Assignment 8.

Date
Page

Goachitla Greg

Explain functional programming of LISP.

Design Control Structure.

1) Atoms are the only primitives.

The only primitive control structures are literals and unquoted atoms since these are the only constructs that donot alter the control flow. Literals represents themselves. Unquoted atoms are bound to either functions (expr property) or data values (apval property). These are only two basic control structure constructors: the conditional expression and the reaursive application of a function to its arguments

The conditional Expression is a major contribution

Unlike previous languages such as FORTRAN and some other newer languages such as Pascal which require the user to drop from the expression level to the statement level in order to make a choice LISP introduced conditional expression.

Eg: $sg(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$

Date
Page

In LISP this function is defined:

```
(defun Sq(n)
    (cond ((plusp n) 1)
          ((zerop n) 0)
          ((minusp n) -1)))
```

The logical connectives are calculated conditionally.

McCarthy took the conditional expression as one of the fundamental primitives of LISP. Therefore it was natural to define the other logical expression in terms of it. eg: the function (or x, y) has the value t (true) if either or both of x & y have the value t in any other case, or has the value nil (false). Another way to say this is if x has the value t, then the or has the value t other or has the value same as y

(or x y) = (if x t y)

Iteration is done by Recursion

Except for the conditional - LISP needs none of the control structures found in conventional programming languages. In particular, all forms of iteration are performed by recursion.

In LISP it is more common to write recursive procedure that performs some operation on every element of list.

eg:- add all element of list

```
(defun plus-red (a)
  if (null a)
  0
```

Date
Page

```
plus (car a) (plus-red) (cdr a))))
(plus-red '(1 2 3 4 5))
=> 15
```

Hierarchical Structures are Processed Recursively-

We can process hierarchical structures that would be difficult to handle iteratively.

As an example we will design the equal function which determines whether two arbitary values are the same. How is this different from the eq primitive? Recall that the eq primitive works only on atoms: its application to non-atomic values is undefined (in most LISP systems). The equal function will do much more. It will tell us if two arbitary list structures are the same.

```
(equal '(a (b c) d (e)) '(a (b c) d (e)))
  t
(equal '(to be or not to be) '((to be) or (not (
    to be)))
  nil
(equal '(1 2 3) '(3 2 1))
  nil
(equal 'Parrs' (Don Smith))
  nil
(equal 'Paris' London)
  nil
(equal nil '(be 2))
```

Date
Page

The equal function is applicable to any two arguments.

Recursion and Iteration are theoretically equivalent.
i The difficulty of programming a function such as equal without recursion might lead us to suspect that recursion is more powerful than iteration. that is that there are things that can be done recursively that cannot be done iteratively which is not the case.

2g) Prove that pass by name is more powerful, dangerous and expensive in r^n to ALGOL. Also mention how it helped conceptually to users.

→ Pass by name has some dark corners and hidden inefficiencies. To show this, we will consider what should be a simple use of name parameters. an exchange procedure. The main idea is to write a procedure swap (x,y) that exchanges the values of the variables x & y. This definition would seem to do it.

```
procedure swap (x, y);
    integer x, y;
begin integer t;
    t := x;
    x := y;
    y := t;
end.
```

Date
Page

To see this works, we can try swap(i,j) and use the copy rule:

```
begin integer f;
    f := i;
    i := j;
    j := f;
end;
```

As we would expect from a swap procedure. ~~swap (j, i) proce~~ swap(j,i) produces exactly the same effect as swap(i,j). Now let's consider swap(A[i], j);

```
begin integer f;
    f = A[i];
    A[i] := i;
    i := f
end;
```

This is correct. Now consider swap(i, A[i]), which should have the same effat:

```
begin integer f;
    f := i;
    i := A[i];
    A[i] := f
end;
```

This does something completely different. If you do not see this, then try executing it assuming that i contains 1 and A[1] contains 27.

The effect will be to assign mistake when a simple procedure, such as swap, has such

Date
Page

Suprising properties. We have programmed this procedure in the obvious way and found that it doesn't work.

BQ) Explain descriptive tool (BNF) of ALGOL.

i) English Descriptions of Syntax Proved Inadequate.

An important skill in any design disciplines the use of descriptive tools to formulate, record, and calculate various aspects of developing design. Although English and other languages can often be used for this purpose, most designers have developed specialized languages, diagrams, and notations for representing aspects of their work that would otherwise be difficult to express.

eg:-

let us consider numeric denotations can be in three forms:

integer '-293'

fraction '3.14159'

scientific '6.022×10^{23}'

then from these examples we will have various questions like

integer '+273' is valid?

fraction '-14159' is valid?

Scientific '10.23' (10^{23}) is valid?

Date
Page

there can be various other ambugity so it is preferred to use seperate languages for this purpose.

Backus Developed a formal Syntatic Notations.

The problem with a description by enamples is that it is always incomplete. We saw in the no enample that there was a seemingly endless list of boarderline cases and questionable instances that weren of handled by our eg. On the other hand, the precise English description dealt with all these boarderline cases but really didn't show us when a no is in a very digestable form. What we need is a way of combining the perceptual clarity of enamples with the precision of formal prose.

Fortran restricts subscript expresions to one of the following forms:

V

V+c or V-C

C*V

C*V+C' or C*V-C'

iii) BNF is Naur's Adaptation of Backus Notation.

To see how Backus Notation works, let's take the no problem at (i). We can see that in our formal prose definition of nos we made use of several names for various syntatic components of numbers. for eg: integer represented strings like '3' '+3' and decimal fraction represented strings like '.02' and '0.1415', what we were really doing was defining certain syntatic categories of strings like decimal nos in terms of other syntatic categories like unsigned integer.

In ~~decim~~ description 3 of the definition of number, we said that a decimal fract

iv) ~~DNF~~ BNF Describes Alternaties

< unsigned integer >.
- < unsigned integer >
< unsigned integer >.

<< format of an << integer >>

Sroachitta Gurung.

MAYUR
Date
Page

Assignment 4.

LISP Data Structures in Brief.

i) The primitive include Numeric Atoms.

We can classify LISP's data structures into primitives and constructors. The principal constructor is the list, it permits more complicated structures to be built from ~~similar~~ simpler structures. The primitive data structures are the starting points for this building process. Thus the primitive data structures that are not built from any others. they have no parts. It is for this reason that they are called atoms ('atom' in Greeks indivisible thing).

ii) Non-numeric Atoms are also provided.

The other kind of primitive data structure in LISP is the non-numeric atom. These atoms are strings of characters that were originally intended to represent words or symbols.

We saw non-numeric atoms in the list (to be or not to be) with few expressions. The only operation that can be performed on non-numeric atoms are comparisons for equality and inequality. This is done with function eg.

(eq x y)

iii) The Principal Constructor is the list :

The characteristic method of data structuring provided by LISP is called the list. List are written in the S-expression notation by surrounding with paranthesis the lists elements, which are seperated by blanks. List can have none, one, or more elements. So they satisfy the zero-one-infinity principle. The elements of lists can themselves be lists. So the zero-one-infinity is also satisfied by nesting level of lists.

For a historical reason relating to the first LISP implementation. The empty list() is considered equivalent to the atom nil. That is

(eq '() nil)
(null '())

are both true (i.e. return if)

iv) Cdr and car acces the parts of lists:

An abstract data type is a set of data values together with a set of operations on these data values. What are the primitive list-processing operations ?

iv) A complete set of operations for a composite data type such as list, requires operations for building the structures and operations for taking them apart. Operations that build a structure are called constructs and those that extract their parts are called selectors.

LISP has one constructor - cons and two selectors -- car & cdr.

v) Information can be represented by Property list.

A personnel record would probably not be represented in LISP in the way we should have just described. It is too inflexible. Since each property of Don Smith is assigned a specific location in the list, it becomes difficult to change the properties associated with a given.

(name (Don Smith) age 45 salary 30000
hire-date (August 25 1980))

vi) Information can be Represented in Association lists.

The property list data structure works best when exactly one value is to be associated with each property. That is, a property list has the form:

(P1 V1 P2 V2 Pn Vn).

MAYUR
Date
Page

vii) Cons Constructs lists

We have seen that the car & cdr functions can be used to extract the parts of a list. How're lists constructed? When we design an abstract data type, we should make sure that constructors and selectors work together smoothly. This is necessary if data type is to be easy to learn & easy to use.

viii) lists are constructed usually recursively:

We saw that the value of (cons '(a b) '(c,d)) was not (a b c d). Suppose however that we want to concatenate two lists, and that (a b c d) is required result form (a b) and (c d). How can this be accomplished? ~~Interesti~~ Investigating the program.

(cons (car L) (cdr L)) = L

(car (cons x L)) = x

(cdr (cons x L)) = L.

Assignment 5-

Date
Page

LISP Name Structures :

i) The primitives Bind atoms to their values

As in the other programming languages, the primitive name structures are the individual bindings of names to their values. In LISP these bindings are established in two ways through property lists and through actual formal correspondence. The former is established by pseudo-functions such asset and defun.

eg:

(Set 'text' (to be or not to be))

ii) Application Binds formals to Actuals

The other primitive binding operation is actual-formal correspondence. This is very similar to other programming language.

```
(defun getprop (p n)
  (if (eq (car n) p)
      (cadr n)
      (get prop p (cddr n))))
```

and evaluate the application

(get prop 'name DS)

Date
Page

Temporary Bindings are a simple syntactic Extension

We have seen two methods of bindings to values. The definition of properties and actual-formal correspondence. What we have not seen is a method of binding names in a local context such as is provided by ALGOL's blocks. To see the need for this. we will work out a simple example. Suppose we want to write a program to compute both roots of a quadratic eq. They are

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We want to define a function root that returns a list containing the two roots. The general form of the function could be.

iv) Dynamic Scoping is the constructs

We have already mentioned that LISP uses dynamic scoping instead of static scoping used by other languages. Dynamic scoping means that a function is called in the environment of the caller, whereas Static Scoping means that if is called in the environment of its definition.