A Scheme működése

- A Scheme mohó nyelv, de mit is jelent ez?
  - a kifejezések kiértékelésénél
    - 1. kiértékeli az összes részkifejezést
    - 2. alkalmazza a legbaloldalibb részkifejezés eredményét (eljárást) a többi részkifejezés eredményére
  - a múlt órai példa:
    - `((if #f = *) 3 4)` ⇒ `12`
- Oké, de, hogy áll le a rekurzió, ha a részkifejezéseket értékeljük ki hamarabb?
- Vannak az eljárások és vannak a speciális formák, mi írhatunk eljárást, vagy a későbbiekben láthatjuk, hogy makrókat is, amik lecserélik a programszöveget valami másra. A speciális formák a nyelvbe beépítettek. Csak az eljárások kiértékelése mohó. Így speciális formák (pl. if) használatával megállítható a rekurzió.
- ```
  (define (my-if-proc Felt KifI KifH)
            (cond (Felt KifI)(else KifH)))
  ```
- `(define (p) ((p)(p)))`
- `(if #t 0 (p))` ⇒ `0`
- `(my-if-proc #t 0 (p))`
  ⇒ `Aborting!: maximum recursion depth exceeded`
- látható, hogy a `my-if-proc` eljárás volta miatt a `(p)` hívás ki lett értékelve a mohó kiértékelés miatt
  ```

List kontra cons*

- `(list` *`object .. object`*`)` olyan listát hoz létre, aminek a vége az üres lista
- `(cons*` *`object .. object`*`)` olyan listát hoz létre, aminek utolsó eleme az utolsó két elemből képzett pár, ezért:
  - utolsó elemnek listának kell lennie
  - ha csak egy argumentummal hívjuk meg, akkor azt adja vissza
- mire használjuk?
- bár a Schemeben az utasítások is adatok, a végrehajtást mégsem tudjuk ráugrasztani egy listára, (aki próbálkozott akárhány argumentumú `compose`-t csinálni az tudja)
- erre való az `apply` *`procedure object .. object`*
  - végrehajtja a `procedure`-t a `(cons*` *`object .. object`*`)` argumentum(okk)al
- így a `compose` úgy néz ki, hogy
  - ```
    (define (compose f g)
            (lambda args (f (apply g args))))
    ```
    ```

Scope 1

- A Scheme lexikus kötést használ (azaz a változók nevét fordítási időben oldja fel), hasonlóan a Common Lisphez és ellentétben a lisp család többi tagjával
- Lokális változókötésre használhatóak a `let`, `let*`, `letrec`, és a `fluid-let`
  - a `let`, `let*`, `letrec` lexikus kötést valósít meg
  - a `fluid-let` azonban dinamikus kötést, ekkor a már létező változó értékét írjuk át
- `(define x 35)`
- `(define (p y) (+ x y))`
- `(let ((x 3)) (p 2))` ⇒ `37`
- `(fluid-let ((x 3)) (p 2))` ⇒ `5`
- a sima `let` értéke azért lett 37, mert a `p`-ben használt `x` a kinti scopeban van, a `let`en belüli értékadás pedig a benti scopera vonatkozik
- a `fluid-let` értéke ezzel szemben azért 5, mert a `fluid-let` a már létező változó értékét írja át ideiglenesen

Scope 2

- a `let` egyenes következménye a mohóságnak és a `lambda`nak:
  - a `(let ((`v_1 e_1`)..(`v_n e_n`)) törzs)` egyenértékű a
  - `((lambda (`v_1 `..` v_n`) törzs)` e_1 `..` e_n`)` hívással
- ebből már az is látszik miért a külső scopeban értékelődnek ki a változók a let esetében, hiszen a mohó kiértékelés szellemében először kiértékelődnek a részkifejezések, tehát a `lambda` (ami egy eljárást ad vissza), valamint az értékek – még az eredeti scopeban – majd az eljárás hívása új scopeot hoz létre az értékadással
- mivel a `let` esetében nincs definiálva, hogy milyen sorrendben értékelődnek ki az értékek, nem ha az új változók egymásra hivatkoznak, erre kínál megoldást a `let*`
  - a `(let* ((`v_1 e_1`)..(`v_n e_n`)) törzs)` egyenértékű a
  - `(let ((`v_1 e_1`))..(let ((`v_n e_n`)) törzs)..)` hívással, vagyis egymásba ágyazott `let`ekkel

Scope 3

- ezzel a szerkezettel ugyan már lehet egymásra hivatkozó változókat bevezetni, de még mindig nem lehet kezelni a rekurziót (mert nincs olyan két változó, ami kölcsönösen látná egymást), erre a problémára kínál megoldást a letrec bevezetése, ami már az új változókkal kibővített scopeban értékeli ki azokat
- `(letrec ((változó`$_1$ `init`$_1$`)..(változó`$_n$ `init`$_n$`)) törzs)`
- így például a:
  - `(letrec (`
    `(my-even? (lambda (n)`
    `(if (= 0 n) #t (my-odd? (- n 1)))))`
    `(my-odd? (lambda (n)`
    `(if (= 0 n) #f (my-even? (- n 1))))))`
    `(my-even? 88))` ⇒ `#t`
  - míg ugyanez `let`tel `unbound variable` hibát adott volna
- ugyanakkor egy fontos megkötés a `letrec`-kel szemben, hogy minden `init`-nek kiértékelhetőnek kell lennie egyetlen `változó` értékére való hivatkozás nélkül, mivel a `változó`knak még nincs értékük és a Scheme az argumentumokat érték szerint adja át
- ez a megkötés automatikusan teljesül, amennyiben `lambda` vagy `delay` kifejezéseket adunk meg (legtöbbször ez a helyzet)

Makrók 1

- makrókat a `(define-syntax` *`név t-spec`*`)` hívással lehet létrehozni, ahol *`név`* értelemszerűen a létrehozott makró neve – a fordító értékeli ki, kiértékelése után, bármely előfordulását (akár fordítási időben, akár futási időben találkozik vele) a Scheme kicseréli a *`t-spec`*-ben definiált átírásra.
- fontos, hogy a belül hivatkozott változókat név szerint adja át és nem érték szerint, ahogy általában teszi
- például a már látott my-if definíciója, ami eljárás formában hibára vezetett makrókkal úgy néz ki, hogy:
  - ```
    (define-syntax my-if-macro
          (lambda (felt i h)
               (list 'cond
                    (list felt i)(list 'else h))))
    ```
- ennek hatására, ha a előfordul egy `(my-if-macro f` e_1 e_2`)` hívás, akkor annak a helyébe a `(cond (f` e_1`) (else` e_2`))` hívást írja
- a már látott példa a `my-if-macro`-val helyesen fut le:
  - `(my-if-macro #t 0 (p))` $\Rightarrow$ `0`
- ez azért lehetséges, mert a define-syntax név szerint adja át a változókat
- példa a már látott apply-ra
  - ```
    (define-syntax my-apply (lambda
          (proc #!rest args)(cons proc args)))
    ```

Makrók 2

- ahhoz, hogy a makrók használható utasításokat adjanak vissza, azokat fel kell építeni a listákat felépítő eljárásokkal, ami fárasztó és sok esetben átláthatatlanná teszi a kódot
- ennek a kikerülésére vezették be a template-eket
  - `` ` `` (backtick) jelöli a template kezdetét
  - `,` (vessző) jelöl egy változóbehelyettesítést
  - `,@` (vesszőkukac) pedig egy listát illeszt be határoló zárójelek nélkül
- a my-if-macro felírása template-ekkel:

```
(define-syntax my-if-macro-template
      (lambda (felt i h)
            `(cond (,felt ,i) (else ,h))))
```

- ha olyan szerkezetet kell létrehoznunk, amiben szükségünk van segédváltozó használatára, akkor ez problémát okozhat, hiszen a segédváltozó elfedhet egy a kinti scopeban létező (a létrehozásakor új scope is létrejön) változót
- ennek a megkerülésére használhatjuk az egy eddig még garantáltan nem használt változónevet visszaadó `(generate-uninterned-symbol)` eljárást

Makrók 3

- nézzük meg a fluid-let átírását makrókkal:

```
(define-syntax my-fluid-let
    (lambda (xexe #!rest törzs)
      (let ((xx (map car xexe))
            (ee (map cadr xexe))
            (old-xx (map (lambda (_) (generate-
                                uninterned-symbol)) xexe))
            (result (generate-uninterned-symbol)))
 `(let ,(map (lambda (old-x x) `(,old-x ,x)) old-xx xx)
,@(map (lambda (x e) `(set! ,x ,e)) xx ee)
 (let ((,result (begin ,@törzs)))
 ,@(map (lambda (x old-x) `(set! ,x ,old-x)) xx old-xx)
,result) ) ) ) )
```

- kövessük végig mit is csinál ez
  - `my-fluid-let` egy makró
  - első argumentuma `xexe` a többi pedig `törzs`
  - `xx`-be bekerülnek az `xexe`-ben található párok fejei
  - `ee`-be pedig az `xexe`-ben található párok második elemei (azért `cadr`, mert lista formájában állnak a párok, vagyis alakjuk `(cons első (cons masodik ()))`)

Makrók 4

        - az értékadás név szerint történik, hiszen egy `define-syntax`on belül vagyunk, vagyis `xx`-ben a változók nevei szerepelnek, `ee`-ben pedig a kifejezések, amiknek a kiértékeléséből kapjuk a változók új értékét
    - `old-xx` egy lista lesz, ami az `xexe` listában szereplő minden párra egy még eddig nem használt szimbólumot tartalmaz
    - `result` pedig egy még eddig nem használt szimbólum
        - nem baj, ha bármelyik név szerepel xexe-ben, hiszen ezeket a változókat csak az átírásnál használjuk, a végeredményben már csak `uninterned-symbol`ok szerepelnek majd
    - ezután elkezdjük a helyettesítést a ` jellel
    - „kiírunk” egy `let`et, majd behelyettesítjük egy `map` eredményét, vagyis egy párokat tartalmazó listát, amiben a párok első eleme a generált változónevekből, a párok második eleme pedig az elmentendő változónevekből áll (ezt lista formátumban kell kiadnunk)
    - most jön a régi változók értékének megváltoztatása, ez már nem lista, hanem szekvencia kell legyen, hiszen ez a `let` törzse, ezért a `,@`-cal lehámozzuk a `map` eredményéről a határoló zárójeleket

Makrók 5

- majd egy beágyazott `let`ben kiértékeljük a `törzs`et, majd a `let` törzsében kiadjuk a változók értékeit visszaállító utasításokat, és a végére kiadjuk a kiértékelt `törzs` értékét tartalmazó változót
- a `(my-fluid-let ((x (+ 2 3))(y 5)) (a x (b y)))` hívás helyébe az íródna, hogy:
-
```
(let (
      (#[uninterned-symbol 1] x)
      (#[uninterned-symbol 2] y)
      )
      (set! x (+ 2 3))
      (set! y 5)
      (let (
      (#[uninterned-symbol 3] (begin a x (b y))
            (set! x #[uninterned-symbol 1])
            (set! y #[uninterned-symbol 2])
      #[uninterned-symbol 3])))))
```

Kiugrás 1

- `call-with-current-continuation` *`eljárás`*
- *`eljárás`* egy egyargumentumú eljárás kell legyen
- `call-with-current-continuation` meghívásakor az éppen aktuális környezetben kiértékelt ún. menekülő eljárással meghívja *`eljárás`*t
- a menekülő eljárás egy egyargumentumú eljárás, amit, ha meghívunk, akkor oda tér vissza, ahol létrehozták (vagyis az őt létrehozó `call-with-current-continuation`ből tér vissza), visszatérési értéke pedig az argumentuma
- `(+ 2 (call-with-current-continuation (lambda (exit) (+ 2 (exit 4))))) ⇒ 6`
- `(+ 2 (call-with-current-continuation (lambda (exit) (+ 2 4)))) ⇒ 8`
- ha végigkövetjük a futást akkor az első esetben azt kapjuk, hogy az `exit` kiértékelésekor a `call-with-current-continuation` visszatér az `exit`nek átadott értékkel (tulajdonképpen az `exit` tér vissza)
- a második esetben pedig minden rend szerint kiértékelődik és a `call-with-current-continuation` úgy tér vissza, mintha egy sima eljáráshívás lenne

Kiugrás 2

- a menekülő eljárás egy teljesen átlagos eljárásnak számít, mindent megtehetünk vele, amit egy eljárással, például el is menthetjük
- `(define exit)`
- `(+ 3 (call-with-current-continuation (lambda (x) (set! exit x) 0))) ⇒ 3`
- `(exit 5) ⇒ 8`

Struktúrák 1

- `(define-structure` *`(név struktúra-opció …) slot-leíró`* `…)`
- az utasítás bevezeti:
  - a típusleírót: a struktúra neve kielégíti a `record-type?`-ot
  - a konstruktort: `make-`*`név`* `slot-értékek`
  - egy típusellenőrző predikátumot: *`név`*`?`
  - minden slothoz egy „elérőt”: *`név`*`-slotneve`, ami egy struktúra típusú argumentumot vár, és visszaadja a `slotneve` nevű slot értékét
  - valamint minden slothoz egy módosítót: `set-`*`név`*`-slotneve!`, ami két argumentumot vár, az első egy struktúra típusú objektum, a második pedig a slot értéke
- a slot-leíró lehet:
  - `slotneve`
  - `(slotneve alapérték [slot-opció érték]`*`)`
- két létező slot-opció van, és ebből egyet használnak a Schemeben:
  - `read-only érték`
  - ha `érték` nem `#f`, akkor a slothoz nem generálódik módosító

Struktúrák 2

- struktúra-opció nagyon sok van:
  - `predicate` *`név`*: a típusellenőrző prédikátum nevét lehet bizgatni vele, ha `#f` akkor nem generál ilyet
  - `copier` *`név`*: létrehoz egy klónozó eljárást, *`név`* névvel, ha *`név`* nincs megadva, akkor `copy-struktúraneve` lesz a klónozóeljárás neve, ha *`név`* értéke `#f` akkor nem csinál ilyet
  - `print-procedure` *`kifejezés`*: ahol *`kifejezés`* eredménye egy kétargumentumú ún. unparser method kell legyen
  - `constructor [`*`név`* `[`*`argumentumlista`*`]]`: *`név`* név alatt létrehoz egy kontruktort, ha megadtunk argumentumlistát, akkor ilyen sorrendben tölti majd fel a slotokat az értékekkel
  - `keyword-constructor [`*`név`*`]`: létrehoz *`név`* néven egy konstruktort, ami úgy hívható, hogy minden páratlanadik számú argumentuma slotnév kell legyen, és az ezt követő kifejezés lesz annak a slotnak az értéke
  - …