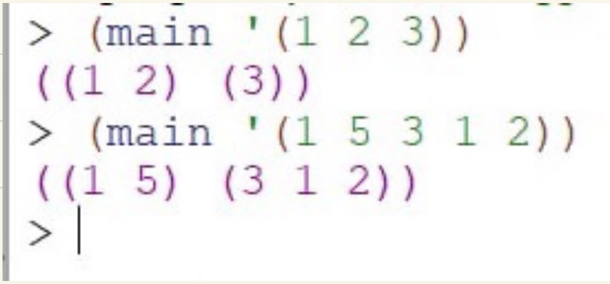
ارشیا مقیمی

1

```
#lang eopl

(define (sum-rest L)
  (cond
    [(null? L) 0]
    [else (+ (car L) (sum-rest (cdr L)))]
    )
  )

(define (divider L sumL counter)
  (cond
    [(null? L) "impossible"]
    [(equal? (+ (car L) sumL) (sum-rest (cdr L))) counter]
    [else (divider (cdr L) (+ (car L) sumL) (+ 1 counter))]
    )
  )

(define (get-n-items lst num)
  (if (> num 0)
      (cons (car lst) (get-n-items (cdr lst) (- num 1)))
      '()
      )
  )

(define (length L)
  (cond
    [(null? L) 0]
    [else (+ 1 (length (cdr  L)))]
    )
  )

(define (printer L n)
  (display (list (get-n-items L (+ n 1)) (reverse (get-n-items (reverse L)
(- (length L) (+ n 1))))))
  )

(define (main L)
  (printer L (divider L 0 0))
  )
```



```
> (main '(1 2 3))
((1 2) (3))
> (main '(1 5 3 1 2))
((1 5) (3 1 2))
> 
```

نتیجه اجرا ←

۲

الف) قوانین استنتاج :

$(0,1,5) \in S$

$$\frac{\overset{f(n)\ \ f(n+1)+2f(n)}{(n,m,k) \in S}}{(n+1,\ k-2m,\ k+2m) \in S}$$

پایین به بالا : مجموعهٔ S را کوچکترین مجموعه روی N تعریف می‌کنیم که :

۱) $(0,1,5) \in S$ و

۲) اگر $(n,m,k) \in S$ آنگاه $(n+1, k-2m, k+2m) \in S$

بالا به پایین : سه‌تایی (n,m,k) عصو مجموعهٔ S هستند اگر وتنها اگر:

۱) $(n,m,k) = (0,1,5)$ یا

۲) $(n-1, \frac{k-3m}{2}, k-2m) \in S$

ب) $S=\{(1,3),(3,4),(5,7),(7,12),\dots\}=\{(n,f(n)) \mid n=2k+1, k\in N, f(0)=3, f(n)=n+f(n-2)\}$

$S=\{(0,5),(2,16),(4,49),\dots\}=\{(n,f(n)) \mid n=2k, k\in N, f(0)=5, f(n)=3f(n-2)+1\}$

۳

الف) روش داده ساختاری:

```
(define (report-no-variable-found)
   'not-set-variable
  )

(define (apply-env env search-var)
  (let loop ([env1 env])
    (cond
      [(eqv? (car env1) 'empty-env)
       (report-no-variable-found)]
      [(eqv? (car env1) 'extend-env)
       (let ([saved-var (cadr env1)]
             [saved-val (caddr env1)]
             [saved-env (cadddr env1)])
         (if (eqv? search-var)
             saved-var saved-val
             (loop saved-env)))]
      )
    )
  )
```

* تنها قسمت مورد نظر آورده شده.

کد کامل در صفحات بعدی قرار دارد.

روش تابعی:

```
(define report-no-variable-found
  (lambda ()
    'not-set-variable
    )
  )

(define empty-env
  (lambda ()
    (list (lambda (search-var) (report-no-variable-found))
          (lambda () #t)
          (lambda (search-var) #f)
      )
    )
  )
```

ب) تابع has-binding?

روش تابعی:

```
(define has-binding?
  (lambda (env search-var)
    ((caddr env) search-var)
    )
  )
```

روش داده ساختاری.

```
(define (has-binding? env search-var)
  (cond [(null? env) #f]
        [(eqv? (caar env) search-var) #t]
        [else (has-binding? (cdr env) search-var)]
        )
  )
```

ج) تابع union .

روش تابعی:

```
(define union
  (lambda (env1 env2)
    (list (lambda (search-var)
              (if (has-binding? env2 search-var)
                  (apply-env env2 search-var)
                  (apply-env env1 search-var)))
          (lambda () #f)
          (lambda (search-var)
            (or
             (has-binding? env1 search-var)
             (has-binding? env2 search-var)))
          )
    )
  )
```

کد کامل environment به روش تابعی.

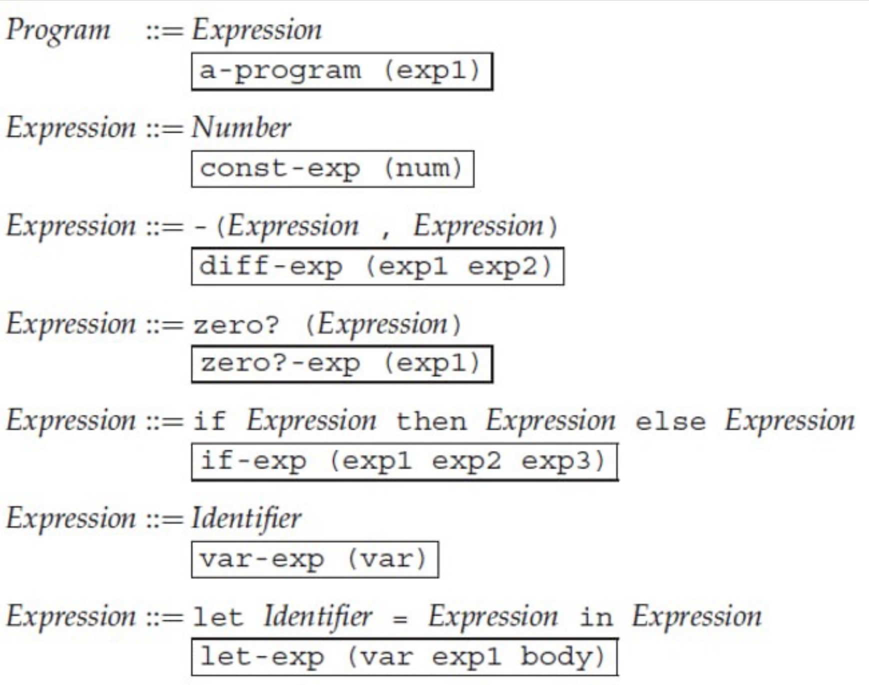
```scheme
(define report-no-variable-found
  (lambda ()
    'not-set-variable
    )
  )

(define empty-env
  (lambda ()
    (list (lambda (search-var) (report-no-variable-found))
          (lambda () #t)
          (lambda (search-var) #f)
      )
    )
  )

(define empty-env?
  (lambda (env)
    ((cadr env))
    )
  )

(define extend-env
  (lambda (var val env)
    (list (lambda (search-var)
            (if (eqv? search-var var)
                val
                (apply-env env search-var)))
          (lambda () #f)
          (lambda (search-var)
            (or
             (eqv? var search-var)
             (has-binding? env search-var)))
          )
    )
  )
```

```
(define union
  (lambda (env1 env2)
    (list (lambda (search-var)
            (if (has-binding? env2 search-var)
                (apply-env env2 search-var)
                (apply-env env1 search-var)))
          (lambda () #f)
          (lambda (search-var)
            (or
             (has-binding? env1 search-var)
             (has-binding? env2 search-var)))
          )
    )
  )

(define apply-env
  (lambda (env search-var)
    ((car env) search-var)
    )
  )

(define has-binding?
  (lambda (env search-var)
    ((caddr env) search-var)
    )
  )
```

کد کامل environment ب روش داده ساختاری:

```
(define (empty-env)
   '('empty-env)
  )

(define (extend-env var val env)
    (list 'extend-env var val env)
  )


(define (report-no-variable-found)
  'not-set-variable
  )

(define (apply-env env search-var)
  (let loop ([env1 env])
    (cond
      [(eqv? (car env1) 'empty-env)
       (report-no-variable-found search-var env)]
      [(eqv? (car env1) 'extend-env)
       (let ([saved-var (cadr env1)]
             [saved-val (caddr env1)]
             [saved-env (cadddr env1)])
         (if (eqv? search-var saved-var)
             saved-val
             (loop saved-env)))]
      )
    )
  )

(define (has-binding? env search-var)
  (cond
    [(null? env) #f]
    [(eqv? (caar env) search-var) #t]
    [else (has-binding? (cdr env) search-var)]
    )
  )
```

۲



Program ::= *Expression*

`a-program (exp1)`

Expression ::= *Number*

`const-exp (num)`

Expression ::= - (*Expression* , *Expression*)

`diff-exp (exp1 exp2)`

Expression ::= zero? (*Expression*)

`zero?-exp (exp1)`

Expression ::= if *Expression* then *Expression* else *Expression*

`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*

`var-exp (var)`

Expression ::= let *Identifier* = *Expression* in *Expression*

`let-exp (var exp1 body)`

Expression ::= let Identifier = String in Expression

let-str (var str body)

String ::= " (char)* "

str-exp (str)

Expression ::= null? (String)

null?-exp (str)

Expression ::= in? (String String)

in-exp? (str1 str2) ← آیا str1 در str2 هست؟

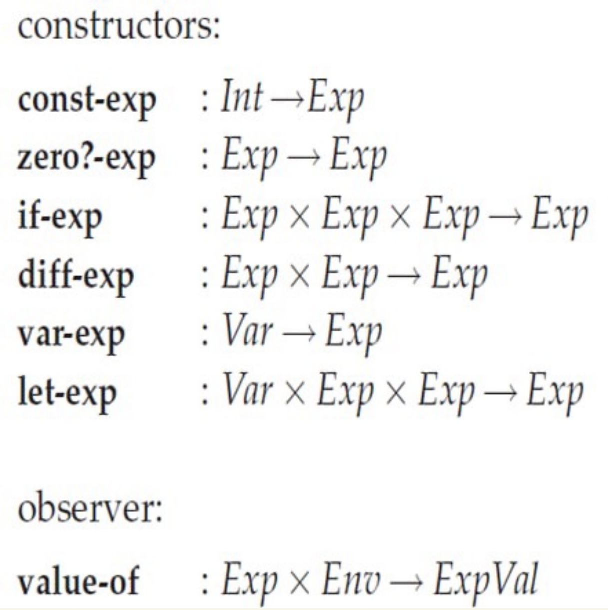
- علاوه بر ExpVal و DenVal حالا نیاز به یک StrVal هم داریم:

StrVal = String

str-val : String → StrVal

strval→string : StrVal → String

- محیط نیازی به تغییر ندارد



constructors:

const-exp : *Int → Exp*

zero?-exp : *Exp → Exp*

if-exp : *Exp × Exp × Exp → Exp*

diff-exp : *Exp × Exp → Exp*

var-exp : *Var → Exp*

let-exp : *Var × Exp × Exp → Exp*

observer:

value-of : *Exp × Env → ExpVal*

: Str × Env → StrVal

let-str : Var × Str × Exp → Exp

str-exp : Str → String

null?-exp : Str → Exp

in?-exp : Str × Str → Exp

```
(value-of (str-exp s) ρ) = (str-val s)
```

```
(value-of (let-str var str body) ρ) = (value-of body [var=(value-of str ρ)]ρ)
```

```
(value-of (null?-exp str) ρ) = ( if (equal? (value-of str ρ) "")
                                    (bool-val #t)
                                    (bool-val #f))
```

```
(value-of (in?-exp str1 str2) ρ) = ( if (in? (string->list str1)
                                             (string->list str2))
                                        (bool-val #t)
                                        (bool-val #f))
```

```
(define (in-helper str1 str2)
    (cond
      [(null? str1) #t]
      [(null? str2) #f]
      [else (cond
          [(eqv? (car str1) (car str2))
                (in-helper (cdr str1) (cdr str2))]
        [else #f]])))
```

```
(define (in? str1 str2)
   (if (null? str2) #f
       (if (in-helper str1 str2)
           #t
           (in? str1 (cdr str2)))))
```