Comp1002

Computational Systems

a)Admin

- COURSEWORK HANDIN MUST BE DONE ON A TEMPLATE!
- HANDIN ON PAPER ASWELL AS ONLINE

- Marks
  - 70% → Coursework (2, 35% each)
  - 30% → Exam “Application of theory”

b)Systems

- DrScheme (scheme ide)
- BIGLOO (scheme compiler, used to compile courseworks)

c)Functional Programming

- 'Quick' to program
- Very expensive
- Assignment BANNDED in this coursework
- A 'style' of programming

d)Evaluator

- Functional version of an interpreter
- eval

e)Scheme

- Teaching dialet of LISP
- created by Indiana & MIT universities
- Language specification is short → 50 to 60 pages
- Regarded by many as a scripting language
- Data & programs take the same form

f)Application Domain v Style

- Domain is the area in which the language is best used
  - .i.e. PHP → Webpages
  - Java → Programs which run 'anywhere'
  - Fortran → Manipulating numbers

- Style is seperate to the domain
- Scheme manipulating S-Expressions in a functional style

Functional principles

a)Principles

- Functional application (combination)
- Functional abstraction
- Naming
- Conditional
- Recursion

Using these principles, anything that can be computed can be computed.

b)Combination

<combination> ::= (<operator> <operand> ….)
.i.e. (cons 1 (1 2 3 4)) → (1 1 2 3 4)

Operator can't be a keyword such as quote!!

c)Variables
Most languages have variables stored in a table called an *environment*

Variables are *bound* to *symbols*

Operators/functions can be stored as variables and are evaluated as such!
Also programs can be stored as variables (as they are all s-expressions)

d)Evaluation of a combination
"Evaluate each subexpression of the combination and apply the value of the first to the values of the other subexpressions"

Order of subexpression evaluation is not specified !!

e)Compose
Putting functions together

(car (cdr (1 2 3 4)) → 2 *the second element of the list*

f)Functional abstraction – Lambda

- mathematical x |-> x + 1
- lambda calculus λx.(x+1)
- scheme (lambda (x) (+ x 1))

Lambda expressions in scheme evaluated to a closure aka function which references the expression and the environment in which it was created

g)Define – naming
"Dont use the value returned by definition, it is used to create a new binding in the current environment."

Define implemented differently on different scheme compilers/interrupters
Good for naming functions/closures though!!

h)Conditional

- if is a keyword

(if <predicate> <consequent> <alternate>)
if predicate is true, evaluate consequent else evaluate the alternate

```
(if (= x 1)
        #t
        #f)
```

- cond is a keyword (for conditional)

conditionals contain nested lists of predicates and actions with a final list with keyword else which is evaluated if all other predicates are false.

```
(cond (x)
        ((< x 0) -1)
        ((= x 0) 0)
        ((> x 0) 1))
```

More about S-Expressions

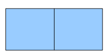
a)Append – concatenation

```
(define append
 (lambda (l1 l2)
  (if (null? L1)
     l2
     (cons (car l1)
      (append (cdr l1) (l2))))
```

b)Conceptual internal notation

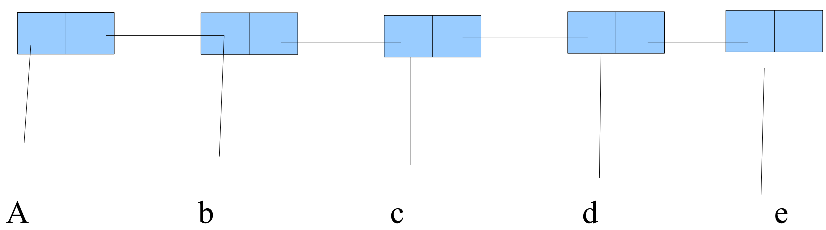
For representing how scheme in memory

Box notion:



These represents pointers in memory

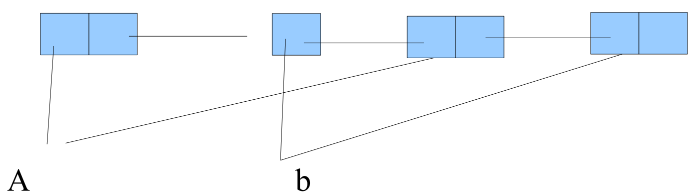
list (a b c d e)



last pointer can be null or point to the empty list

- Each symbol has a unique representation in memory

list (a b a b)



c) eq?
- Internal memory equality
- Like == in java

(define l1 '(1 2 3))

(define l2 (cons 1 (cdr (l2)))

(eq? l1 l2) → #f

(eq? (cdr l1) (cdr l2)) → #t as they point to the same memory location

d)Consing S-expressions

(cons 'a 'b) → (a . b)

This happens when constructing a where the second argument isn't a list/a pair so a pair is created (x . y) where a is the first element of the pair and b is the second element of the pair.

e)equal?

Equality based on the external representation of a structure

i.e. (equal? '(1 2 3 4) '(1 2 3 4)) → #t

"structual equality"

like .equals(obj o) in java

f)Vector

Like an array

(vector 'a l #t) → #(a l #t)

(vector-ref (vector 'a l #t) 0) → a ;; vector ref gets a certain element of a vector

(list-ref ..) does the same for lists

Vectors are quicker to access than lists as they are continuously stored in memory

Lists are stored as linked lists between the car and cdr so they take a linear time to access

How to recurse

a)Aim

Base case:
Inductive step:
Idea is to reduce the size of the problem until its trivial

"For recursion on a data-structure, look at its abstract data type structure and you will see how its defined"

b)Example: finding an item in a list
Base case: - empty list → #f
- car list == item → #t

Inductive step:
- return result of searching cdr of the list

c)memq and member
Tests whether an item is in a list
memq – phyiscal/memory equality
member – structural equality

d)Association list
A-List
Values associated with symbols
- ((x . 4) (y . 5) (z . 10))

List of pairs of symbols
assq used to find pair of values in a list for a symbol

e)Flat lists
Lists of symbols only
No sublists!

f)Binary trees
For this course only – they're represented as nodes (with no values just pointers to left and right)
Leaf nodes – contain a value and an identifier to say they're a leaf

Points can be obtained by using linked lists
- leafs are lists of 1 elements : value
- nodes are lists of 2 elements: pointers to left and right

Or using vectors
- leafs have 2 pointers: identifier, value
- nodes have 3 pointers: identifer, left and right nodes

g)Binary tree recursion
Base-case for leaves
Inductive case composed of recursive calls on the left and right sub trees and composing the result

Improve Recursion

a)Accumulators
Passing a partial solution as an argument in a recursive call

example: factorials
```
(define fact
 (lambda (n)
  (if (equal? N 0)
     1
    (* n (fact (- n 1))))
```

```
(define fact-iter
 (lambda (n acc)
   (if (equal? N 0)
    acc
    (fact-iter (- n 1) (* acc n))))
```

This helps reduce the need for a stack as no operations are needed when a function returns – THIS IS CALLED TAIL RECURSION!

In Java/C etc... this is done with while and for loops

Scheme compilers optimise to act the same as while loops with GOTO statements!!
This makes them *"properly tail recursive compilers"*

b)Iterations
Generally more desirable than recursion
Depends on language and compiler used

c)When not to use
When operation needed for iterations is more worse than linear complexity
This makes iterations quadratic and a recursive function might return it to linear time

d)Deeplists
Lists with lists
(1 3 (3 2(2 33))))

<deeplist> ::== () | (<element> . <deeplist>)
<element> ::== <symbolt> | <deeplist> | <number> | <boolean>

e)N-Array trees & XML
Can be implemented as deep lists in scheme
For N-Array trees see my implementation of binary trees!

f)Map
Allows you to map a function over a set
i.e. perform the function on every element of a list
Doesn't improve time complexity but improves readability

<u>More scheme functions</u>

a)Filters
Applies a predicate to a list
(filter odd? '(1 2 3 4 5)) gives (1 3 5)
Built into scheme
'higher order function'

b)Local Bindings
Applying a closure to a set of expressions creates a new temporary environment with bindings between the parameters of the closure & the values of the expressions. Only available when evaluating the closure.

c)Let
(let (var1 <expr1>)
....
(varn <exprn>)
<body>)

Special evaluation rule
built into scheme
Creates a local scope for the variables
Expressions evaluated first using outter environment then binded to variables to create the new environment

Cant be used to declare locally recursive functions!

d)Letrec
Used to define locally recursive functions

Variables binded to undeclared expressions first, then expressions evaluated using these
Allows for recursive functions to be declared locally
Can not have variables declared twice

e)Let*
Forces order of evaluation from left to right
Allows for variables that rely on each other

(let* ((x 3) (y (+ x 2))) (* x y)) gives 15

Order is left to right

g)Display
Used to display things to the stdout
Normally scheme interrupters write out the value of the scheme program to the stdout but display forces it to display more

h)Write
Same as display but scheme doesn't evaluate anything so this can be used to create fresh scheme programs from scheme
(write "hello") → "hello"
(display "hello")-> hello