```
(<expression1> + <expression2>)  == add(<expression1>,<expression2>)
(<expression1> - <expression2>)  == diff(<expression1>,<expression2>)
(<expression1> * <expression2>)  == mult(<expression1>,<expression2>)
(<expression1> / <expression2>)  == quot(<expression1>,<expression2>)
<integer> == numb(<integer>)
```

For example:

```
1 == numb(1)
(1 + 2) == add(numb(1),numb(2))
((1 * 2) + 3) == add(mult(numb(1),numb(2)),numb(3))
((1 * 2) + (3 - 4)) == add(mult(numb(1),numb(2)),diff(numb(3),numb(4)))
```

Write a function which evaluates an arithemtic expression in this representation, for example:

```
- eval (numb(1));
> 1 : exp

- eval (add(numb(1),numb(2)));
> 3 : exp

- eval (add(mult(numb(1),numb(2)),numb(3)));
> 5 : exp

- eval (add(mult(numb(1),numb(2)),diff(numb(3),numb(4))));
> 1 : exp
```

10. FUNCTIONAL PROGRAMMING AND LISP

10.1. Introduction

LISP(LISt Processor) is a widely used artificial intelligence language. It is weakly typed with run-time type checking. Functions calls are evaluated in applicative order. LISP lacks structure and pattern matching.

Although LISP is not a pure functional language, it has a number of functional features. Like the λ calculus, LISP has an incredibly simple core syntax. This is based on bracketed symbol sequences which may be interpreted as list data structures or as function calls. The shared representation of data and program reputedly makes LISP particularly appropriate for artificial intelligence applications.

The original LISP programming system was devised by McCarthy in the late 1950’s as an aid to the Advice Taker experimental artificial intelligence system. McCarthy’s early description of LISP was based on a functional formalism influenced by λ calculus, known as **M-expressions(Meta expressions)**. These were represented in an extremely simple **S-expression’(Symbolic expression)** format for practical programming. Contemporary LISP systems are based solely on the S-expression format although other functional languages are reminiscent of the richer M-expressions. We won’t consider M-expressions here.

LISP, like BASIC, is not a unitary language and is available in a number of widely differing dialects. The heart of these differences, as we shall see, lies in the way that name/object associations are treated. Here we will consider COMMON LISP which is a modern standard. We will also look briefly at Scheme, a fully functional LISP. Other LISPs include FRANZ LISP which is widely available on UNIX systems, MACLISP and INTERLISP which COMMON LISP subsumes, and Lispkit Lisp which is another fully functional LISP.

It is important to note that LISP has been refined and developed for many years and so is not a very ‘clean’ language. LISP systems differ in how some aspects of LISP are implemented. Some aspects of LISP are extremely arcane and subject to much disputation amongst the cognoscenti. Furthermore, while LISP has much in common with functional languages it is actually used as an imperative programming language for most applications.

We will only look at enough LISP to see how it corresponds to our functional approach. Many details will, necessarily, be omited.

10.2. Atoms, numbers and symbols

The basic LISP objects are **atoms** composed of sequences of printing characters. Whenever a LISP system sees an atom it tries to evaluate it.

COMMON LISP provides distinct representations for integer, ratio, floating point and complex **number** atoms. Here, we will only consider integers. These consist of digit sequences preceded by an optional sign, for example:

```
0  42  -99
```

The result of evaluating a number is that number.

Symbols or **literals** are non-numeric atoms and correspond to names, for example:

```
banana BANANA forty_two --> +
```

Symbols have associated values. The result of evaluating a symbol is its associated value. There are a large number of system symbols with standard associated values known as **primitives.** As we will see, symbols are also objects in their own right.

10.3. Forms, expressions and function applications

The **form** is the basic LISP construct and consists of an atom or a left bracket followed by zero or more atoms or forms ending with a right bracket:

```
<form> ::= <atom> | ( <forms> ) | ()
<forms> ::= <form> | <form> <forms>
```

Forms are used for all expressions and data structures in LISP. Forms are always strictly bracketed except when special shorthand constructs are introduced.

Expressions are always prefix. The first form in a bracketed form sequence is interpreted as the function. Subsequent forms are interpreted as arguments. Thus, in a bracketed sequence of forms, we will refer to the first form as the **function** and to the subsequent forms as the **arguments.** We will also refer to primitives as if they were system functions.

Forms are evaluated in applicative order from left to right. For expressions, we will use `->` to indicate a result after applicative order evaluation.

Note that the function form may be the name of a function or a lambda expression but may NOT be an expression returning a function!

Note that the argument form may NOT be a lambda function or the name from a global definition or the name of a primitive!

Special primitives and techniques are used to treat functions as values.

10.4. Logic

```
t
```

is the primitive for `TRUE` and:

```
nil
```

is the primitive for `FALSE.`

```
not and or
```

are the primitives for the logical negation, conjunction and disjunction functions respectively. These may be used to construct simple logical expressions as forms, for example:

```
(not t) ->
nil

(and t nil) ->
nil

(or (and t nil) (not nil)) ->
t
```

In LISP, unlike most programming languages, `and` and `or` may have more than two arguments. For `and,` the final value is the conjunction of all the arguments, for example:

```
(and t nil t) ->
nil
```

For `or,` the final value is the disjunction of all the arguments, for example:

```
(or t nil t) ->
t
```

10.5. Arithmetic and numeric comparison

```
+ - * /
```

are the primitives for the addition, subtraction, multiplication and division functions. These may be used with numbers to construct simple arithmetic expressions, for example:

```
(+ 40 2) ->
42

(- 46 4) ->
42

(* 6 (+ 3 4)) ->
42

(/ (+ 153 15) (- 7 3)) ->
42
```

As with `and` and `or,` these functions may have more than two arguments, so `+` returns the sum, `-` the difference, `*` the product and `/` the overall quotient, for example:

```
    (+ 12 25 5) ->
    42

    (- 59 8 9) ->
    42

    (* 3 2 7) ->
    42

    (/ 336 4 2) ->
    42
```

`/` is actually a real division operator. The primitive

```
    truncate
```

rounds a single argument down to the nearest integer. If `truncate` is given two arguments then it divides one by the other and rounds the result down to the nearest integer, for example:

```
    (truncate 43 6) ->
    7
```

The primitive:

```
    rem
```

returns the integer remainder after division, for example:

```
    (rem 43 6) ->
    1
```

The numeric less than, less than or equal, equality, greater than and greater than or equal primitive comparison functions are:

```
     < <= = >= >
```

These may all be used with more than two arguments. Thus = checks that all its arguments are equal, for example:

```
    (= 2 2 2 2 2) ->
    t
```

<= checks that its arguments are in ascending order, for example:

```
    (<= 1 2 2 3 4 5) ->
    t
```

< checks that its arguments are in strictly ascending order, for example:

```
    (< 1 2 2 3 4 5) ->
    nil
```

>= checks that its arguments are in descending order, for example:

```
    (>= 5 4 3 3 2 1) ->
    t
```

and > checks that its arguments are in strictly descending order, for example:

```
(> 9 8 7 6 5) ->
t
```

The primitive:

```
numberp
```

returns true if its argument is a number. For example,

```
(numberp 42) ->
t
```

10.6. Lambda functions

LISP uses a notation like that for the λ calculus to define nameless functions.

It is important to note that LISP functions do not have all the properties we might expect from the λ calculus. In particular, special techniques are needed to pass functions as arguments, to return functions as values and to apply functions returned as values to new arguments.

Functions are defined as forms with the primitive:

```
lambda
```

followed by a flat list of bound variables and the body form:

```
(lambda (<bound variables>) <body>)
```

where:

```
<bound variables> ::= <bound variable> | <bound variable> <bound variables>
```

For example, to square a number:

```
(lambda (x) (* x x))
```

or to find the sum of the squares of two numbers:

```
(lambda (x y) (+ (* x x) (* y y)))
```

or to find the value of the quadratic:

ax^2+bx+c

given `a`, `b`, `c` and `x`:

```
(lambda (a b c x) (+ (* a (* x x)) (* b x) c)))
```

Note that functions are normally uncurried in LISP.

Note that LISP systems will reject attempts to present lambda functions directly as values other than in the function position in a form. `lambda` is not a primitive which denotes a system function. Instead it acts as a marker to indicate a function form. However, if a LISP system sees a naked lambda function form it will try to find a function associated with `lambda` and fail. The special techniques needed to manipulate function values are discussed below.

A function is applied to arguments in a form with the function followed by the arguments. The function’s body is evaluated with the bound variables associated initially with the corresponding arguments:

```
(<function> <argument1> <argument2> ... )
```

Note that arguments for uncurried functions are not bracketed but follow straight after the function.

For example:

```
((lambda (x) (* x x)) 2) ->
4

((lambda (x y) (+ (* x x) (* y y))) 3 4) ->
25

((lambda (a b c x) (+ (* a (* x x)) (* b x) c))) 1 2 1 1) ->
4
```

10.7. Global definitions

LISP systems are usually interactive and accept forms from the input for immediate evaluation. Global definitions provide a way of naming functions for use in subsequent forms.

A definition is a form with the primitive:

```
defun
```

followed by the name, bound variable list and the body form:

```
(defun <name> (<bound variable list>) <body>)
```

Many LISP systems print the defined name after a definition. For example:

```
(defun sq (x) (* x x)) ->
sq

(defun sum_sq (x y) (+ (* x x) (* y y))) ->
sum_sq

(defun quad (a b c x) (+ (* a (* x x)) (* b x) c)) ->
quad
```

A defined name may then be used instead of a lambda function in other forms. For example:

```
(sq 2) ->
4

(sum_sq 3 4) ->
25

(quad 1 2 1 2) ->
9
```

In particular, defined names may be used in other definitions. For example, the last two definitions might be shortened to:

```
(defun sum_sq (x y) (+ (sq x) (sq y)))

(defun quad (a b c x) (+ (* a (sq x)) (* b x) c))
```

It is important to note that a global definition establishes a special sort of name/value relationship which is not the same as that between bound variables and values.

10.8. Conditional expressions

LISP conditional expressions are forms with the primitive:

```
cond
```

followed by a sequence of options. Each option is a test expression followed by a result expression which is to be evaluated if the test expression is true:

```
(cond (<test1> <result1>)
      (<test2> <result2>)
      ...
      (t <resultN>))
```

Note that the last option's test expression is usually `t` to ensure a final value for the conditional.

When a conditional expression is evaluated, each option's test expression is tried in turn. When a true test expression is found, the value of the corresponding result expression is returned. This is like a nested sequence of `if ... then ... else` expressions.

Note that a conditional expression is not evaluated in applicative order.

For example, to find the larger of two values:

```
(defun max (x y)
 (cond ((> x y) x)
       (t y)))
```

or to define logical `not:`

```
(defun lnot (x)
 (cond (x nil)
       (t t)))
```

or to define logical `and:`

```
(defun land (x y)
 (cond (x y)
       (t nil)))
```

COMMON LISP provides a simpler conditional primitive:

```
if
```

which is followed by a test and expressions to be evaluated if the test is true or false:

```
(if <test>
    <true result>
    <false result>)
```

For example, to find the smaller of two numbers:

```
(defun min (x y)
 (if (< x y)
         x
         y))
```

or to define logical `or`:

```
(defun lor (x y)
 (if x
     t
     y))
```

10.9. Quoting

We said above that in LISP forms are used as both program and data structures. So far, we have used forms as function calls in general and to build higher level control structures using special system primitives which control form evaluation. In order to use forms as data structures additional primitives are used to prevent form evaluation. This is a different approach to the λ calculus where data structures are packaged as functions with bound variables to control the subsequent application of selector functions.

In LISP, a mechanism known as **quoting** is used to delay form evaluation. This is based on the idea from ordinary language use that if you want to refer to something’s name rather than that thing then you put the name in quotation marks. For example:

```
Edinburgh is in Scotland
```

is a statement about a city, whereas:

```
‘Edinburgh’ has nine letters
```

is a statement about a word. Putting in the quotation marks shows that we are interested in the letter sequence rather than the thing which the letter sequence refers to. In LISP, quoting is used to prevent form evaluation. Quoting a form shows that we are interested in the sequence of sub-forms as a structure rather than the form’s final value. Quoting is a special sort of abstraction for delaying evaluation. Later on, we will see how it can be reversed.

When the LISP primitive:

```
quote
```

is applied to an argument then that argument is returned unevaluated:

```
(quote <argument>) -> <argument>
```

In particular, a symbol argument is not replaced by its associated value but becomes an object in its own right.

Quoting is so widely used in LISP that the special notation:

```
’<argument>
```

has been introduced. This is equivalent to the above use of the `quote` primitive.

10.10. Lists

LISP is perhaps best known for the use of list processing as the basis of programming.

The empty list is the primitive:

```
nil
```

which may be also written as:

```
()
```

The tester for an empty list is the primitive:

```
null
```

This is the same as `not` because `FALSE` is `nil` in LISP and anything which is not `FALSE` is actually `TRUE`!

Lists may be constructed explicitly with the:

```
cons
```

primitive:

```
(cons <head> <tail>)
```

The `<head>` and `<tail>` arguments are evaluated, and a list with `<head>`s value in the head and `<tail>`s value in the tail is formed.

If the eventual tail is not the empty list then the **dotted pair** representation is used for the resultant list:

```
<head value> . <tail value>
```

For example:

```
(cons 1 2) ->
1 . 2

(cons 1 (cons 2 3)) ->
1. (2 . 3)

(cons (cons 1 2) (cons 3 4)) ->
(1 . 2) . (3 . 4)

(cons (cons 1 2) (cons (cons 3 4) (cons 5 6))) ->
(1 . 2) . ((3 . 4) . (5 . 6))
```

When a list ends with the empty list then a flat representation based on forms is used with an implicit empty list at the end. For example:

```
(cons 1 nil) ->
(1)

(cons 1 (cons 2 nil)) ->
(1 2)

(cons (cons 1 (cons 2 nil)) (cons (cons 3 (cons 4 nil)) nil)) ->
((1 2) (3 4))
```

Thus, as in chapter 6, the empty list `nil` is equivalent to the empty form `()`.

Note that lists built by `cons` and ending with the empty list appear to be forms but are not actually evaluated further as function calls.

Lists may be constructed directly in form notation and this is the most common approach. Note, however, that list forms must be explicitly quoted to prevent function call evaluation. For example:

```
(1 2 3)
```

looks like a call to the function `1` with arguments `2` and `3,` whereas:

```
'(1 2 3)
```

is the list:

```
1 . (2 . (3 . nil))
```

The primitive:

```
list
```

is a multi-argument version of `cons` but constructs a list ending with the empty list. For example:

```
(list 1 2) ->
(1  2)

(list 1 2 3) ->
(1 2 3)

(list (list 1 2) (list 3 4)) ->
((1 2) (3 4))

(list (list 1 2) (list 3 4) (list 5 6)) ->
((1 2) (3 4) (5 6))
```

The primitive:

```
listp
```

returns true if its argument is a list.

For example:

```
(listp '(1 2 3 4)) ->
t
```

10.11. List selection

The head of a LISP list is known as the **car** and the tail as the **cdr.** This is from the original IBM 704 implementation where a list head was processed as the ‘Contents of the Address Register’ and the tail as the ‘Contents of the Decrement Register’. Thus:

```
car
```

is the head selection primitive and:

```
cdr
```

is the tail selection primitive. For example:

```
(car ’(1 2 3)) ->
1

(cdr ’(1 2 3)) ->
(2 3)

(car (cdr ’(1 2 3))) ->
2

(cdr (cdr ’(1 2 3))) ->
(3)
```

Note that sub-lists selected from lists appear to be forms but are not further evaluated as function call forms by `car` or `cdr`.

10.12. Recursion

In LISP, recursive functions are based on function definitions with the defined name appearing in the function body. For example, to find the length of a linear list:

```
(defun length (l)
 (if (null l)
     0
     (+ 1 (length (cdr l)))))
```

and to count how often a value appears in a linear list of numbers:

```
(defun count (x l)
 (cond ((null l) 0)
       ((= x (car l)) (+ 1 (count x (cdr l))))
       (t (count x (cdr l)))))
```

For example, to insert a value into an ordered list:

```
(defun insert (x l)
 (cond ((null l) (cons x nil))
       ((< x (car l)) (cons x l))
       (t (cons (car l) (insert x (cdr l))))))
```

and to sort a list:

```
(defun sort (l)
 (if (null l)
     nil
     (t (insert (car l) (sort (cdr l))))))
```

10.13. Local definitions

LISP provides the:

```
let
```

primitive for the introduction of local variables in expressions:

```
(let ((<variable1> <value1>)
          (<variable2> <value2>)
          ...)
         (<result>))
```

is equivalent to the function call:

```
    ((lambda (<variable1> <variable2> ... ) <result>) <value1> <value2> ... )
```

For example, to insert a value into an ordered list if it is not there already:

```
    (defun new_insert (x l)
     (if (null l)
         (cons x nil)
         (let ((hl (car l))
               (tl (cdr l)))
              (cond ((= x hl) l)
                    ((< x hl) (cons x l))
                    (t (cons hl (new_insert x tl)))))))
```

Here, if the list is empty then a new list with the value in the head is returned. Otherwise, the head and tail of the list are selected. If the head matches the value then the list is returned. If the value comes before the head then it is added before the head. Otherwise the value is inserted in the tail.

10.14. Binary trees in LISP

In chapter 7 we looked at the construction of binary trees using a list representation. This translates directly into LISP: we will use `nil` for the empty tree and the list:

```
    (<item> <left> <right>)
```

for the tree with node value `<item>`, left branch `<left>` and right branch `<right>`.

We will use:

```
    (defun node (item left right) (list item left right))
```

to construct new nodes.

LISP has no pattern matching so it is useful to define selector functions:

```
    (defun item (l) (car l))

    (defun left (l) (car (cdr l)))

    (defun right (l) (car (cdr (cdr l))))
```

for the node value, left branch and right branch.

Thus, to add an integer to an ordered binary tree:

```
    (defun tadd (i tree)
     (cond ((null tree) (node v nil nil))
           ((< i (item tree)) (node (item tree) (tadd i (left tree) (right tree))))
           (t (node (item tree) (left tree) (tadd i (right tree))))))
```

For example:

```
(tadd 7 nil) ->
(7 nil nil)

(tadd 4 ’(7 nil nil)) ->
(7
 (4 nil nil)
 nil)

(tadd 10 ’(7 (4 nil nil) nil)) ->
(7
 (4 nil nil)
 (10 nil nil))

(tadd 2 ’(7 (4 nil nil) (10 nil nil))) ->
(7
 (4
  (2 nil nil)
  nil)
 (10 nil nil))

(tadd 5 ’(7 (4 (2 nil nil) nil) (10 nil nil))) ->
(7
 (4
  (2 nil nil)
  (5 nil nil))
 (10 nil nil))
```

Hence, to add a list of numbers to an ordered binary tree:

```
(defun taddlist (l tree)
 (if (null l)
     tree
     (taddlist (cdr l) (tadd (car l) tree))))
```

Finally, to traverse a binary tree in ascending order:

```
(defun traverse (tree)
 (if (null tree)
     nil
     (append (traverse (left tree))
             (cons (item tree) (traverse (right tree))))))
```

For example:

```
(traverse ’(7 (4 (2 nil nil) (5 nil nil) (10 nil nil))) ->
(2 4 5 7 10)
```

10.15. Dynamic and lexical scope

LISP is often presented as if it were based on the λ calculus but this is somewhat misleading. LISP function abstraction uses a notation similar to the λ abstraction but the relationship between bound variables and variables in expressions is rather opaque.

In our presentation of λ calculus, names have **lexical** or **static** scope. That is, a name in an expression corresponds to the bound variable of the innermost enclosing function to define it.

Consider the following contrived example. We might define:

```
def double_second = λx.λx.(x + x)
```

This is a function with bound variable:

```
x
```

and body:

```
λx.(x + x)
```

Thus, in the expression:

```
(x + x)
```

the `x`s correspond to the second rather than the first bound variable. We would normally avoid any confusion by renaming:

```
def double_second = λx.λy.(y + y)
```

For lexical scope, the bound variable corresponding to a name in an expression is determined by their relative positions in that expression, before the expression is evaluated. Expression evaluation cannot change that static relationship.

Early LISPs were based on **dynamic** scope where names' values are determined when expressions are evaluated. Thus, a name in an expression corresponds to to the most recent bound variable/value association with the same name, when that name is encountered during expression evaluation. This is effectively the same as lexical scope when a name is evaluated in the scope of the corresponding statically scoped bound variable.

However, LISP functions may contain free variables and a function may be created in one scope and evaluated in another. Thus, a name might refer to different variables depending on the scopes in which it is evaluated. Expression evaluation can change the name/bound variable correspondence.

For example, suppose we want to calculate the tax on a gross income but do not know the rate of tax. Instead of making the tax rate a bound variable we could make it a free variable. Our approach to the λ calculus does not allow this unless the free variable has been introduced by a previous definition. However, this is allowed in LISP:

```
(defun tax (gross)
 (/ (* gross rate) 100))
```

Note that `rate` is free in the function body. For a LISP with dynamic scope, like FRANZ LISP, `rate`'s value is determined when:

```
(/ (* gross rate) 100)
```

is evaluated. For example, if the lowest rate of tax is 25% then we might define:

```
(defun low_tax (gross)
 (let ((rate 25))
      (tax gross)))
```

When `low_tax` is applied to an argument, `rate` is set to `25` and then `tax` is called. Thus, `rate` in `tax` is evaluated in the scope of the local variable `rate` in `low_tax` and will have the value `25.`

In LISPs with dynamic scope this use of free variables is seen as a positive advantage because it delays decisions about name/value associations. Thus, the same function with free variables may be used to different effect in different places.

For example, suppose the average tax rate is 30%. We might define:

```
(defun av_tax (gross)
 (let ((rate 30))
      (tax gross)))
```

Once again, the call to `tax` in `av_tax` evaluates the free variable `rate` in `tax` in the scope of the local `rate` in `av_tax`, this time with value `30.`

It is not clear whether or not dynamic scope was a conscious feature or the result of early approaches to implementing LISP. COMMON LISP is based on lexical scope but provides a primitive to make dynamic scope explicit if it is needed. Attempts to move lexically scoped free variables in and out of different scopes are faulted in COMMON LISP.

10.16. Functions as values and arguments

In looking at the λ calculus, we have become used to treating functions as objects which can be manipulated freely. This approach to objects is actually quite uncommon in programming languages, in part because until comparatively recently it was thought that it was hard to implement.

In LISP, functions are not like other objects and cannot be simply passed around as in the λ calculus. Instead, function values must be identified explicitly and applied explicitly except in the special case of the function form in a function application form.

The provision of functions as **first class citizens** in LISP used to be known as the **FUNARG problem** because implementation problems centred on the use of functions with free variables as arguments to other functions. This arose because of dynamic scope where a free variable is associated with a value in the calling rather than the defining scope. However, it is often necessary to return a function value with free variables frozen to values from the defining scope. We have used this in defining typed functions in chapter 5.

The traditional way round the FUNARG problem is to identify function values explicitly so that free variables can be frozen in their defining scopes. The application of such function values is also made explicit so that free variables are not evaluated in the calling scope. This freezing of free variables in a function value is often implemented by constructing a **closure** which identifies the relationship between free and lexical bound variables.

COMMON LISP is based on lexical scope where names are frozen in their defining scope. None the less, COMMON LISP still requires function values to be identified and applied explicitly.

In COMMON LISP, the primitive:

```
function
```

is used as a special form of quoting for functions:

```
(function <function>)
```

It creates a function value in which free variables are associated with bound variables in the defining scope. As with `quote,` an equivalent special notation:

```
#’<function>
```

is provided.

Note that most LISP systems will not actually display function values as text. This is because they translate functions into intermediate forms to ease implementation but lose the equivalent text. Some systems may display an implementation dependent representation.

For example, we could define a general tax function:

```
(defun gen_tax (rate)
 #’(lambda (gross) (/ (* gross rate) 100)))
```

We might then produce the low and average tax rate functions as:

```
(gen_tax 25)
```

and

```
(gen_tax 30)
```

which return lambda function values with `rate` bound to `25` and `30` respectively.

The primitive:

```
funcall
```

is used to apply a function value to its arguments:

```
(funcall <function value> <argument1> <argument2> ...)
```

For example, to apply the low tax function to a gross income:

```
(funcall (gen_tax 25) 10000) ->
2500
```

This call builds a function value with `rate` bound to `25` and then applies it to the gross income `10000.` Similarly:

```
(funcall (gen_tax 30) 15000) ->
4500
```

builds a function value with `rate` bound to `30` and then applies it to the gross income `15000.`

For example, the mapping function `mapcar` may be defined as:

```
(defun mapcar (fn arg)
 (if (null arg)
     nil
     (cons (funcall fn (car arg)) (mapcar fn (cdr arg)))))
```

Note that the function `fn` is applied explicitly to the argument `car arg` by `funcall.` Now, we could square every element of a list with:

```
(defun sq_list (l)
 (mapcar #’(lambda (x) (* x x)) l))
```

Note that the function argument for `mapcar` has been quoted with `#’`.

Note that even if a function value argument is identified simply by name then that name must still be quoted with `#’` before it may be used as an argument. For example, to apply `sq` to every element of a list:

```
(mapcar #’sq ’(1 2 3 4 5))
```

10.17. Symbols, quoting and evaluation

Normally, symbols are variable names but they may also be used as objects in their own right. Quoting a symbol prevents the associated value being extracted. Thereafter, a quoted symbol may be passed around like any object.

One simple use for quoted symbols is as Pascal-like user defined enumeration types. For example, we might define the days of the week as:

```
'Monday 'Tuesday 'Wednesday 'Thursday 'Friday 'Saturday 'Sunday
```

We could then write functions to manipulate these objects. In particular, they may be compared with the equality primitive which is true if its arguments are identical objects:

```
eq
```

Note that in Pascal enumeration types are mapped onto integers and so have successor functions defined automatically for them. In LISP, we have to define a successor function explicitly if we need one. For example:

```
(defun next_day (day)
 (cond ((eq day 'Monday)    'Tuesday)
       ((eq day 'Tuesday)   'Wednesday)
       ((eq day 'Wednesday) 'Thursday)
       ((eq day 'Thursday)  'Friday)
       ((eq day 'Friday)    'Saturday)
       ((eq day 'Saturday)  'Sunday)
       ((eq day 'Sunday)    'Monday)))
```

The primitive:

```
eval
```

forces evaluation of a quoted form as a LISP expression. Thus, functions can construct quoted forms which are program structures for later evaluation by other functions. Thus, compiling techniques can be used to produce LISP from what is apparently data. For example, the rules for an expert system might be translated into functions to implement that system. Similarly, the grammar for an interface language might be used to generate functions to recognise that language.

For example, suppose we want to translate strictly bracketed infix arithmetic expressions:

```
<expression> ::= <number> |
                 (<expression> + <expression>) |
                 (<expression> - <expression>) |
                 (<expression> * <expression>) |
                 (<expression> / <expression>)
```

into prefix form so:

```
(<expression> + <expression>) == (+ <expression> <expression>)
(<expression> - <expression>) == (- <expression> <expression>)
(<expression> * <expression>) == (* <expression> <expression>)
(<expression> / <expression>) == (/ <expression> <expression>)
```

We need to extract the operator and place it at the head of a list with the translated expressions as arguments:

```
(defun trans (l)
 (if (numberp l)
     l
     (let ((e1 (trans (car l)))
           (op (car (cdr l)))
           (e2 (trans (car (cdr (cdr l))))))
          (list op e1 e2))))
```

For example:

```
    (trans ’(1 + 2)) ->
    (+ 1 2)

    (trans ’(3 * (4 * 5))) ->
    (* 3 (* 4 5))

    (trans ’((6 * 7) + (8 - 9))) ->
    (+ (* 6 7) (- 8 9))
```

Note that quoted symbols for infix operators have been moved into the function positions in the resultant forms.

Now, we can evaluate the translated expression, as a LISP form, using `eval:`

```
(defun calc (l)
 (eval (trans l)))
```

For example:

```
    (calc ’((6 * 7) + (8 - 9))) ->
    41
```

The treatment of free variables in quoted forms depends on the scope rules. For dynamic scope systems, quoted free variables are associated with the corresponding bound variable when the quoted form is evaluated. Thus, with dynamic scope, quoting can move free variables into different evaluation scopes.

In COMMON LISP, with lexical scope, quoted free variables are not associated with bound variables in the defining scope as might be expected. Instead, they are evaluated as if there were no bound variables defined apart from the names from global definitions.

10.18. λ calculus in LISP

We can use `function` and `funcall` for rather clumsy applicative order pure λ calculus. For example, we can try out some of the λ functions we met in chapter 2. First of all, we can apply the identity function:

```
    #’(lambda (x) x)
```

to itself:

```
    (funcall #’(lambda (x) x) #’(lambda (x) x))
```

Note that some LISP implementations will not print out the resultant function and others will print a system sepecific representation.

Let us now use definitions:

```
    (defun identity (x) x) ->
    identity

    (funcall #’identity #’identity)
```

Note once again that there may be no resultant function or an internal representation will be printed.

Let us apply the self application function to the identity function:

```
(funcall #’(lambda (s) (funcall s s)) #’(lambda (x) x))
```

Notice that we must make the application of the argument to itself explicit. Now, let us use definitions:

```
(defun self_apply (s) (funcall s s)) ->
self_apply

(funcall #’self_apply #’identity)
```

Finally, let u try the function application function:

```
(defun apply (f a) (funcall f a)) ->
apply
```

with the self application function and the identity function:

```
(funcall #’apply #’self_apply #’identity)
```

Pure λ calculus in LISP is complicated by this need to make function values and their applications explicit and by the absence of uniform representations for function values.

10.19. λ calculus and Scheme

Scheme is a language in the LISP tradition which provides function values without explicit function identification and application. Scheme, like other LISPs, uses the bracket based form as the combined program and data structure, is weakly typed and has applicative order evaluation. Like COMMON LISP, Scheme is lexically scoped. We are not going to look at Scheme in any depth. Here, we are only going to consider the use of function values.

In Scheme, functions are true first class objects. They may be passed as arguments and function expressions may appear in the function position in forms. Thus, many of our pure λ calculus examples will run with a little translation into Scheme. However, Scheme systems may not display directly function values as text but may produce a system dependent representation.

Let us consider once again the examples from chapter 2. We can enter a lambda function directly, for example the identity function:

```
(lambda (x) x)
```

We can also directly apply functions to functions, for example we might apply the identity function to itself:

```
((lambda (x) x) (lambda (x) x))
```

Alas, the representation of the result depends on the system.

Scheme function definitions have the form:

```
(define (<name> <argument1> <argument2> ...) <body>)
```

Defined functions may be applied without quoting:

```
(define (identity x) x) ->
identity
```

```
    (identity identity)
```

Note once again that there may be no resultant function or a system specific internal representation may be printed.

To continue with the self application and function application functions:

```
    (lambda (s) (s s))

    ((lambda (s) (s s)) (lambda (x) x))

    (define (self_apply s) (s s)) ->
    self_apply

    (self_apply identity)

    (define (apply f a) (f a)) ->
    apply

    (apply self_apply identity)
```

This explicit use of function values is much closer to the λ calculus than COMMON LISP though there is still no uniform representation for function values.

10.20. Other features

It is impossible to consider an entire language in a small chapter. Here, we have concentrated COMMON LISP in relation to pure functional programming. COMMON LISP has come along way since the original LISP systems. In particular, it includes a wide variety of data types which we have not considered including characters, arrays, strings, structures and multiple value objects. We also have not looked at input/output and other system interface facilities.

10.21. Summary

In this chapter we have:

- surveyed quickly COMMON LISP
- seen how to implement algorithms from preceding chapters in COMMON LISP
- seen that treating functions as objects in COMMON LISP involves explicit notation and that there is no standard representation for function results.
- seen that Scheme simplifies treating functions as objects but lacks a standard representation for function results.

10.22. Exercises

1) (c.f. Chapter 4 Exercises 1, 2, 3 & 4) Write functions to:

i) find the sum of the integers between `n` and `0.`

ii) find the product of the numbers between `n` and `1.`

iii) find the sum of applying a function `fun` to the numbers between `n` and `0.`

iv) find the sum of applying a function `fun` to the numbers between `n` and `zero` in steps of `s.`

2) (c.f. Chapter 6 Exercise 2)

i) Write a function which indicates whether or not a list starts with a sub-list. For example:

```
(lstarts ’(1 2 3) ’(1 2 3 4 5)) ->
t

(lstarts ’(1 2 3) ’(4 5 6)) ->
nil
```

ii) Write a function which indicates whether or not a list contains a given sub-list. For example:

```
(lcontains ’(4 5 6) ’(1 2 3 4 5 6 7 8 9)) ->
t

(lcontains ’(4 5 6) ’(2 4 6 8 10)) ->
nil
```

iii) Write a function which counts how often a sub-list appears in another list. For example:

```
(lcount ’(1 2) ’(1 2 3 1 2 3 1 2 3)) ->
3
```

iv) Write a function which removes a sub-list from the start of a list, assuming that you know that the sub-list starts the list. For example:

```
(lremove ’(1 2 3) ’(1 2 3 4 5 6 7 8 9)) ->
(4 5 6 7 8 9)
```

v) Write a function which deletes the first occurence of a sub-list in another list. For example:

```
(ldelete ’(4 5 6) ’(1 2 3 4 5 6 7 8 9) ->
(1 2 3 7 8 9)

(ldelete ’(4 5 6) ’(2 4 6 8 10)) ->
(2 4 6 8 19)
```

vi) Write a function which inserts a sub-list after the first occurence of another sub-list in a list. For example:

```
(linsert ’(4 5 6) ’(1 2 3) ’(1 2 3 7 8 9)) ->
(1 2 3 4 5 6 7 8 9)

(linsert ’(4 5 6) ’(1 2 3) ’(2 4 6 8 10)) ->
(2 4 6 8 10)
```

vii) Write a function which replaces a sub-list with another sub-list in a list. For example:

```
(lreplace ’(6 5 4) ’(4 5 6) ’(9 8 7 6 5 4 3 2 1)) ->
(9 8 7 4 5 6 3 2 1)

(lreplace ’(6 5 4) ’(4 5 6) ’(2 4 6 8 10)) ->
(2 4 6 8 10)
```

3) (c.f. Chapter 6 Exercise 3)

i) Write a function which merges two ordered lists to produce an ordered list.

ii) Write a function which merges a list of ordered lists.

4) (c.f. Chapter 7 Exercise 1)

The time of day might be represented as a list with three integer fields for hours, minutes and seconds:

```
(<hours> <minutes> <seconds>)
```

For example:

```
(17 35 42) == 17 hours 35 minutes 42 seconds
```

Note that:

```
24 hours = 0 hours
1 hour == 60 minutes
1 minute == 60 seconds
```

i) Write functions to convert from a time of day to seconds and from seconds to a time of day. For example:

```
(too_secs (2 30 25)) ->
9025

(from_secs 48975) ->
(13 36 15)
```

ii) Write a function which increments the time of day by one second. For example:

```
(tick (15 27 18)) ->
(15 27 19)

(tick (15 44 59)) ->
(15 45 0)

(tick (15 59 59) ->
(16 0 0)

(tick (23 59 59) ->
(0 0 0)
```

iii) In a shop, each transaction at a cash register is time stamped. Given a list of transaction details, where each is a string followed by a time of day, write a function which sorts them into ascending time order. For example:

```
(tsort '((haggis (12 19 57))
         (bannocks (18 22 48))
         (white_pudding (10 12 35))
         (oatcakes (15 47 19)))) ->
((white_pudding (10 12 35))
 (haggis (12 19 57))
 (oatcakes (15 47 19))
 (bannocks (18 22 48)))
```

5) (c.f. Chapter 7 Exercise 2)

i) Write a function which compares two integer binary trees.

ii) Write a function which indicates whether or not one integer binary tree contains another as a sub-tree.

iii) Write a function which traverses a binary tree to produce a list of node values in descending order.