1.2.1 Linear recursion and iteration

once factorial icin recursive bir prosedur yaziyor ve bunu yaparken

once factorial 6 dan basliyor, 6\*5\*4\*3\*2\*1 diye hesap yapiyor , daha sonra farkli bir yontem deniyor, diyor ki 1 den baslayip 6 ya gidelim ve bir counter im olsun nerede duracagimi bileyim diyor. 1\*2\*3\*4\*5\*6 yapan iterative bir prosedur yaziyor.

product = product\* counter oluyorken

counter = counter + 1 oluyor

recursive yapida sanki islem zincirlerini uc uca ekliyoruz ve islem zinciri surekli buyuyor. interpreter bu islemleri daha sonra yapmak uzere zincir olusturuyor.

STANDUP-1

\* lisp ilk ciktiginda sayisal hesaplama konusunda pek basarili degilmis. Algol bu konuda daha basarliymis. scheme ile lisp ve algol birlestirilmis. lisp in listeleri kontrol etme kabiliyeti cok yuksek. adi da list processing zaten.

\* Kıtapta hocaların amacı bılgısayar bılımlerıne yenı baslayacak kısılere programlamanın estetıgı hakkında guzel dusunceler kazandırmak, onlara complexity yi kontrol edebilmeleri icin temel teknikleri ogretmek ve bunu cok buyuk programlar yazarken yapabilmelerini saglamak.

\* complexity yi kontrol etmek icin nasil bir yontem kullaniyor? abstractions lar insa ederek. Yeni bir dil olusturarak complexity yi kontrol edebiliriz.

\* data giriyor bizim program datayi alip baska bir data ya donusturuyor, bunu yapmanin degisik yollari mevcut, iste bu yollara computational process diyoruz.

\* IDIOMS = Dogrulugundan emin oldugumuz standart program yapilarindan olusan bir depo

yani idioms larimiz varsa bunlari ogrenir ve gelistirirsek bunlari nasil biraraya getirebilecegimizi bilirsek kaliteli biri oluruz

\* lisp yapay zeka icin uretildi

\* bir data structure da 100 fonksiyonun calismasi, 10 data structure da 10 fonksiyonun calismasindan daha iyidir. ????

\* matematik what is? sorusuna cevap ararken computation how to? sorusuna cevap ariyor. yani factorial 5 nedir? 5\*4\*3\*2\*1 dir peki bunu nasil yazarim. matematik degismez ama computation degisebilir.

\* We can make procedural and data abstractions, we can use higher-order functions to capture common patterns of usage, we can model local state using assignment and data mutation, we can link parts of a program with streams and delayed evaluation, and we can easily implement embedded languages.

\* metalinguistic abstractions yeni bir olusturmak aslinda compextiy yi kontrol edebilmek icin.

\* lexical scoping ic ice fonksiyon tanimladiginda distaki fonksiyon calistiginda icteki de calisir diger turlu icteki fonksiyon calismaz, iste buna denir.

\* debug = ornegin programi yazdin ama yanlis calisiyor, o zaman debug yaparsin ve problemi cozersin.

\* pragramlada kulllanilan seyler, yani programlamanin elementleri 3 tanedir

1-primitive expressions ( semboller, sayilar..)

2-means of combinations (cond, if...)

3-means of abstractions (define, set ...)

\* interpreter yazdigimiz kodu cozumleyip bize sonucu veren yapi

\* evaluate yazdigimiz kodun cozumlenmesi

\* combinations parantez icinde gosterdiklerimiz aslinda, operator ve operand lardan olusan yapilar

\*prefix notation = operator u once yazmamiz

\* pretty printing = ifadeleri daha anlasilir olmasi icin alt alta yazabilme

\* kod yazarken saklama ihtiyai duddugumuz ifadeleri tutmak icin programin kullandigi hafizaya environment denir.

1.1.3 evaluating combinations

LISP DILININ OZELLIKLERI

\* Lisp, ilk fonksiyonel programlama dili,

\* Kullanimda en eski ve en guclu,

\* John McCarty --> Yapay zeka teriminin ve Lisp 'in mucidi (1958),

\* Bir donem MIT'de lisans öğrencilerine okutulan ilk programlama dersi,

\* Daha az sinirlayici syntax,

\* Basit ve esnek programlama yapisi,

\* Butun girdiler her yonuyle degistirilebilir yani dinamik bir yapida,

\* Değişken tipi tanımlamaya gerek duymaz.

LISP VERSIYONLARI

\* Scheme

\* Common-Lisp (Türevlerinin avantajlarının birleştirildiği ve dezavantajlarının

\* Emacs giderildiği bir Lisp versiyonudur.)

\* AutoLisp

\* ...

LISP YAPISAL OZELLIKLERI

----> primitive data (numbers,symbols)

\* Primitive expressions --|

----> primitive procedures (+,-,/,\*)

\* Means of combination (cond, if...)

\* Means of abstraction (define, let, set...)

\* (+ 19 20) -----> compound expressions

/ \ |

/ \ |

/ \|

operator operands

\* prefix notation

Examples;

(+ 1 2)

(\* 1 2 3 4 5)

(define (abs x)

(\* (+ 3 5) 2) (if (< x 0)

-x

(define x 5) x))

(\* x 5) (abs -3)

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

(square 5)

Booleans

(equal? 5 4) --> #f

(number? 5) --> #t

(symbol? 'i) --> #t

(symbol "i") --> #f

(not #t) --> #f

Conditionals

(if <p> <c> <a>) ----> (if (< 3 5) true false)

(cond (<p1> <e1>)

(<p2> <e2>) ----> (cond ((number? 1) deniz)

... ((symbol? 1) pinar))

(<pn> <en>))

Tree Accumulation

Example; (+ (- 5 3) (\* 2 3)) 8

/|\

/ | \

(+ 2 6) + 2 6

answer--> 8 /|\ \

/ | \ \

- 5 3 /|\

/ | \

\* 2 3

How does the Lisp program evaluate?

Example; (define (square x) (\* x x))

(define (sum-of-square x y)

(+ (square x) (square y)))

(define (f a)

(sum-of-square (+ a 1) (\* a 2)))

(f 5) =?

//Normal Order Evaluation\\ //Applicative Order Evaluation\\

(sum-of-square (+ 5 1) (\* 5 2)) (sum-of-square (+ 5 1) (\* 5 2))

(+ (square (+ 5 1)) (square (\* 5 2))) (sum-of-square 6 10)

(+ (\* (+ 5 1) (+ 5 1)) (\* (\* 5 2) (\* 5 2))) (+ (square 6) (square 10))

(+ (\* 6 6) (\* 10 10)) (+ 36 100)

136 136

(+ 36 100) 136

136

Example; (define (p) (p))

(define (test x y)

(if (= x 0)

0

y))

(test 0 (p)) = ?

\_\_\_\_\_\_\_\_\_\_\_\_\_/\\_\_\_\_\_\_\_\_\_\_\_\_

| |

applicative order ev. normal order ev.

(test 0 (p)) (test 0 (p))

(test 0 (p)) (test 0 (p))

... 0

error

BLOCK STRUCTURE ?

LEXICAL SCOPING ?

Recursion and Iteration

Example; (+ 3 4) = ?

/\

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| |

Recursive Iterative

(define (+ x y) (define (+ x y)

(if (= x 0) (if (= x 0)

y y

(+ 1 (+ (- x 1) y)))) (+ (- x 1) (+ y 1))))

Recursive Iterative

i r

(+ 3 4) (+ 3 4) asagiya dogru time x x

(+ 1 (+ 2 4)) (+ 2 5) saga dogru olan space 1 x

(+ 1 (+ 1 (+ 1 4))) (+ 1 6)

(+ 1 (+ 1 (+ 1 (+ 0 4)))) (+ 0 7)

(+ 1 (+ 1 (+ 1 4))) 7

(+ 1 (+ 1 5))

(+ 1 6)

7

Recursion and Iteration

Example; (fact 6) = ?

/\

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| |

Recursive Iterative

(define (fact n) (define (fact n)

(if (or (= n 1) (= n 0)) (fact-iter 1 1 n))

1 (define (fact-iter product counter max-count)

(\* n (fact (- n 1))))) (if (> counter max-count)

product

(fact-iter (\* counter max-count)

(+ counter 1)

max-count)))

Recursive Iterative

(fact 6) (fact 6)

(\* 6 (fact 5)) (fact-iter 1 1 6)

(\* 6 (\* 5 (fact 4))) (fact-iter 1 2 6)

(\* 6 (\* 5 (\* 4 (fact 3)))) (fact-iter 2 3 6)

(\* 6 (\* 5 (\* 4 (\* 3 (fact 2))))) (fact-iter 6 4 6)

(\* 6 (\* 5 (\* 4 (\* 3 (\* 2 (fact 1)))))) (fact-iter 24 5 6)

(\* 6 (\* 5 (\* 4 (\* 3 (\* 2 1))))) (fact-iter 120 6 6)

(\* 6 (\* 5 (\* 4 (\* 3 2)))) (fact-iter 720 7 6)

(\* 6 (\* 5 (\* 4 6))) 720

(\* 6 (\* 5 24))

(\* 6 120)

720

BIG 'O' NOTATION

SORU; Asagida tanimlanan fonksiyon recursive midir yoksa iterative midir?

(define (ebob a b)

(if (= b 0) --> Tail-Recursive

a

(ebob b (remainder a b))))

Higher Order Procedures

(define x 4)

(set! x 10)

(define (f x)

(let ((y 3))

(+ (\* 2 x) y)))

(define sum-of-squares (define (sum-of-squares x y)

(lambda (x y) (+ (square x) (square y)))) -----> (+ (square x) (square y)))

(begin (define x 10) x)

LISTS

(list 1 2 3 4 5) --> (1 2 3 4 5)

(define x (list 1 2 3 4 5))

x=?

--> (1 2 3 4 5)

(car x) --> 1

(cdr x) --> (2 3 4 5)

(cons 1 2) --> (1 . 2)

(cons 9 x) --> (9 1 2 3 4 5)

(define x (list 1 2 3 4 5))

(cons 9 x) --> (9 1 2 3 4 5)

(append (list 9) x) --> (9 1 2 3 4 5)

(list 9 x) --> (9 (1 2 3 4 5))

Bank Account

(define (make-account balance)

(define (withdraw amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))

(define (deposit amount)

(set! balance (+ balance amount))

balance)

(define (dispatch m)

(cond ((eq? m 'withdraw) withdraw)

((eq? m 'deposit) deposit)

(else (error "Unknown request -- MAKE-ACCOUNT"

m))))

dispatch)

(define account-deniz (make-account 100))

((account-deniz 'withdraw) 50)

((account-deniz 'deposit) 10)

LISP- SCHEME

(list "a" "b")

(list a b) ----> is not work

set ile bir counter olustursak ve tekrar tekrar bu counter i cagirsak her defasinda degisik cevaplar aliriz.

(list 4 5 6)

(length (list 4 5 6))

(define my-list (list 1 2 3 4 5))

my-list

(list "a" "b")

(define four-over-two (list '/ 4 2))

(eval four-over-two)

(sort (list 5 4 3 2 1) <)

(sort (list "abc" "bc" "a") string<?)

(cdr (cons 1 null))

(define my-list (list 1 2 3))

(equal? my-list (list 1 2 3))

(list-ref (list 1 2) 0)

(define (red a b)

(if (zero? a) b 0))

(red 0 1)

(define baz (list 1 2 3))

(define (double x) (\* x 2))

(map double baz)

(define qux (list 1 2 3 4))

(foldr + 0 qux)

(foldr \* 1 qux)

(display "a")

(newline)

(define (make-add-one)

(define (inc x)

(+ x 1))

inc)

(define mfyn (make-add-one))

(mfyn 2)

(begin (display "(") (display "a") (display 3))

(length x) = (length (cdr x)) + 1

(define x (list 1 2 3))

(define y (list 4 5 6))

(append x y) = (1 2 3 4 5 6)

(cons x y) = ((1 2 3) 4 5 6)

(list x y) = ((1 2 3) (4 5 6))

(list "a" "b") = a ve b birer string

(list 'a 'b) = a ve b birer sembol

eger bos bir liste tanimlamak istiyorsan = (list)

(define z 1) (symbol? z) = #f diye cevap verir. (number? z) = #t

(integer? z) = #t

(pair? z) = #f

(define square '(\* x x)) yani square i bir sembol olarak tanimladik. (eval square) dedigimiz zaman artik bu bir prosedure donusuyor ve islem yapabiliyoruz. andy balaam 5. dersin sonunda ornegi var.

(even? 2) = cift mi diye bakar.

(quotient 5 2) = 5 in 2 ile bolumunden kalani verir.

(sqr 10) = 100

(sqrt 100) = 10

(expt 2 3) = 8

(reverse (list 1 2 3)) = (3 2 1)

(random 5) = 0 ile 5 arasinda bir sayi atar.

(filter positive? '(1 -2 3 4 -5)) = (1 3 4)

(remove 2 (list 1 2 3 2 4)) = (1 3 2 4)

(string-ref "exp" 0) = 'e

(substring "exp" 1) = 'xp

(equal? (string-ref "exp" 0) '#\@) dogru yazilisi budur

(string-append "Apple" "Banana")

"AppleBanana"

(require racket/trace)

(define (add42 x)

(+ x 42))

(trace add42)

(define (assembler exp)

(+ exp 5)

(set! counter (+ 1 counter)))

(define new-withdraw

(let ((balance 100))

(lambda (amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))))

SAME --|YUKARIDAKI ILE

(define counter 0)

(define balance 100)

(define (withdraw amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))

(define (make-accumulator x)

(lambda (sonuc) (set! x (+ sonuc x)) x))

(define A (make-accumulator 5))

(A 10)

(A 5)

(A 6)

sonuclar;

15

20

26

(put <op> <type> <item>)

(get <op> <type>)

BANK ACCOUNT

(define balance 100)

(define (withdraw amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))

(define new-withdraw

(let ((balance 100))

(lambda (amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))))

(define (make-withdraw balance)

(lambda (amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds")))

(define W1 (make-withdraw 100))

(define W2 (make-withdraw 100))

(W1 50)

(W2 70)

(define (make-account balance)

(define (withdraw amount)

(if (>= balance amount)

(begin (set! balance (- balance amount))

balance)

"Insufficient funds"))

(define (deposit amount)

(set! balance (+ balance amount))

balance)

(define (dispatch m)

(cond ((eq? m 'withdraw) withdraw)

((eq? m 'deposit) deposit)

(else (error "Unknown request -- MAKE-ACCOUNT"

m))))

dispatch)

(define account-deniz (make-account 100))

((account-deniz 'withdraw) 50)