316 12-08-09

Note Title 12/8/2009

2-Branch Conditional Expressions: (if B Exp_1 Exp_2)

Boolean Expression — True Case — False Case

Eval(B, σ)

Structured operational semantics

if Eval(B) = $\perp_V$ then $\perp_V$

else if Eval(B) = true then Eval(Exp_1)

else Eval(Exp_2)

The state σ is implicitly given by the values of the function parameters

Multi-Branch Conditional Expressions: (cond (B_1 Exp_1) ... (B_n Exp_n))

B_i, $1 \le i \le n$ = Boolean Expression

Exp_i, $1 \le i \le n$ = Expression to be evaluated if $B_1 \Rightarrow$ true

keyword

Any B_i may be keyword else which always evaluates to true.

if Eval(B_1) = $\perp_V$ then $\perp_V$

else if Eval(B_1) = true, then Eval(Exp_1)

else if Eval(B_2) = $\perp_V$, then $\perp_V$

= true, then Eval(Exp_2)

⋮

else if Eval(B_n) = $\perp_V$ then $\perp_V$

= true, then Eval(Exp_n)

Function Definitions

(define (funName $p_1 \dots p_n$) Exp), $n \ge 0$

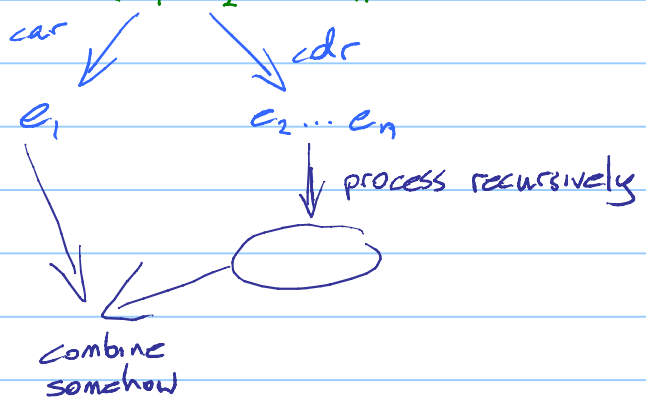
keyword — formal parameters — body expression

(define (square x) (* x x))

(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))

Recursive Processing of Lists

- Process bottom case - process lists of 0, 1, 2 elements
- Recursive case - $(e_1\ e_2 \ldots e_n)$




```
(define (length list) ; return the # of elements in list
(if (null? list) 0 ---- built-in function to see if list is ()
    (+ 1 (length (cdr list))) )
```

```
(length '(1 2 3)) ⇒
(+ 1 (length '(2 3))) ⇒
(+ 1 (+ 1 (length '(3)))) ⇒
(+ 1 (+ 1 (+ 1 (length '())))) ⇒
(+ 1 (+ 1 (+ 1 0))) ⇒
(+ 1 (+ 1 (+ 1 1))) ⇒
(+ 1 2) ⇒
3
```

$(e_1 \ldots e_n)(e_1' \ldots e_m') \Rightarrow (e_1 \ldots e_n\ e_1' \ldots e_m')$

```
(define (append list1 list2)
    (if (null? list1) list2
        (cons (car list1) (append (cdr list1) list2)
        )
    )
)
```

```
(append '(1 2) '(a b)) ⇒
(cons 1 (append '(2) '(a b))) ⇒
(cons 1 (cons 2 (append '() '(a b)))) ⇒
(cons 1 (cons 2 '(a b))) ⇒
(cons 1 '(2 a b)) ⇒
(1 2 a b)
```

In Lisp, all operators and user-defined functions can be passed as actual parameters of functions

```
(define (map f list) ; apply unary operator/function f to each element of list
   (if (null? list) ()
       (cons (f (car list)) (map f (cdr list)))
       )
   )
)
(map square '(1 2 3)) ⇒
(cons 1 (map square '(2 3))) ⇒
(cons 1 (cons 4 (map square '(3)))) ⇒
(cons 1 (cons 4 (cons 9 (map square '())))) ⇒
(cons 1 (cons 4 (cons 9 '()))) ⇒
(cons 1 (cons 4 '(9))) ⇒
(cons 1 '(4 9)) ⇒
(1 4 9)
```

```
(map length '(a b) (1 2 3) (c d f)) ⇒ (2 3 3)
(define (filter f list) ; f is a boolean unary function. drop from list all elements
                        for which f returns false
   (cond ((null? list) ())
         ((f (car list)) (cons (car list) (filter f (cdr list))))
          B2                  Exp2
```

(else (filter f (cdr list))) (filter number? '(a 3 4 5.6 #36 a 10.5e-5))

B_3 Exp_3 $\Rightarrow$ (3 4 5.6 10.5e-5)

)

)

(define (accumulate f list init-val) ; Given a list = $(x_1, \ldots, x_n)$ and a binary operator/function f,

; return $f(x_2, f(x_2, \ldots, f(x_n, \text{init-val}) \ldots))$

if (null? list) init-val

(f (car list) (accumulate f (cdr list) init-val))

)

)

(define (sum-list list)(accumulate + list 0)) (sum-list '(1 4 5)) $\Rightarrow 1+4+5=10$

(define (product-list list)(accumulate * list 1))

(define (append-list list)(accumulate append list '()))

(define (or-list list) (accumulate or list false))

(define (and-list list) (accumulate and list true))