25/4/2018

MODULE - 4

Functional Programming

* Functional programming language are designed to handle symbollic computation and list processing and it is based on mathematical functions
* categorized into 2:
  - → Pure functional language ⇒ functional programming paradigm (Haskell)
  - → Impure functional language
    ⇓
    functional PP + imperative Pgming Paradigm
    (Scheme, Lisp)

Characteristics

* functional programming language are designed on the concept of mathematical functions that can use conditional expression and recursion to perform computation.
* It supports higher order functions & lazy evaluation features.
* It doesn't supports flow controls like loop statements and conditional statements.
* like OOP, the functional programming language supports abstraction, encapsulation, inheritence, etc.

Applications

* Machine Learning
* Natural Language Processing
* Artificial Intelligence

Overview of Scheme Programming Concept.

* Example: (i) (+ 3 4)
  → Interpreter will print 7
  (ii) 7
  → Interpreter will print 7.
  (iii) ((+ 3 4))
  → Interpreter will print error
  (iv) (quote (+ 3 4)) or '(+ 3 4)
  → Interpreter will print (+ 3 4)

* Scheme is a statically scoped programming language.
* Each use of variable is associated with lexically apparent binding of that variable
* Procedures in scheme can be created dynamically, stored in data structure and returns as results of procedures.

Binding

* Binding can be achieved in 3 ways:
  (i) let ⇒ parallel
  (ii) let* ⇒ sequential
  (iii) letrec ⇒ processing time.

Lamda Expression

* Syntax for Lamda Calculus

E := var (variable)
| E1 E2 (function application)
| λx. E (function creation)

Ex (lamda (+ x y))
(λ (+ x y))

Examples:

(i) (+ (* 5 6) (* 8 3)) ⇒ (+ (30) (* 8 3))

(+ (30) (24)) = 54

(ii) (lamda (x)
(x+1))

→ Input: lamda (5), then result will be 6.

Example for Let:

(i) (let ((x 3) (y 4)
(square (lamda (x) (* x x)))
(plus +))
(sqrt (plus (square a) (square b))))

3×3 + 4×4

⇒ The result will be 5

(ii) (let ((a 3))
(let ((a 4) (y a))
(+ a y)))

parallel assignment

(let* ((a 4) (y a)) sequential assignment

↓

Then result will be 8

⇒ The result will be 7

20/11/18

Equality Testing and Searching

Equality Testing

Numerical comparison (=)

General Purpose:

(i) eqv? ⇒ to check whether 2 values are equal or not.

Used in while loops

shallow comparison is used.

Eg: (eqv? '2 '2) ⇒ true

(ii) equal? ⇒ Used in recursive functions

Deep comparison is used.

Eg: (equal? '(a) '(a)) ⇒ true

(iii) eq?

Eg: (eq? 'a 'a) ⇒ true

Comparison between variables

(eq? '(a) '(a)) ⇒ false [list]

Equality Searching

(memq 'z '(x y z w))

check whether z is present in the given list

(memv '(z) '(x y (z) w))

(member '(z) '(x y (z) w))

- (memq 'z '(x y z w)

  ⇒ (z w)

  * checks whether z is present in the given list (x y z w) and retrieves the list from the element z onwards
  * Works as eq?

- (memv '(z) '(x y (z) w))
  * Works as eqv?

* Shallow comparison
* Checks only values
* i.e, checks whether the value is present in the list

∴ false is retrieved as the o/p. Since this is a variable.

- (member 'z '(x y (z) w))
  * Works as ~~equ~~ equal?

Control Flow and Assignment

control flow

& If..then..else ...

```
(cond
    ((< 3 2) 1)    // if (3<2) printf("1")
    ((< 4 3) 2)    // else if (4<3) printf("2")
    (else 3))      // else 3
```

o/p: 3

Assignment

set! , ~~set~~ set-car! , set-cdr!

- ```
  (let ((x 2)          // x = 2
       (l '(a b)))     // l = '(a b)
  (set! x 3)           // x = 3
  (set-car! l '(c,d))  // l = '((c d) a b)
  (set-cdr! l '(e))    // l = '((c d) e)
  ... x
  ... l
  ```

Output: x = 3

l = '((c d) e)
  ```

Sequencing

```
(begin
   (display "hello")
   (display "world"))
```

Iteration (Fibanocci Series)

```
(begin (iter-fib (lambda (n)
 (do ((i 0 (+ i 1))
      (a 0 b)
      (b 1 (+ a b)))
     ((= in) b)
     (display b)
     (display " "))))
```

28/11/18

Horn Clause

* It consist of a head and a body consisting of terms B_i.

$P_1, P_2, P_3 \ldots P_n \rightarrow Q$

$B_1, B_2, B_3 \ldots B_n \rightarrow H$ (OR)

$H \leftarrow B_1, B_2, \ldots B_n$

Structure

atoms → lowercase letter sequence of character, quoted character string } eg:- foo, my_cost, 'Hai'

variables → Beginning with uppercase letters → eg: X, My_var

function or structure → consist of an atom and list of arguments

- teacher (arun, cs403)

Prolog

→ Based on facts and rules

Facts → factors

Rules → Horn clause

* **Facts**
  * without RHS

    Eg: rainy(india).

- **Rules**
  * has RHS

    Eg: snowy(x) :- rainy(x), cold(x).

- **Query**

→ rainy(india).

rainy(japan).

?rainy(x) // query

x = india // output

;

x = japan

;

No

} // ; is used to list all the resultant variables.

Resolution

takes(arun, CS401).

takes(arun, CS403).

takes(ajit, CS431).

takes(ajit, CS403).

classmates(x,y) :- takes(x,z), takes(y,z).

classmates(arun, x)?