1

Lisp Programs

Every Lisp program has a representation as an s-expr

Lisp programs are organized into forms and functions.

Forms are expressions that can be evaluated to produce values (and side-effects).

Not all s-exprs are meaningful forms.

Forms that produce values are:
- atoms with values
- lists that are one of:
  - function call
  - special form
  - lambda form
  - macro call (to be discussed later)

Functions are invoked by *applying* them to arguments. A function performs computation by evaluating forms.

©1993 John Mylopoulos

2

Evaluating an Atom Form

Any atom may have a value (the same way variables have values in procedural languages).

Some atoms evaluate to themselves; for example, numbers, `nil, t,` any number, etc..Other, non-self-evaluating atoms return their value, if one has been assigned

For example:

The value of `abc` could be `23.`
The value of `abc` could be
`(John McCarthy 1961).`
The value of `abc` could be `abc.`
The value of `abc` could be *undefined.*

An atom need not have a value.

If an atom has an atomic value, its value could have a value as well:

The value of `abc` could be `xyz.`
The value of `xyz` could be `(+ 3 5)`

©1993 John Mylopoulos

3

Evaluating a List as a Form

Examples of valid forms are:

```
(+ 23 56)
(cons ham sandwich)
(cons (first city ) (rest province))
(first (quote (U of T)))
```

The first of the form determines the kind of form.

	first	**kind of form**
atom	function name	function invocation
list	lambda expression	lambda form
atom	special-form name	special-form
atom	macro name	macro call

©1993 John Mylopoulos

4

Evaluating a Function Call Form

A function call form looks as follows:

```
(function-name  arguments)
```

A function call is evaluated as follows:

- Get the *function name* that defines a function
- Evaluate each of the arguments in the *rest,* from left to right.
- Apply the function to the arguments to produce a value.

For example, suppose that

the value of `husband` is `(John Doe)`
the value of `wife` is `(Jane Smith)`

```
(cons (first wife) (rest husband))
```

- `cons` is the function name of a predefined function
- If the value of `(first wife)` is `Jane` and the value of `(rest husband)` is `(Doe)`, then applying `cons` to `Jane` and `(Doe)` gives:

???

©1993 John Mylopoulos

5

Evaluating Special Forms

Such forms are "special" because evaluation does not proceed according to the rules given for function form evaluation. Most special forms violate these rules in one of two ways:

• Arguments of the special form are not evaluated before the form itself is evaluated

e.g.,
`(setq x e)` evaluates `e`, but not `x`, and assigns the value of `e` to `x`

• The form admits a variable number of arguments

e.g.,
`(list` e_1 e_2 `...` e_n`)` creates a new list with n elements (`val(`e_1`) val(`e_2`) ... val(`e_n`))`

©1993 John Mylopoulos

6

The `Quote` Special Form

Sometimes we want to treat arguments as constants, rather than forms to be evaluated. For example, if we want to compute the length of the list `(a b c d e)`, we might write

```
(length (a b c d e))
```

but this won't work because Lisp will try to evalaute `(a b c d e)`, before applying `length` to the result.

To specify an argument without evaluating it, it must be "quoted" using the special form `quote`. Because quoting is done so often , Lisp supplies a special shorthand for it (`'`). The following two s-exprs are equivalent:

```
(first (quote (U of T)))   is   U
(first '(U of T))          is   U
```

So, to find the length of the list `(a b c d e)` we write

```
(length '(a b c d e))
```


©1993 John Mylopoulos

7

The `list` Special Form

• `list` takes a variable number of arguments (hence special form) and groups them together into a list.

For example,

```
--> (list 'a 'b 'c)
   (a b c)

--> (cons  'a  (list 'b 'c))
   (a b c)

--> (list)
   ?

--> (list ())
   ?
```


©1993 John Mylopoulos

8

Assigning Values to Non-Self-Evaluating Atoms

Lisp forms can also have side-effects, such as giving an atom a value. The special function `set`, and the special form function `setf` both return the value of an s-expr, but in addition they cause an assignment as side-effect.

For example:

```
(set  'section1 'csc324)
(set  section1 '(functional programming))
(set  'section2
           '(functional programming))
(setf section2 '(functional programming))
```

The first form assigns the atom `csc324` to the atom `section1`

The second form evaluates `section1` to the atom `csc324`, and assigns the value `'(functional programming)` to `csc324` using the function `set`.

The third form assigns `'(functional programming)` to the atom `section2`.

The last form is a special form that quotes `section2`, and assigns `'(functional programming)` to `section2` (in older Lisp dialects it was called `setq`)

©1993 John Mylopoulos

9

Conditional Special Forms: `if`

This special form allows a function to choose among two alternatives.

```
(if expr1 expr2 expr3)
```

is interpreted the same way as

```
if expr1 then expr2 else expr3
```

in a conventional programming language.

For example to calculate the absolute value of `x`, one can use `if` along with the following form:

```
(if   (> x 0)
      x
      (- x))
```


©1993 John Mylopoulos

10

Conditional Special Forms

`cond`

This special form takes an arbitrary number of **cond-pairs,** each a list of two elements:

```
(cond (cond1   val1)
      (cond2   val2)
      ...
      (condn   valn))
```

In order, it evaluates the first element of each pair (the condition) until it gets to one whose value is `t`. It then evaluates the second half of that pair and returns the result as its value. Later cond-pairs are never evaluated. If all conditions are `nil` then `cond` returns `nil`.

Here is the absolute value function redefined in terms of `cond`:

```
(cond
      ((> x 0) x)
      (t (- x))
                 )
```


©1993 John Mylopoulos

11

Lambda Expressions

In some Lisp dialects (including Common Lisp), new functions can be defined "on-line" in terms of the special form function `lambda`. A lambda form is a three element list consisting of:

- the atom `lambda`
- a list of atoms that name the arguments of the function being defined
- a form that defines the function body

For example:

```
(lambda (x) (first (rest x)))
(lambda (a b) (cons (car b) (cdadr a)) )
(lambda (something) something)
(lambda (x) (quote (Hello)) )
(lambda (x) (if (> x 0) x (- x)) )
```


©1993 John Mylopoulos

12

Applying Lambda-Defined Functions to Arguments

Such lambda expressions can be used wherever other functions can:

```
(lambda-expression argument-list)
```

```
((lambda (x) (first (rest x)) )
     '(my (very best) friend) )
-->     (very best)

((lambda (a b) (cons ( first b)
   (rest (first (rest a )))))
   '(my (very best) friend)
   '(Boole (and (also maybe)) Turing))
-->     (Boole best)

((lambda (x) ( if (> x 0) x (- x)))
     -3)
-->     3


((lambda (x) (first (rest x))) '(a b))
((lambda (something) something) 5)
((lambda (x) (quote (Hello)) ) '(a b))
```


©1993 John Mylopoulos