)
            (if  (= n 0) 1
                 (* n (factorial (- n 1))))))
```

- Ugyanilyen módon definiálhatóak kölcsönösen rekurzív eljárások:

```
(define is-even?
   (lambda (n)
       (if  (= n 0) #t
            (is-odd? (- n 1)))))
(define is-odd?
   (lambda (n)
       (if  (= n 0) #f
            (is-even? (- n 1)))))
```

Rekurzió - 2

- Az eddig bemutatott lokális értékadási formák egyike sem használható kölcsönösen rekurzív eljárások létrehozására, ezért a Scheme-ben ilyen célból használható a letrec utasítás:

```
(letrec    ((local-even?
                (lambda (n)
                    (if  (= n 0) #t
                         (local-odd? (- n 1)))))
           (local-odd?
                (lambda (n)
                    (if  (= n 0) #f
                         (local-even? (- n 1))))))
           (list (local-even? 23) (local-odd? 23)))
```

- A Scheme-ben a rekurzió az iteráció egyetlen módja, nem rendelkezik ciklus szervező utasításokkal (de azok makrók segítségével könnyedén létrehozhatók, ld. később).

SCHEME I/O

I/O – portok

- A Scheme az I/O-t portokon keresztül valósítja meg, amik önmaguk is teljes értékű kifejezések, saját adattípussal.
- A portok hozzárendelhetők:
  - a konzolhoz
  - file-okhoz
  - karakterfüzérekhez.
- Az I/O műveletek nevének végén, annak ellenére, hogy mellékhatásuk van és nem tekinthetők tisztán funkcionális utasításoknak, alapesetben nem szerepel a ! (a ! gyakorlatilag akkor szerepel, ha az adott utasítás valamely változó – pontosabban név – kötést változtat meg).

I/O – olvasás

- Minden Scheme olvasó utasításhoz megadható opcionálisan valamilyen bemeneti port, amennyiben ezt a programozó elhagyja, úgy az alapértelmezett bemeneti portot használja, ami rendszerint a konzol.
- A beolvasás lehet
  - karakter (read-char),
  - sor (read-line),
  - s-kifejezés alapú (read).
- Minden olvasási művelet megváltoztatja a bemeneti port állapotát, így a soron következő olvasó utasítás már csak a még be nem olvasott tartalomhoz fér hozzá.
- Ha az adott porton már nincs több olvasható tartalom, akkor az olvasási művelet egy speciális file vég jelet ad vissza. Ez az egyetlen jel, amire az eof-object? predikátum igaz értéket ad.

I/O - írás

- Az olvasáshoz hasonlóan minden Scheme író utasításhoz megadható opcionálisan valamilyen kimeneti port, amennyiben ezt a programozó elhagyja, úgy az alapértelmezett kimeneti portot használja, ami rendszerint a konzol.
- Az írás csak karakter és s-kifejezés alapú lehet.
- write-char kiírja a megadott karaktert (#\ nélkül)
- A write és a display kiírja a megadott s-kifejezést, a különbség annyi, hogy a write az nyelv által is értelmezhető formában (pl. karakterfüzéreket ”-vel határolva, a karaktereket #\-sel együtt stb.) írja ki, míg a display nem.
- A newline utasítás új sor jelet küld a kimenetre.

I/O – file-ok

- Az alapéretelmezett be és kimeneti port a standard input és output, ekkor nem kell megadni semmit. Ha mégis, akkor használhatóak az argumentumok nélküli current-input-port és current-output-port utasítások.
- A file-ok portokhoz rendelése a file megnyitásával történik:
  - open-input-file
  - open-output-file
- A file portokat használat után mindig le kell zárni a close-input-port, illetve a close-output-port utasításokkal.
- Például (tegyük fel, hogy a haho.txt file a haho szót tartalmazza):
  (**define** i (open-input-file ”haho.txt”))
  (read-char i) $\Rightarrow$ #\h
- A file-ok megnyitása és lezárása a call-with-input-file és a call-with-output-file makrókkal automatikusan elvégezhető.

I/O – karakterfüzérek

- A karakterfüzérek kezelése a file-okéhoz analóg módon történik, vagyis a tényleges „I/O” műveletek előtt a portot meg kell nyitni:
  - open-input-string
  - open-output-string
- Példa:
```
(define o (open-output-string))
(write 'hello o)
(write-char #\, o)
(display "#\space" o)
(display "world" o)
```
- A kimeneti porton összegyűjtött karakterfüzér a get-output-string eljárással kapható meg. A fenti példa folytatásaként:
  (get-output-string o) ⇒ "hello, world"
- A karakterfüzérekre nyitott portokat nem kell lezárni (erre nem is léteznek utasítások).

Program file-ok beolvasása

- Program file-ok a load eljárással olvashatók be, a megadott útvonalat relatív útvonalként kezeli, ahol kiindulási pont az a könyvtár, ahonnan a programozó vagy felhasználó a futattó környezetet (vagy futó Scheme programot) indította. Ez az esetek többségében nem túl kényelmes, rendszerint a teljes elérési (abszolút) útvonal megadását teszi szükségessé.
- A Scheme implementációk jelentős része ennek a problémának a kiküszöbölésére bevezette a load-relative utasítást, amely szintén relatív útvonalat kér, de nem a keretrendszer, hanem az azt kiadó program kódot tartalmazó file könyvtárát tekinti kiindulási pontként.

MAKRÓK

Makrók - bevezetés

- A Scheme makrók definiálásával lehetőséget ad a programozóknak, hogy saját speciális eszközeiket létrehozzák.
- Makrók definiálására a define-macro szolgál.
- A makró definíció egy szimbólumhoz hozzárendeli a megfelelő transzformáló eljárást. Az interpreter vagy a fordító minden esetben végrehajtja ezt az eljárást, amikor az adott szimbólummal találkozik.
- Példa: a when utasítás definíciója (ha az adott implementáció nem támogatja):

```
(define-macro when
   (lambda (test . branch)
      (list 'if test
         (cons 'begin branch))))
```

Makrók – template-ek

- A when definiálásának példáján is látszott, hogy a makrók jellemzően az utasítások listáinak felépítését végzik el. Bonyolultabb esetben ez már nagyon átláthatatlanná válhat, ezt a problémát oldja fel a template-ek használata.
- Példa: a when utasítás template-ekkel:

```
(define-macro when
   (lambda (test . branch)
      `(IF ,test
         (BEGIN ,@branch))))
```

- A ` vezeti be a lista template-et, a , és a ,@ pedig a behelyettesítéseket jelöli (a , egyszerű behelyettesítés, míg a ,@ listát vár és azt zárójelek nélkül illeszti be). A terminátor elemeket hagyományosan csupa nagy betűkkel írják (lévén alapesetben a Scheme implementációk nem különböztetik meg a kis és nagy betűket).

Makrók – egyedi változók

- Tekintsük az **or** alábbi konstrukcióját:

```
(define-macro my-or
   (lambda (x y)
      `(if ,x ,x ,y)))
```

- A fenti definíció hibás, mert az x-nek átadott kifejezést kétszer értékeli ki, így ha annak mellékhatásai vannak, akkor nem várt eredményre vezethet. Megoldásnak igérkezik egy ideiglenes változó bevezetése:

```
(define-macro my-or
   (lambda (x y)
      `(let ((temp ,x))
         (if temp temp ,y))))
```

- De ez sem tökéletes, pl. (**define** temp 3) (my-or #f temp) ⇒ #f (és nem 3)
- Ezt a problémát küszöböli ki teljesen a gensym utasítás használata, ami minden hívásakor egy garantáltan egyedi változónevet ad vissza.

Makrók – fluid-let (példa)

```
(define-macro fluid-let
  (lambda (xexe . body)
    (let ((xx (map car xexe))
          (ee (map cadr xexe))
          (old-xx (map (lambda (ig) (gensym)) xexe))
          (result (gensym)))
      `(let ,(map (lambda (old-x x) `(,old-x ,x))
                  old-xx xx)
         ,@(map (lambda (x e)
                  `(set! ,x ,e))
                xx ee)
         (let ((,result (begin ,@body)))
           ,@(map (lambda (x old-x)
                    `(set! ,x ,old-x))
                  xx old-xx)
           ,result)))))
```

EGYÉB ADATSZERKEZETEK

Struktúrák

- Az alap Scheme-ben nincsenek struktúrák, de a **defstruct** makró segítségével definiálhatók, ami egy vektort és a hozzá tartozó kezelő eljárásokat hoz létre, pl.:

```
(defstruct address city street zip)
(define dummy-addr
    (make-address     'city 'Budapest
                      'street "Vidra u. 2."
                      'zip 1051))
(address.zip dummy-addr) ⇒ 1051
(set!address.zip dummy-addr 1052)
```

- A struktúra definíciójakor megadhatók alapértelmezett értékek, pl.:

```
(defstruct address (city 'Budapest) street zip)
```

Asszociációs listák

- Az asszociációs lista nem külön adattípus, csak a Lisp-ekben hagyományosan használt forma.
- Az asszociációs lista gyakorlatilag olyan párokból álló lista, ahol a párok első eleme a kulcs, a második pedig az adott kulcshoz tartozó érték: pl. ((a . 1) (b . 2) (c . 3))
- Az assv eljárás a megadott kulcshoz megkeresi a megfelelő párt, ehhez az eqv? predikátumot használja (az eqv? nem különbözteti meg a kis és nagy betűket, ezért ez a megoldás nem minden esetben használható).
- Az asszociációs listák segítségével bonyolultabb adatszerkezetek (pl. tömbök) is megvalósíthatók.

CONTINUATION

call-with-current-continuation

- A call-with-current-continuation (röviden call/cc) az argumentumát (aminek egy egy paraméteres eljárásnak kell lennie) meghívja a current-continuation-nak hívott környezetben, pl.:

```
(+ 1 (call/cc
    (lambda (k)
        (+ 2 (k 3))))) ⇒ 4
```

- A call/cc rendszerint kiugró (escaping) continuationra használható, de a környezet elmentésével bonyolultabb konstrukciókra is alkalmas.

Kiugró continuation

```
(define list-product
  (lambda (s)
    (let recur ((s s))
      (if (null? s) 1
        (* (car s) (recur (cdr s)))))))

(define list-product/cc
  (lambda (s)
   (call/cc  (lambda (exit)
     (let recur ((s s))
       (if (null? s) 1
         (if (= (car s) 0) (exit 0)
           (* (car s) (recur (cdr s)))))))))
```

Korutinok

- A call-with-current-continuation használatával bonyolultabb vezérlési konstrukció is előállítható, ami gyakorlatilag a korutionok mintájára működik:

```
(define-macro coroutine
   (lambda (x . body)
      `(letrec    ((local-control-state
                        (lambda (,x) ,@body))
                   (resume
                      (lambda (c v)
                         (call/cc
                            (lambda (k)
                               (set! local-control-state k)
                               (c v))))))
          (lambda (v)
             (local-control-state v)))))
```

OBJEKTUM-ORINETÁLT PROGRAMOZÁS

Egyszerű objektum rendszer

- Az objektum rendszer alapja az osztály (class), ami gyakorlatilag objektum típusnak tekinthetünk (azonos tulajdonságú és viselkedésű objektumok halmaza)
- Egy adott objektumot a megfelelő osztály egyedének (instance) nevezünk.
- Az osztályok hierarchikus rendszert alkotnak, amit örökléssel valósítunk meg (a példa implementáció egyszeres öröklést mutat be, később lesz szó arról, hogy miképp valósítható meg a többszörös).

Meta-osztályok

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots
                'superclass
                'method-names
                'method-vector)
          #t
          '(make-instance)
          (vector make-instance)))

(vector-set! standard-class 0 standard-class)

(and (vector? x) (eqv? (vector-ref x 0) standard-class))
(vector-ref c 1)  ; (standard-class.slots c) helyett
(vector-ref c 2)
(vector-ref c 3)
(vector-ref c 4)
(send 'make-instance standard-class ...)
```

Többszörös öröklés - 1

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots 'class-precedence-list
                'method-names 'method-vector)
          '()
          '(make-instance)  (vector make-instance)))

(define-macro create-class
  (lambda (direct-superclasses slots . methods)
    `(create-class-proc
      (list ,@(map (lambda (su) `,su) direct-superclasses))
      (list ,@(map (lambda (slot) `',slot) slots))
      (list ,@(map (lambda (method) `',(car method))
                   methods))
      (vector ,@(map (lambda (method) `,(cadr method))
                     methods)))))
```

Többszörös öröklés - 2

```
(define create-class-proc
  (lambda (direct-superclasses slots
           method-names method-vector)
    (let ((class-precedence-list
           (delete-duplicates
             (append-map
               (lambda (c) (vector-ref c 2))
               direct-superclasses))))
      (send 'make-instance standard-class
            'class-precedence-list
              class-precedence-list
            'slots
            (delete-duplicates
              (append slots (append-map
                              (lambda (c) (vector-ref c 1))
                            class-precedence-list)))
            'method-names method-names
            'method-vector method-vector))))
```

Többszörös öröklés - 4

```
(define send
  (lambda (method-name instance . args)
    (let ((proc
           (let ((class (class-of instance)))
             (if (eqv? class #t) (error 'send)
                 (let loop ((class class)
                            (superclasses
                              (vector-ref class 2)))
                   (let ((k (list-position
                              method-name
                              (vector-ref class 3))))
                     (cond (k (vector-ref
                                (vector-ref class 4) k))
                           ((null? superclasses)
                             (error 'send))
                           (else (loop (car superclasses)
                                       (cdr superclasses))))
                     ))))))
      (apply proc instance args))))
```

Többszörös öröklés - 4

```
(define append-map
  (lambda (f s)
    (let loop ((s s))
      (if (null? s) '()
          (append (f (car s))
                  (loop (cdr s)))))))
```

NEM-DETERMINISZTIKUS VÉGREHAJTÁS

Az amb operátor

- Az amb nem-determinisztikus (ambiguous) operátor egykorú a Lisp-pel (bár nem része annak).
- Az amb nulla vagy több paramétert vár, és nem-determinisztikusan választ egyet közülük.
- Pl. (amb 1 2) $\Rightarrow$ 1 vagy 2
  (amb) $\Rightarrow$ ERROR!!!
- Program környezetben az amb operátornak úgy kell működnie, hogy amennyiben lehet a program konvergáljon.
- Pl. (amb 1 (amb)) $\Rightarrow$ 1 (mindig)
  (amb (amb) 1) $\Rightarrow$ 1 (mindig)
  (if (amb #t #f) 1 (amb)) $\Rightarrow$ 1 (az amb a #t-t választja minden esetben)