CS 316 3/19/2014 1

Remider

Do the reading assignment and exercises (an the syntax of expression) before Exam!

Lisp

Double Recursion ("divide-and-conquer")

A doubly recursive function computes its result from two different recursive calls (for at least some argument values).

No. of recursive calls grows exponentially with the depth of recursion. (But in many doubly recursive function the depth of recursion grows only Logarithmically with argument size and so the no. of recursive calls does not grow exponentially with argument size).

Ex MSORT from Asn 5.

(msort L) ⇒ a sorted list of the elements of L.

L: List of real number; sorted: ascending order.

(msort '(3 7 1 6 5 8 2))
⇒ (1 2 3 4 5 6 7 8).

(msort '(2 7 3 0 4 5 8))

(msort '(2 3 4 8)) ⇒ (2 3 4 8)

(msort '(7 0 5)) ⇒ (0 5 7)

(merge-list '(2 3 4 8) '(0 5 7))
⇒ (0 2 3 4 5 7 8).

Another picture (2 7 3 0 4 5 8).

split-list

(2 3 4 8)   (7 0 5)

msort → (2 3 4 8)   msort → (0 5 7)

merge-lists
↓
(0 2 3 4 5 7 8).

Another example:

(Qsort L) ⇒ Same result as (Msort L).

(Qsort '(4 1 7 -2 5 6 8 3)). wrt = with respect to.

Partition wrt car

(1 -2 3) (4 7 5 6 8)

qsort qsort

(-2 1 3) (4 5 6 7 8)

(-2 1 3 4 5 6 7 8).

When does the above recursive strategy fail?

① L is NIL

or ② First element of L is the first element is the smallest element of the list.

L = (-4 1 7 -2 5 6 8 3).

partition wrt car

( ) (-4 1 7 -2 5 6 8 3).

A special case of ② is when L is of length 1, but this ~~can~~ could be handled as a base case. A different recursive strategy is needed in case ②!

Functions that take functions as arguments

Ex: $\sum_{i=2}^{4} i^2 = 2^2 + 3^2 + 4^2 = 29$

```
(defun sqr (n) (* n n))
```

Ex: (sigma #'sqr 2 4) ⇒ 29

How can we write function SIGMA that is like Σ?

In Common Lisp [this does NOT apply to scheme] the value of a symbol as a variable (if any) is not related to the function definition of that symbol (if any)

SETF is one way to set the value

LET/LET* is another way to set the value.

Most importantly when a function is called, the value of each formal parameter is set to the value of the corresponding actual argument.

DEFUN sets the function definition of a symbol. (DEFUN has no effect on the value of the symbol.)

```
(defun sqr (n) (* n n))
(setf sqr 5)
```

$$\boxed{\begin{array}{l} > \text{(sqr sqr)} \\ 25 \end{array}}$$

#'f is an expression whose value is the ~~defini~~ function definition of f.

This explain why we wrote (sigma #'sqr 2 5) and NOT (sigma sqr 2 5).

~~Function~~ Funcall

(funcall g a_1 a_2 a_3 ... a_k).

calls the function given by the value of g with a_1, ..., a_k as the actual arguments.

```
(setf g #'sqr)
(defun g (n) (+n 17))
```

>(g 3) 20	>(funcall g 3) ∈ g = sqr. 9	>(funcall #'g 3) 20

In scheme, there is no difference between function definition and value. So FUNCALL and #' are not used in scheme.

Now we are ready to write the SIGMA function.

```
(defun sigma (f m n)
    (if (>m n))
      0
      (+ (funcalls f m)
         (sigma f (+ m 1) n))))
```

$$\sum_{i=m}^{n} f(i) = f(m) + \cdots + f(n)$$

$$= \begin{cases} f(m) + \sum_{i=m+1}^{n} f(i) & \text{if } m \le n \\ 0 & \text{of } m > n \end{cases}$$

CS316 3/19/2014 4

MAPCAR (map in scheme)

(mapcar #'round '(7.1 6.8 3.2 3.1 5.9))

⇒ (7 7 3 3 6)

(mapcar f L) ⇒ a list of same length as L in which the ith element is obtained by appling f to the ith element of L.

(mapcar #'sqr (2 1 6 3 9)) ⇒ (4 1 36 9 81)

Let's write our own vesion of Mapcar.

(defun our-mapcar (f L)

L = (e1 e2 ... ek)   (our-mapcar f L) should ⇒ ((f e1) (f e2) ... (f ek))

```
(defun our-mapcar (f L)
    (if (endp L)
        ()
        (cons (funcall f (car L))
              (our-mapcar f (cdr L)))))
```

The build-in ~~mapcall~~ Mapcar (unlike our-MAPCAR) can also be used to map function of 2 or more arguments.

(mapcar #'+ '(1 7 6 3) '(2 4 5 8) '(0 1 6 1)) ⇒ (3 12 17 12)

(mapcar #'cons '(A B C) '((1 2) (3 4 5) (6)))

⇒ ((A 1 2) (B 3 4 5) (C 6))

~~Lambdy~~ Lambda Expression (also ~~avaitd~~ available in Jave, and any other res 8)

Lambda expression evaluate to ~~number~~ nameless function

Syntax is the same as for DEFUN but without the function name.

(lambda (n) (* n n))

((lambda (n) (* n n)) 5) ⇒ 25

(sigma (lambda (k) (* k k)) 2 4) ⇒ 29. $\sum_{k=2}^{4} k^2$

Lamda Expression (contd.)

```
> ((lamda (x y z) (+ x (* y z))) 2 1 7)
9
```

(optional #')

```
> (mapcar '(lambda (x) (+ x (* x x x)))
          '(2 1 0 4))
(10 2 0 64)
```

```
> (mapcar (lambda (x y) (+ x (* y 2)))
          '(1 0 2 3)
          '(0 1 1 2)) ⇒ (1 2 4 7)
```

Σ. (sigma (lambda (i) (+ (* i i) (* i 3))) 0 3) $\sum_{i=0}^{3}(i^2+3i)$.

$\Rightarrow 0^2+3*0 + 1^2+3*1 + 2^2+3*2 + 3^2+3*3$

$\quad 0 \quad\quad 4 \quad\quad 10 \quad\quad 18$

$\Rightarrow 32$

Important A lambda expression can use variables defined outside that lambda expression
relevant to FOO problem (Q14 on Asn 5)

Ex: Write a function SUM-POWERS such that

(sum-powers k m n) $= \sum_{i=m}^{n} i^k = m^k + (m+1)^k + \cdots + n^k$.

```
(defun sum-powers (k m n)
       (sigma (lamda (i) (expt i k)) m n)
```

Ex: INC-LIST-2: from assigment.

(inc-list-2 '(2 7 1 6 5) 3) ⇒ (5 10 4 9 8)

```
(defun inc-list-2 (L n)
       (mapcar (lambda (i) (+ i n)) L).
```

Apply Apply is like funcall but allows a list of arguments to be passed.

*last argument must be a list.

```
(apply #'+ '(2 1 7 9))
= (funcall #'+ 2 1 7 9)
= (+ 2 1 7 9) = 19.
= (apply #'+ 2 '(1 7 9))
= (apply #'+ 2 1 '(7 9))
= (apply #'+ 2 1 7 '(9))
= (apply #'+ 2 1 7 9 ())
```

(appy f i_1 i_2 ... i_k L) ← a List

calls f with the values of i_1 i_2 ... i_k and the elements of the values of L an arguments.

(funcall f i_1 i_2 ... i_k)

= (apply f (list i_1 i_2 ... i_k))

= (apply f i_1 (list i_2 ... i_k)) ... etc.

Ex. Sum from Asn-4

[(sum '(2 7 1 6)) ⇒ 16]

```
(defun sum (L)
    apply #'+ L).
```

(apply #'cons '((+ 2 3) (+ 4 A))) ⇒ ((+ 2 3) + 4 A)

This is not 5 here.

Ex: (relevant to 16c).

```
(apply #'mapcar #'+ '((1 2 3)
                      (4 0 7)
                      (0 1 1)
                      (3 3 2)))
= (mapcar #'+ '(1 2 3) '(4 0 7) '(0 1 1) '(3 3 2)) ⇒ (8 6 13)
```

syntax

syntax of Expression.

Expression can be written in many different notation / syntaxes.

Consider the expression : $f(g(h(1,2), f(3,4)), 5)$ (in java/C++ syntax)

This expression can be written in.

1. infix notation as : $((1\ h\ 2)\ g\ (3 f 4)) f\ 5$
   $((1 * 2) - (3+4)) + 5$
2. infix notation, with f and g having equal precedence. and belonging to a left associative precedence class, h having higher precedence than f and g.

   $1\ h\ 2\ g\ (3\ f\ 4)\ f\ 5.$ (if f = +, g = -, h = *)

   $1 * 2 - (3+4)+5$
3. Lisp notation (f (g (h 1 2) (f 3 4)) 5)
4. prefix notation = Lisp notation without parentheses. $\boxed{.\ f\ g\ h\ 1\ 2\ f\ 3\ 4\ 5.}$

   [This assumes you know how many arguments each operator/function takens]
5. "anti-Lisp" notation — like Lisp but function names appear at the ends of lists (instead of ~~that~~ at the beginings of list).

   (((1 2 h) (3 4 f) g) 5 f)
6. Postfix notation = anti-Lisp without parentheses.

   1 2 h 3 4 f g 5 f.

Syntax of infix notation

Terminology operator = symbol that denotes a function ~~the~~

The arity of an operator is the no. of arguments taken by the function

Each argument maybe called an operand.

An operator of arity n is called an n-ary operator

binary = 2-ary
unary = 1-ary
ternary = 3-ary.

CS316 3/24/2014 ⑥

Syntactically valid infix expressions (SVIEs) can be ~~called~~ defined as follows:

prefix unary operators

-y

-s

--x

*s

x++ postfix unary o[perator]

e is ~~an~~ a SVIE just of one of the following is true:

① e is a constant or a variable

② e is (e_1) where e_1 is an SVIE.

③ e is e_1 op e_2 where each of e_1 and e_2 is an SVIE and op is a binary operator

④ e is op e_1 where e_1 is an SVIE and op is a prefix unary operator

⑤ e is $e_1$op where e_1 is an SVIE and op is a postfix unary operator.

This definition of SVIE's as a serious weakness. Some syntact decompositions of SVIEs that are suggested by this definition are inconsistent with the desired semantics.

Ex: $e = x + y * z - w$

Can be decomposed using value ③ into

$$\begin{array}{ccc} e_1 & * & e_2 \\ \| & & \| \\ x+y & & z-w \end{array}$$

This is inconsistent with the usual semantics of arithmetic expressions.

After the exam, you will study sec 2.5 ~~with~~ which gives another way to specify SVIES that is unambiguous specification that is consistent with given precedence and associativity rules.

One way to fix the problem is to add to ③, ④, and ⑤, the rule that op should be the operator that is applied last, and to give rules that determitive which operator is applied last.

CS 316. Read assignment (P49) Sethi

2.1 a) $a*b+c = * + a\ b\ c$

b) $a*(b+c) = + * b\ c\ a$

c) $a*b+c*d = + * a\ b * b\ d$

d) $a*\cancel{b}(b+c)*d = * * a + b\ c\ d$

e) $(b/2 + sqrt((b/2)*(b/2) - a*c))/a =$

$= \cancel{*} / - + / b\ 2\ sqrt - * / b\ 2 / b\ 2 * a\ c\ a$

2.2. a) $a*b+c = a\ b * c +$

b) $a*(b+c) = b\ c + a *$

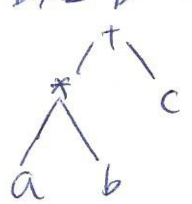
c) $a*b+c*d = a\ b * c\ d * +$

d) $a*(b+c)*d = b\ c + a * d *$

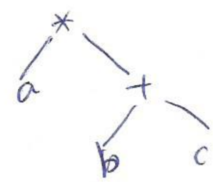
e) $(b/2 + sqrt((b/2)*(b/2) - a*c))/a$

$= \cancel{b/2}\ b\ 2 / b\ 2 / b\ 2 / * a\ c * - sqrt + a /$

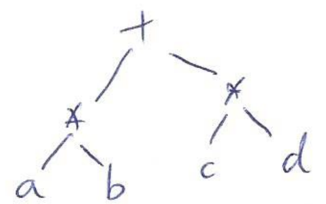
2.3. a)



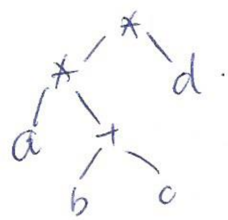
b)



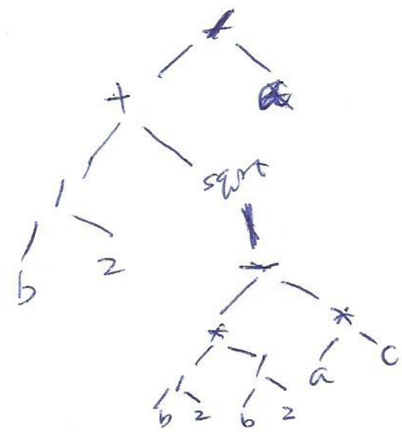
c)



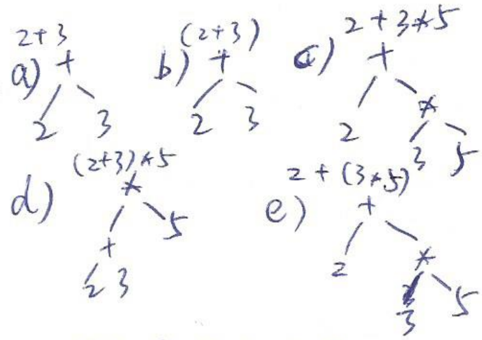
d)



e)



2.7.



2.8. 7 7 * 4 2 * 3 * -