Expression := proc ({identifier}*(,)) Expression → proc-exp(lst body)

:= (Expression Expression)

Call-exp(rator rand-list)

Expval = toInt + toBool + toProc Expval سه نوع دارد:

کلا در این زبان یا atom داریم یا لیست. پس اگر تعداد تهی atom را بگوییم: تابع پاس بدهیم باید همه‌ی آنها را در یک لیست نگه داریم و آن لیست را به تابع پاس بدهیم و Signature تابع هم یک لیست دودی بگیرد. (مثل زبان C)

اول: تغییر the-grammar که کلیه قاعده‌ها را داردیک مورد اضافه می‌کنیم:

```
(define the-grammar
  ( [program (expression) a-program]

    کلیه قاعده‌های قبلی
    ;

    [expression ("proc" "(" (separated-list identifier ",")
                ")" expression) proc-exp] ))
```

سپس تایپ proc را تعریف می‌کنیم

```
(define-datatype proc proc?
  [procedure [vars (list-of symbol?)]
             [body expression?]
             [env environment?]]
  [built-in-procedure [name symbol?]])
```

و همانطور که گفتیم یک نوع جدید: expval اضافه شده است

```
(define-datatype expval expval?
  [num-val [value number?]]
  [bool-val [boolean boolean?]]
  [proc-val [proc proc?]])
```

```scheme
(define expval->proc
  (lambda (v)
    (cases expval v)
      [proc-val (proc) proc]
      [else (eopl:error 'expval-extractors
              "looking for a ~s, found ~s"
            'proc   v)]
```

اینم هم برای cast کردن type به proc.

برای اینکه بشود لیست را ورودی گرفت، باید دو خط را به تابع value-of اضافه کنیم:

```scheme
(define value-of
  (lambda (exp env)          ; اینا قبلا هم بودن
    (cases-exp exp
      [const-exp (num) (num-val num)
      .
      .
      .     ; (انواع exp ها)

      ; جدید:
      [ proc-exp (vars body) (proc-val (procedure
                                 vars body env))]

      [call-exp (rator rands) (let ([proc (expval->proc
                                      (value-of rator env))]
           [args (map (lambda (rand)
                          (value-of rand env)) rands)])
           (apply-procedure proc args))]
```

: apply-procedure 를 고쳐

```
(define apply-procedure
  (lambda (proc1 args)
    (cases proc proc1
      [procedure (vars body saved-env)
        (let loop ([env saved-env]
                   [vars vars]
                   [args args])
          (if (null? vars)
              (value-of body env)
              (loop (extend-env (car vars) (car args) env)
                    (cdr vars)
                    (cdr args)
                    )))])))
```

3.23 . می‌بینیم که این تابع (یعنی proc(x)) اگر x صفر باشد، 0 برمی‌گرداند و اگر نه، جمع 4 با proc(x-1) را برمی‌گرداند (یعنی 4- را از آن کم می‌کند). یعنی برای عدد n، P_n را چنان پیدا می‌کنیم که:

$$P_0 = 0 \quad , \quad P_n = P_{n-1} - (-4)$$

که به‌ی عددی می‌شود $P_n = 4n$. یعنی این تابع با x بار جمع کردن 4، $4x$ را برای ما می‌سازد.

حال باید $|x|$ را که با ضرب کردن x عدد در یکدیگر (اعداد 1 تا x) ایجاد می‌شود بدین صورت تولید کنیم که به جای -، * قرار دهیم و به جای 4-، x، 1.

برای انجام عمل ضرب، تابع makemult را می‌نویسیم

```
let makemult = proc(maker)
                 proc(x)
                   proc(y)
                     if zero?(x)
                       then 0
                       else -((((maker maker) -(x,1)) y)
                              , -(0,y))
in let mult = (makemult makemult)
   in let makefact = proc(maker)
                       proc(x)
                         if zero?(x)
                           then 1
                           else ((mult x) ((maker maker)
                                           -(x,1)))
      in (makefact makefact)
)
```

3.31) اپ apply-env را تغییر دهیم که وقتی که تابعی call می‌شود آن را پیدا کند:

```
(define apply-env
  (lambda (env search-sym)
    (cases environment env
      [empty-env      مثل قبل ]
      [extend-env     مثل قبل ]
      [extend-env-rec (p-names b-vars p-bodies
                       saved-env)
        (let loop ([p-names p-names]
                   [b-vars b-vars]
                   [p-bodies p-bodies])
          (if (null? p-names)
              (apply-env saved-env search-sym)
              (if (eqv? search-sym (car p-names))
                  (proc-val (procedure (car b-vars)
                                       (car p-bodies)
                                       env))
                  (loop (cdr p-names)
                        (cdr b-vars)
                        (cdr p-bodies)))))])))
```

جدید برای تابع بازگشتی

ابتدا نوع داده‌ی environment را این‌گونه تعریف می‌کنیم:

```
(define-datatype environment environment?
  [empty-env  مثل قبل]
  [extend-env  مثل قبل]
  [extend-env-rec  [ids (list-of symbol?)]
                   [bvars (list-of (list-of symbol?))]
                   [bodies (list-of expression?)]
                   [saved-env environment?]])
```

حال تابع value-of را با این خط ادامه می‌دهیم:

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      [const-exp  مثل قبل]
      .
      .
      .
```

جدید ↓

```
      [letrec-exp (p-names b-vars p-bodies letrect-body)
        (value-of letrec-body (extend-env-rec
                                p-names b-vars p-bodies env))]
```

(4.17 این قانون را اضافه می‌کنیم:

(apply-procedure (procedure (list var1 var2 ... var n) body ρ)
(list val1 val2 ... val n) σ)

= (value-of body [var$_n$ = ℓ_n] ... [var$_2$ = ℓ_2] [var1 = ℓ_1] ρ
[ℓ_n = val$_n$] ... [ℓ_2 = val$_2$] [ℓ_1 = val$_1$] σ)

(value-of exp_1 ρ σ_0) = (val_1, σ_1)

(value-of exp_2 ρ σ_1) = (val_2, σ_2)

...

(value-of exp_n ρ σ_{n-1}) = (val_n, σ_n)

(value-of ((let-exp (list var1 var2 ... ,var$_n$)
(list exp1 exp2 ... expn) body) ρ σ_0)
= (value-of body [var$_n$ = ℓ_n] ... [var$_2$ = ℓ_2] [var$_1$ = ℓ_1] ρ
[ℓ_n = val$_n$] ... [ℓ_2 = val$_2$] [ℓ_1 = val$_1$] σ_n)

(4.40

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      [const-exp (num) (newref (num-val num))]
      [var-exp (var) (apply-env env var)]
      [proc-exp (var body) (newref
                             (proc-val (procedure var body env)))]
      [else (newref (a-thunk exp env))])))
```