Common Lisp in a Nutshell

19th September 2003

Contents

1 Alan Perlis on Programming[1]

Educators, generals, dieticians, psychologists, and parents program. Armies, students, and some societies are programmed. An assault on large problems employs a succession of programs, most of which spring into existence en route. These programs are rife with issues that appear to be particular to the problem at hand. To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs – many of them. It doesn't matter much what the programs are about or what applications they serve. What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs. The programmer must seek both perfection of part and adequacy of collection. In this book the use of "program" is focused on the creation, execution, and study of programs written in a dialect of Lisp for execution on a digital computer. Using Lisp we restrict or limit not what we may program, but only the notation for our program descriptions.

Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in de-

[1] I didn't write this section. It is Alan Perlis's foreword to the Computer Science classic SICP. I couldn't resist the temptation to put it up in this second edition of my Lisp notes. Never mind if you can't follow it! Just read it in full - and try to reflect a little bit.I have made a few additions - notably on currying, the metacircular evaluator and Church's Lambda Calculus - I had hoped to include streams and lazy evaluation, but couldn't.

tail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle. The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!

For all its power, the computer is a harsh taskmaster. Its programs must be correct, and what we wish to say must be said accurately in every detail. As in every other symbolic activity, we become convinced of program truth through argument. Lisp itself can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness argument. Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs. Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure – we call them idioms – and learn to combine them into larger structures using organizational techniques of proven value. These techniques are treated at length in this book, and understanding them is essential to participation in the Promethean enterprise called programming. More than anything else, the uncovering and mastery of powerful organizational techniques accelerates our ability to create large, significant programs. Conversely, since writing large programs is very taxing, we are stimulated to invent new methods of reducing the mass of function and detail to be fitted into large programs.

Unlike programs, computers must obey the laws of physics. If they wish to perform rapidly – a few nanoseconds per state change – they must transmit electrons only small distances. The heat generated by the huge number of devices so concentrated in space has to be removed. An exquisite engineering art has been developed balancing between multiplicity of function and density of devices. In any event, hardware always operates at a level more primitive than that at which we care to program. The processes that transform our Lisp programs to "machine" programs are themselves abstract models which we program. Their study and creation give a great deal of insight into the organizational programs associated with programming arbitrary models. Of course the computer itself can be so modeled. Think of it: the behavior of the smallest physical switching element is modeled by quantum mechanics described by differential equations whose detailed behavior is captured by numerical approximations represented in computer programs executing on computers composed of ...

It is not merely a matter of tactical convenience to separately identify the three foci. Even though, as they say, it's all in the head, this logical separation induces an acceleration of symbolic traffic between these foci whose richness, vitality, and potential is exceeded in human experience only by the evolution of life itself. At best, relationships between the foci are metastable. The computers are never large enough or fast enough. Each breakthrough in hardware technology leads to more massive programming enterprises, new organizational principles, and an enrichment of abstract models. Every reader should ask himself periodically "Toward what end, toward what end?" – but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy.

Among the programs we write, some (but never enough) perform a precise mathematical function such as sorting or finding the maximum of a sequence of numbers, determining primality, or finding the square root. We call such programs algorithms, and a great deal is known of their optimal behavior, particularly with respect to the two important parameters of execution time and data storage requirements. A programmer should acquire good algorithms and idioms. Even though some programs resist precise specifications, it is the responsibility of the programmer to estimate, and always to attempt to improve, their performance.

Lisp is a survivor, having been in use for about a quarter of a century. Among the active programming languages only Fortran has had a longer life. Both languages have supported the programming needs of important areas of application, Fortran for scientific and engineering computation and Lisp for artificial intelligence. These two areas continue to be important, and their programmers are so devoted to these two languages that Lisp and Fortran may well continue in active use for at least another quarter-century.

Lisp changes. The Scheme dialect used in this text has evolved from the original Lisp and differs from the latter in several important ways, including static scoping for variable binding and permitting functions to yield functions as values. In its semantic structure Scheme is as closely akin to Algol 60 as to early Lisps. Algol 60, never to be an active language again, lives on in the genes of Scheme and Pascal. It would be difficult to find two languages that are the communicating coin of two more different cultures than those gathered around these two languages. Pascal is for building pyramids – imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms – imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. The organizing principles used are the same in both cases, except for one extraordinarily important difference: The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. Lisp programs inflate libraries with functions whose utility transcends the application that produced them. The list, Lisp's native data structure, is largely responsible for such growth of utility. The simple structure and natural applicability of lists are reflected in functions that are amazingly nonidiosyncratic. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish.

To illustrate this difference, compare the treatment of material and exercises within this book with that in any first-course text using Pascal. Do not labor under the illusion that this is a text digestible at MIT only, peculiar to the breed found there. It is precisely what a serious book on programming Lisp must be, no matter who the student is or where it is used.

Note that this is a text about programming, unlike most Lisp books, which are used as a preparation for work in artificial intelligence. After all, the critical programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation become larger. This explains why there is such growing interest in Lisp outside of artificial intelligence. As one would expect from its goals, artificial intelligence research generates many significant programming problems. In other programming cultures this spate of problems spawns new languages. Indeed, in any very large programming task a useful organizing principle is to control and isolate traffic within the task modules via the invention of language. These languages tend to become less primitive as one approaches the boundaries of the system where we humans interact most often. As a result, such systems contain complex language-processing functions replicated many times. Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems. Finally, it is this very simplicity of syntax and semantics that is responsible for the burden and freedom borne by all

Lisp programmers. No Lisp program of any size beyond a few lines can be written without being saturated with discretionary functions. Invent and fit; have fits and reinvent! **We toast the Lisp programmer who pens his thoughts within nests of parentheses.**

2 '(Lots (of) (Irritating Silly) (Parenthesis))

Eric S Raymond's *How to become a Hacker* is a must read for anyone who is at least moderately interested in the fascinating art and science of programming. Raymond tells us that we should learn at least five programming languages - Python, C/C++, Lisp, Perl and Java. This is what he has to say about LISP:

> LISP is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot.

How very true! LISP was conceived by Prof. John Mccarthy at the Mecca of Computer Science, the Massachusettes Institute of Technology, around 1960. Don't ever dare to think that it's an 'old' language and hence outdated. Most of the 'advanced' programming techniques being touted by advocates of 'modern' languages in todays 'language wars' had their origins in LISP. But then how come LISP never became a mainstream programming language? Part of the blame lies on the way the language was packaged and marketed. From the beginning itself, LISP came to be associated with high-brow academic research[2] - almost completely divorced from practical industrial use[3]. It seems that the 'intellectuals' who used the language gratified their ego by perpetuating the myth[4] that the language was meant to be used only by an elite group of people like themselves! Only now has the situation started to change, with people like *Sriram Krishnamurthy*[5] actively campaigning for the use of Lisp dialects (Scheme) in elementary Computer Science education[6].

[2]You should expect a language motivated by Alonzo Church's Lambda Calculus to be pretty sophisticated. It really is. I can show you 'brain exploding' LISP programs which are barely a few lines long! Dan Friedman asks at the end of a lengthy discussion on the Y combinator - do you know why Y works? He gently mocks at you - maybe, now you need a bigger hat!

3 A bad approach - Lisp as C

Lisp is a 'functional' programming language[7]. To write good LISP code, you have to know the 'Philosophy' of LISP. There are very few textbooks which deal with the 'LISP approach' - most books simply explain the syntax of the language - in fact, the only book I know of is 'Structure and Interpretation of Computer Programs'[8] which is used to teach a common programming language course to fresh engineering students at some of the worlds top universites like MIT. The book is available on the Web for free - download it and read at least the Preface, Table of Contents[9] and parts of the first chapter - you will

[3]Which is false. Common LISP is an industrial-strength tool and is used extensively to solve complex problems.

[4]Lisp was also plaugued by myths of being slow and inefficient. Another problem was that when Artifical Intelligence failed to deliver its lofty promises in the early 1980's, LISP too was considered to have 'failed' - AI and Lisp had got so mixed together.

[5]An American University Professor of Computer Science.

[6]Krishnamurthy's TeachScheme project has developed a fascinating programming environment called DrScheme and a magnificent book called 'How to Design Programs'. The book as well as the Scheme programming environment are freely downloadable from the project web site.

[7]I will try to tell you something about the various programming 'paradigms' in some other article.

[8]Also called SICP. Another name is the 'Wizard Book'.

[9]The book has only 5 chapters - the 4th chapter deals with metalinguistic abstractions - the metacircular evaluator, streams, lazy evaluation....! The final chapter is 'Computing with register machines' - which is basically an introductory treatise on compiling LISP. It is said that haughty students, with plenty of experience in 'C/C++/Java' who take this course are brought down to earth in the first two or three lectures - they are made to realize that computing is vastly different from what they think it is. Many drop off, and those who stick on usually go on to take PhD's and become world renowned scientists!

very soon realize how backward we are in Computer Science education (and for that matter, in any kind of education).

It is not possible for me to teach you a lot[10] about 'LISP Philosophy' for the very simple reason that I know so little of it - I abandoned SICP at the end of the second[11] chapter! So what I will do is I will try to levereage your understanding of recursion and present some simple programs (and some not so simple programs) in as simple (or rather, I shall coin a term - as C'ish) a manner as possible - with the full understanding that what I am following is not the right approach - but, at the moment, a practical one. This document is merely a quick reference to the Common LISP syntax and standard functions - my classes will make up for the details. Those students who desire true enlightenment are requested to meditate under some kind of a tree with printouts of the following books:-

1. How toDesign Programs - Sriram Krishnamurthy, etc
2. Structure and Interpretation of Computer Programs - Sussman etc.
3. The Little Lisper - Daniel Friedman.

4 A little bit about the functional programming paradigm

A small exercise. Read up the definition of the word 'paradigm' in an English dictionary.

The human being is extremely ingenious - he delights in solving problems, creating newer ones, solving them, creating newer (and maybe better) solutions to problems which have been already solved.... Thus the Computer Scientist and the programmer is not satisfied with the motto 'there is only one way to do it'. He has a better one, and it is call TIMTOWTDI (There Is More Than One Way To Do It - read as 'Tim-Toddy' - which is also the official motto of the Perl[12] programming language).

Your text books may give you tons of reasons as to why one programming methodology is better than the other - but the people who write these things themselves know that what they are writing is not completely true. There are some problems which can be solved better by using a functional programming methodology and there are others which can be solved better by using a declarative style or an object oriented style. Programming languages and methodologies are merely tools used for the software simulation of complex real world systems - how well the simulation works depends as much on the skill, experience and wisdom of the person (or persons) who implement these systems as on the methodology they employ. Though it is unlikely that a team of brilliant programmers who know only COBOL would build a complex Artificial Intelligence system in COBOL, it is much more probable than a team of mediocre programmers trained in LISP implementing the same in LISP. The interested student may refer the magnificent book 'The Mythical Man Month' by Fred Brooks[13] which discusses the real problems inherent in designing complex software systems.

We start off by accepting the fact that there are different 'paradigms' (styles, methodologies) of programming - imperative, functional, object oriented (and object based), declarative. We say that LISP is a functional language and C++ an Object Oriented language - but that does not mean that LISP is not Object Oriented or C++ is not functional - it is possible to write functional style programs in C++ and

[10]But I shall try a little bit. I can tell you a little bit about lambda calculus, higher order functions, currying, lexical closures and streams. Some of you might get inspired to study more.

[11]Update for year 2003 - I am now able to digest randomly chosen parts of SICP - I did a metacircular evaluator and some cool lambda calculus stuff!

[12]Perl (note the spelling!) is a great language designed by the linguist and programmer Larry Wall. Perl is behind most of the useful interactive content on the world wide web - when you search Google or Yahoo, you are in fact executing Perl scripts residing on the server machines.

[13]I would classify this book as a 'must read' for any software engineer. By the time the reader finishes the book, he will realize that 'there are no magic bullets' - ie, there is no single process or methodology which could solve all the problems being faced by developers involved in the design of complex systems. The book is there in your college library - and unfortunately remains neglected.

Object Oriented programs in LISP. Think of it this way - when I say that XYZ is a good boy, what I mean is that XYZ is predominantly good - goodness is the quality which dominates. But I do not rule out the possibility of XYZ being nasty and mean - he can be, under certain situations. Likewise, LISP is predominantly functional, but we don't rule out the possibility of writing imperative or object oriented code.

Enough of beating around the bush. When does a programming language acquire the tag of a functional programming language? I shall adopt the explanation provided by John Hughes in his widely circulated memo 'Why functional programming matters'.

- ⊕ All computations are done through the application of functions.
- ⊕ Recursive application of functions dominates - iteration constructs are either not available at all or are added as 'syntactic sugar'[14]. Iteration is understood to be merely 'tail recursion', though it is not essential that tail recursive constructs are recognized and optimized[15].
- ⊕ Functions are first class - ie, they can be returned from functions and passed to functions just like other variables - thus facilitating the creation of higher order functions.
- ⊕ Programs are side effect free - functions are called simply to get the value being computed by them - function execution does not alter the global state of the program.
- ⊕ Lastly[16], as a functional programmer, you will often mouth terms like Currying, Lazy Evaluation, Lexical Closures, metacircular evaluators etc - which will make your friends come to the conclusion that:
  - ⊖ You are mentally unstable
  - ⊖ You are a genius
  - ⊖ You are both!

If your language allows you to write (or more correctly, *induces* you to write, or *motivates* you to write) programs which have the above properties, maybe, you can call your language a functional programming language. Note that it is possible to write C program in this style[17] - but the language does not motivate you to write programs that way, so you don't call C a functional language. I would be able to show you most of these features in the next few sections.

5 What is Functional Programming? What is Imperative Programming

[Note: DONT read this section now. Read it after you have finished with almost all other sections of this document. The issues discussed are really subtle! But then, no trouble in having a glance]

In the above section, I have given a lengthy description of what FP is. Now, let me give you a very brief description (from SICP):

> Programming without using assignment is called Functional Programming. Programming with extensive use of assignment is called Imperative Programming (what we normally do in C, Pascal).

Most of us are surprised when we suddenly realize that we know very little of something which we thought we knew very well. This is a good sign - it is the beginning of real wisdom. Most of our good students as well as all our teachers should sometimes have this experience - otherwise, they are doomed to

[14]This is a term which you frequently encounter when you study programming language theory. Other than sweetening up things a bit, sugar does not have too much use. Likewise, a feature which is added just for the sake of making things a bit more pleasant, and with no other objective, is sometimes called syntactic sugar.

[15]If this sounds like Greek - don't worry. I shall provide a simple explanation later.

[16]Don't take this seriously ;-)

[17]Well, almost. GNU C does have closures as an extension - implemented through trampoline code generated on the fly on the stack. If you believe you know something about programming, I challenge you to read and understand the paper which describes how this feature is implemented.

remain in a happy state of ignorance. Reading SICP is a nice prescription to deflate the ego of good programmers.

How many of us have given a little bit of thought to the humble assignment operator which we take for granted? Its time we did.

5.1 The substituition model

How do we understand the working of a lisp procedure, say:

```
(defun sum-of-squares (a b)
   (+ (square a) (square b)))
```

Assume that sqr is defined as:

```
(defun sqr (x) (* x x))
```

Let's say we are calling:

```
(sum-of-squares 2 3)
```

We can think of the Lisp interpreter as substituting the above expression with:

```
(+ (square 2) (square 3))
```

Now, we can again think of the Lisp interpreter as replacing the above expression by:

```
(+ (* 2 2 ) (* 3  3))
```

We note that we are simply replacing the function name with the body of the function, with the formal parameter names being substituted by the actual paramters. Similar is the case with functions like the 'factorial' which beginning Lisp programmers write to get acquainted with the language. Once we write a few Lisp programs, we start realizing that this substituition model is very simple to use and understand - it immediately gives us a clear picture of the working of the code - we also feel more sure about the correctness of the code. We are almost always able to argue about the correctness of the code in a 'mathematical' way - we are able to make up arguments of this sort -

> If our list is empty or if it contains just one element, our functions returns the same list. So our 'reverse' function works properly on an empty or one element list. Now let us suppose it works properly for a list A, that is R (A) = B. Now what about a new list R(PA)? It should be BP. It is, because we are simply appending list P to R(A). [We are talking of a reversal function which we will write later]

Because two invocations of the same function with the same argument always return the same value, we are also free from the duty of thinking about the past history of function invocations or the context in which our function is called. *In effect, we are able to reason rigorously about a Lisp program which is a collection of functions just as we reason about a mathematical system of equations.*

This is a very good thing, as it gives us the ability to actually PROVE that our programs are correct - think of the money and effort spent in testing programs - if, with a little bit of initial efforts, we are able to prove that our programs are correct, we would have perfectly working programs which never crash.

What is the peculiarity of functional programs which give them this nice property? Simple. When you call a procedure, you can think of the body of the procedure getting executed with all the formal argument names being UNIFORMLY replaced by the name of the arguments which you actually supply. Think of it like this. You have a mathematical function:

$$f(x) = (1+ x*x)\ (x + 1)\ (x - 1)$$

You want to compute f(2). You simply write:

$$(1+2*2)(2+1)(2-1)$$

What is wrong with this picture? Why doesn't Bill Gates and everybody else write functional programs and live in a 'software nirvana' of fast, always correct, crash proof programs?

The answer lies in the fact that Computer Programs are most often written to model real world phenomenon where 'state' is important. In such cases, *assignment* becomes a necessary evil and destroys all the nice 'mathematical' properties which our programs had, making it very difficult to provide rigorous proofs about their correctness. Once we introduce assignment in to our programs, the substituiton model of procedure evaluation becomes almost worthless. Why? Let's think of the C function:

```
int fun(int x)
{
  int p, q;
  q = x;
  x = x - 1;
  p = x;
  return p * q;
}
```

Let's say we are calling the function as fun(3). We see that it is now impossible to uniformly replace all occurrences of the symbol 'x' with the number 3. The substituition model has broken down. We are no longer able to think of a C function as a 'mathematical' function. But then, we note that 'fun' can be rewritten in the following way:

```
int fun(int x)
{
  return (x-1)*x;
}
```

And once again, we can think of fun as being a 'mathematical' function. The trouble is that this kind of rewrite is difficult in situations where our program models 'real world' concepts. Let's think of a program which models a banking account. We start with an initial amout of Rs.1000/-. We having a procedure called 'withdraw' which accepts the amount to withdraw and returns the current balance. We write it like this:

```
(setf current-balance 1000)
(defun withdraw (x)
   (setf current-balance
     (- current-balance x))
   current-balance)
```

Or, in C

```
int current_balance;
int withdraw(int x)
{
   current_balance = current_balance - x;
   return current_balance;
}
```

We note that assignment is an important part of this program - it helps us model the fact that the 'state' of our bank account is governed by the 'current_balance' and 'withdraw' makes mutations to that state. You will note that in writing any program where we are not just computing some mathematical function, the use of the assignment operator assumes vital significance. Note that in the above program, even though we are not altering the value of 'x', our function loses one nice 'mathematical' property - two invocations of 'withdraw' with the same argument do not yield the same value. When mathematicians see expressions like this:

```
f(x,y) = (1+xy) + g(x) + (f(y) + g(x))
```

they like to simplifiy it to:

```
f(x, y) = (1+xy) + 2g(x) + f(y)
```

We realize that it is difficult to apply such simplifications to an expression which contains the 'withdraw' procedure because two invocations of 'withdraw' return two different values.

Assignment introduces subtle problems with the sequencing of statements in our program. Let's think of an imperative definition of factorial:

```
int fact(int n)
{
   int f = 1, p = 1;
   while(p <= n) {
     f = f * p;
     p = p + 1;
   }
   return f;
}
```

What if we interchange the two lines in the body of the loop? We will have to very carefully think of the order of the statements. Such problems simply do not arise when the code is rewritten functionally.

To summarise, when we write programs 'functionally', without using assignments, our programs look like a bunch of mathematical functions whose correctness can be verified easily using formal arguments. Assignment spoils this beautiful structure. But it is often an unavoidable evil because modelling real world situations often necessitates its use.

6 Let's start

I can keep on writing like this for long - but you are getting bored. So let's look at some code! I can't resist the temptation to begin with some numerical programs from SICP. The original code, written in Scheme, has been modified to work under Common LISP.

6.1 Statutory Warning

If we start learning a functional language as our first programming language, then I feel that we would all like it. Unfortunately, we have been trained in C for some time - so we may find things a bit awkward, maybe, a wee bit difficult. Also, Common LISP (as opposed to Scheme), is a very complex language (Cltl2, which stand for Common Lisp, the Language, edition 2, is the standard reference and it runs into more than a thousand pages!). I am as much of a novice as you are - and we may find the ride not very joyous :-(

6.2 Free Common Lisp Implementations

There are two excellent Common Lisp implementations for Unices - one is CMU-CL and the other one is CLISP. We will be using CLISP. Just type *clisp* at the Unix prompt and we are ready for action.

6.3 Hello, World

Let's not break the 'hello world' tradition. We try

```
(print “hello world")
```

Print[18] is a function, but why are we calling it (print “hello world") rather than print("hello world"). This is the TIMTOWTDI principle in action - if there is more than one way to do it, then try all those ways - *(print “helloworld")* is a much more cleaner representation - think of a lisp expression like (+ 1 2) - we are simply calling a function + with two arguments, 1 and 2. In C, we would write this as 1+2. Lisp uses the same notation for invoking standard operators as well as functions - because operators are also functions! Don't get turned off by this.

The output generated by (print “helloworld") is

```
“hello world"

"hello world"
```

Why are we getting two strings? The LISP interpreter is sitting in a 'read-eval-print' loop; it reads an expression, evaluates it and prints the value returned by the expression. The 'print' function returns its argument - it also produces a 'side-effect', the side effect is the act of displaying the message "hello world" on the screen. So the first string is the 'side-effect' produced by the print function and the second string is the value returned by the function - which gets printed by the interpreter which is sitting in a 'read-eval-print' loop.

The behaviour changes when we run the code in batch mode - that is, we create a file named hello.lsp and run it by typing:

```
clisp hello.lsp
```

The LISP interpreter will not automatically print back the value of each and every expression it evaluates (when it is operating in batch mode) - something is printed on the screen only as:

1. A side effect of executing functions like format, print
2. A result of fatal errors.

[18]No case sensitivity!

So remember this - LISP is full of braces; when you open a brace, the first thing you put in it should be the name of a function. The LISP interpreter prints the result of evaluating the expression - only if you are running the program in interactive mode.

6.4 Arithmetic

What does the following program print?

```
(* (+ (* 2 3)
      (* 1 2))
   (- 2 4))
```

Look at the way the code is indented[19]. Even though there is a profusion of parenthesis, you soon get used to it. Try common arithmetic operations involving big and small integers and real numbers. Try out the functions *mod, gcd, max, min, or, and, not.* Note the peculiar behaviour of the logical operators. In LISP, only nil is supposed to be false. Everything else is true. The symbol 't' also stands for true. Try playing with the type-of function.

6.5 Defining Functions

When we learn a language like C, we spend considerable time discussing if statements, while, for etc and then only do we start discussing functions. But when we learn a functional language we are forced to look at function definitions within the first half an hour of study itself - we can't write meaningful code without defining functions.

Function definitions are extremely simple. Make sure that you understand the syntax properly.

(defun *function-name* (param1, param2, ... , paramN) *body*)

Let us try out our first function

(defun sqr (x) (* x x))

That doesn't look intimidating! Now how do you call this function?

(sqr 20)

Find out what happens when you call sqr without any argument and with more than one argument.

6.6 setf

The function definition is in fact binding the name sqr to a function. We can also bind names to values like this[20]:

(setf a 10)

a ; prints 10. Value of 'a' as a variable is 10.

(+ a 20) ; prints 30

(a) ; error - symbol a is not bound to a function

(defun a (x) x) ; a is now a function accepting one argumenting and returning the same value.

(a 20) ; evaluates to 20.

(+ a 20) ; evaluates to 40.

The above sequence shows that the symbol 'a' leads a double life - as a variable and as a function.

If you pursue this line of investigation further, you will end up in trouble - and it may be difficult for me to clear up the mess. It is better to avoid too much subtleties -but then, as Einstein (or some other fellow like him) said:

Make things as simple as possible, but no simpler.

So we will continue. Lisp is populated by symbols which act as variables, functions and then, macros and special forms. The symbol 'defun' and setf are macros. It is easy to understand why 'setf' can't be a function. Let us try:

(defun add (x y) (+ x y))

(add 1 2) ; OK, we get 3

(add 1 p) ; error, CLISP says that variable p has no value

[19] Vi does this automatically for you. You have to invoke it as: vim -l expr.lsp

[20] Note that ; is used to start a comment

Why is this error occurring? Simple. A function always evaluates its arguments - now, we have not assigned a value to ‘p’ - so its an error.

Now think of setf as a function. If it’s a function, it will always evaluate both its arguments. So, an expression of the form:

(setf a 1)

will not get evaluated because the symbol ‘a’ has no value! So setf can’t be a function. It’s a special form.

6.7 Some more simple functions

(defun add ()) ; no argument, no body.

(add) ; returns nil. Note that there is no explicit return statement. A function always returns a value.

(defun mul (x y) (* x y))

(mul 2 3) ; returns 6

(defun fun () 1 2 3)

(fun) ; returns 3. 1 and 2 and 3 are expressions - result of evaluating the last expression is returned.

(defun fun () (1 2)) ; error - the LISP interpreter knows that the body (1 2) is meaningless - 1 can’t be a function. So there is an error the moment you type this expression.

(defun fun () (m 1)) ; interpreter is not giving any error message - this tells you something about the dynamic nature of LISP. The symbol ‘m’ may not have any meaning when you define fun. But at the time of executing fun, m can have a value.

(fun) ; error - m is not a function

(defun m (x) x) ; define m as accepting a value x and returning the same value.

(fun) ; OK. returns 1

6.8 Condition checking

Let us try

(if 1 2 3)

We get 2. General form of the ‘if’ statement is:

(if expression then-form else-form)

Evaluate the conditional expression, if it is true, return the value of the next form, otherwise return the value of the other form (I call these forms then-form and else-form).

You should be thorough with the syntax of if. So let’s try our some variations.

(if 1) ; error. Too few paramters for special form. You observe that if too seems to behave like a function. It has got three arguments - if the value of the first argument is true, the value obtained by evaluating the second argument is returned, otherwise, the value obtained by evaluating the third argument is returned. Only catch is that the then and else forms need to be evaluated depending on the value of the conditional expression - a function always evaluates its arguments - so it is difficult to implement if as a simple function. But it is not impossible.

(if 2 3) ; OK. 2 is true so return 3.

(if 0 3); OK. 0 is also true, so return 3.

(if nil 3 4); OK. nil is false, so return 4.

(if nil 3); OK. else part is missing, so returns nil

(if (> 1 2) 2 3) ; OK, returns 3.

(if (> 3 2) nil (+ 4 5)) ; OK, returns nil

6.9 Defining my-min

Let us try to define a function called my-min:

```
;;;find minimum of 3 numbers

(defun my-min (a b c)
  (if (< a b)
         (if (< a c)
                a
                c)
```

```
              (if (< b c)
                     b
                     c)))
```

We may also take an easier route:

```
;;; An easier way would be

(defun my-min2 (a b)
  (if (< a b)
         a
         b))

(defun my-min3 (a b c)
  (my-min2 (my-min2 a b) c))
```

Again, note the way the code is indented.

6.10 Sequencing through 'progn'

Sometimes, we may need to evaluate more than one expression in the then or else part of an if statement. The special form 'progn' can be used to do sequencing. Here is a small example:

```
;;; demonstrating progn

(defun fun (x)
  (if (< x 20)
         (progn
                (print "cold")
                (+ x 100))
         (progn
                (print "hot")
                (- x 5))))
```

Note the syntax carefully. The sequencing operator may not be required everywhere. For example:

(defun fun () (progn (+ 1 2) (* 3 4)))

(fun) ; OK. Prints 12.

(defun fun () (+ 1 2) (* 3 4))

(fun) ; This too works fine. Prints 12. No 'progn' required.

6.11 A better 'if'

(cond (1 2) (2 3)) ; returns 2. 1 is true.

(cond (0 2) (2 3) (3 4)) ; returns 2. 0 is true.

(cond (nil 2) (0 2) (1 4)) ; returns 2. 0 is true, nil is false.

(cond ((< 1 2) 3) (0 4)) ; returns 3. 1 less than 2 is true.

(cond ((> 1 2) 3) ((= 3 4) 5) (t 6)) ; returns 6. t is always true. First two conditions false.

(cond ((> 1 2) 3) ((= 3 4) 5)) ; returns nil. All conditions false.

Here comes a small program:

```
;;; find absolute value
(defun abs (x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

The general form of 'cond' is:

(cond (pred1 ret1) (pred2 ret2) ... (predN retN))

Note that 'progn' is not required when using 'cond'. As an example:

(cond (1 (+ 2 3) (* 3 4)))

The predicate is 1 and the return value is computed by evaluating (+ 2 3) and (* 3 4) in sequence. 12 is returned.

6.12 Procedural Abstraction[21]

6.12.1 Miller's experiment

Dr.Miller, a psychologist, conducted numerous experements on human subjects. His objective was difficult - identify how much information a human being

[21] Which, Sussmann says, is 'an abstraction of a procedure'. Now what the hell is that?

can process (correlate, comprehend) simultaneously. The results were disheartening. Even the most intelligent could not process more than seven (plus or minus 2) chunks of information in parallel.

Software systems (also, biological systems, chemical systems) are incredibly complex. Then how do we human beings manage to study them? This is a fundamental question which many philosophers have tackled. There is only one solution - and that is abstraction.

Think of a novice car driver. The car is a complex mechanical system. Our driver insists that he will drive the car only after he understands its working completely. So he first enrols as an apprentice mechanic in a reputed repair shop. There he starts tinkering with the engine. He realizes that he could become a mechanic only when he understands the working of the IC engine. So he enrols as a diploma student in a local polytechnic - there he happens to read a book on IC engines which contained only a single equation - the diligent fellow realizes that to learn about IC engines, he has to learn thermodynamics - so he enrols for an engineering degree at GEC. At GEC, he reads a book on thermodynamics which had 10 equations in each page - the fellow now realizes that to learn thermodynamics he has to learn Physics - last heard of, he had obtained a PhD in Physics and Mathematics and was thinking of enrolling for a PhD in Philosophy and Logic - it is said that he still won't drive a car.

The story is not told in jest. It has very serious implications. Let us first look at a definition of abstraction:

> A simplified description or specification of a system that emphasizes some of the systems details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, atleast for the moment, immaterial or diversionary.

As far as the driver of an aeroplane is concerned, the cockpit and the instruments in it are an abstraction of the aircraft. Most drivers are satisfied with this abstraction, and do not attempt to go below that level, like our enterprising car driver did. As far as a little child is concerned, the cheap toy plane he holds in his hands is an abstraction of the real aeroplane - it has wings, a shiny body, tiny tires - it will even fly a little bit - in his imagination!

So how do we really comprehend things? We choose an abstraction which we can reasonably comprehend with our current experience and skills. Once we are comfortable with this abstraction, we try to dig a bit deeper and identify another 'level'. We go on digging till we get as much idea about the working of the system as we want.

6.12.2 Abstractions in software

Now look at the way I try to understand the working of a complex software system. I am a professional programmer with just about average skills - and still I don't find maintaining a large program (say 5 lakh lines of code) difficult. How?

That's the genius of the designer of the program. He has constructed it in such a way that I can view the system as layers upon layers of abstractions. What I see first is a long list of files having .c or .h extensions - from the names of the files themselves - I am able to form a mental model of the overall structure of the program - I have my first layer of abstraction. I look for a file by the name 'main.c'. I start reading it. I see that main calls a lot of functions - 'get_this(), do_that(), get_this_and_that()'. I look at these functions as black boxes, I assume that get_this() does what it is supposed to do. Do I see a function called 'ding_dong()'? The comment besides the function seems to be interesting. Let me look into that function! I start looking for the file which contains ding_dong(). My journey goes on....

What am I really doing? I am ignoring some parts of the system and I am zooming on some other part - I keep on repeating this process recursively. How is it that I am able to do this. Because the designer of the system has built it in such a way. How is it that the designer is able to build it this way? By studying the works of other master designers.

6.12.3 Procedures as abstraction mechanism

The procedure(or function) is a fundamental abstraction mechanism. The user of a procedure only needs to know what all things have to be passed to it and what it returns - he does not have to bother about *how* the procedure accomplishes its purpose. Thus, as far as a user is concerned:

1. A car is simply a mechanical system which runs when something called the accelarator is pushed, stops when the brake is pressed and turns when the steering wheel is turned. We may think of the break, steering wheel and accelerator as being the 'user interface' or abstraction of a car.

2. *sqr* is an object which consumes a number and produces a number as the result - with a certain relationship between the input and the ouput. How exactly sqr performs its function is a matter which the user is simply not concerned with - unless of course, he wishes to descend into another layer of abstraction - like our enterprising driver. The name, parameters as well as the return value of the procedure are it's user interface - it is what we view as a 'procedural abstraction'.

6.12.4 Abstraction and mathematics

Abstractions have an interesting property, they help us to see the unifying thread between disconnected entities. For a moment, go back in time and think of the way the ancient man's mind might have worked. The caveman sees a flock of birds, a bunch of rabbits, a group of trees, a set of pebbles - all seemingly disconnected entities. One bright fellow might have felt that the essential property of a bunch of birds can be summed up by asking: (a) what 'type' of birds are they and (b) how many are there? The question of 'how many' lead to the birth of counting, and thus the whole of mathematics while the question of type (or classification) lead to the development of the physical sciences.

Come back to more recent times. Why do you think many of the worlds biggest stock markets employ PhD's in Physics? Think of the movement of gaseous molecules - scientists study such systems by adopting a *mathematical abstraction* that the molecules of the gas are points in space which interact with each other in a non deterministic manner. A stock exchange, in its mathematical abstraction, is a complex non deterministic system - the same laws of statistical mechanics which are employed to study gases can be used to analyze the working of a financial exchange. Look at how mathematical abstractions help us *link together seemly disconnected entities*!

6.13 Back to LISP

That was a pretty long digression - we are back on track again. Let us try to define some simple functions in LISP.

6.13.1 Printing numbers

```
;;; Print numbers from a to b, inclusive
;;; of both a and b. Assume a less than b.

(defun print-num (a b)
  (if (> a b)
        nil
        (progn
               (print a)
               (print-num (+ a 1) b))))
```

How do we print numbers from a to b? Print a and then call print-num recursively until a becomes greater than b. Note that the function returns nil. This value is not at all significant. We are concerned with the side-effects of executing print-num rather than its return value.

Again, note the way the code is indented. Also, look at the name 'print-num'. We are using a minus sign in the variable name. This is OK in LISP. There are no restrictions on the symbols used in naming variables, functions etc.

6.13.2 Sum of all numbers from a to b

```
;;; Sum of all numbers from a to b. Assume
;;; a less than b, both a and b positive.
```

```
(defun sum-all (a b)
  (if (> a b)
      0
      (+ a (sum-all (+ a 1) b))))
```

What happens if a is negative? If b is negative? If both are negative? Write a generalized version of this function which will work for any value of a and b.

6.13.3 The Classical Factorial

We can now define the classical factorial function in LISP:

```
;;; Factorial

(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

6.14 Square roots by Newton's[22] Method

By this time, you should have got the 'flavour'[23] of LISP. Let us examine a typical problem decomposition.

Newton's method is simple. Suppose you wish to take the square root of a number x. Let us make a guess that the root is y. Now we check whether this guess is 'close-enough', ie, the difference between y*y and x should be very small. If it is not close enough, we 'improve' the guess, by taking the average of y and x/y.

[22]The algorithm is a special case of Newton's method - which is a general procedure for finding roots of equations. The square root algorithm itself was developed by Heron of Alexandria in the first century A.D.

[23]You might have realized that the three golden rules of Lisp programming are - (a) Think Functions (b)Think Functions and, (c) Think Functions. A LISP program should read like prose (or poetry) - you define small small functions to give 'names' to your ideas - you combine simple functions to get complex functions.... Try to read aloud the definition of sqrt-1 in this section - you will see what I mean.

```
;;; Square root by Newton's method

(defun good-enough? (guess x)
  (< (abs (- (* guess guess) x)) 0.001))

(defun average (x y) (/ (+ x y) 2))
(defun improve (guess x)
  (average guess (/ x guess)))

(defun sqrt-1 (guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-1 (improve guess x) x)))

(defun sqrt (x)
  (sqrt-1 (/ x 2) x))
```

Isn't the code elegant? Does it not reflect the way we think about the problem? It does - because it has been written by true masters - the authors of SICP.

Notice the way the functions are named. The question mark in the declaration of *good-enough?* does not mean anything in particular - it is simply to give the user the impression that the function is asking a question and expects a yes/no answer. Also, note the use of a procedure called sqrt-1. We define such functions often in LISP - sqrt-1 exists simply to facilitate the writing of an sqrt function which accepts only one argument.

A question. Let us define a function called *my-if:*

```
(defun my-if (predicate then else)
  (cond (predicate then)
  (t else)))
```

Now, we rewrite sqrt-1 to use my-if instead of if. What would happen?

6.14.1 Programming as language building

Programming can be thought of as a language building activity. How do you explain the square root algorithm in plain English (we are not thinking about the program - assume that we don't even know what programming is)? In normal conversation, we would say:

> Well, make a guess that y is root of x. Then check if that guess is good enough. If not, improve the guess. How do you check whether the guess is good enough? How do you improve it? Well, you can do such and such things...

Note that we are able to provide a very high level view of a complex activity by using a few 'terms'. Think of these terms as forming the vocabulary of a 'language'. If our audience wishes to know more, we will explain each 'term' by a combination of some other 'lower-level' terms (again, think of these terms as forming another, 'lower-level language'). Functions help us in creating new terminology to describe things. The writing of a program can thus be thought of as construction of a set of terms and thus a language (in the case of complex programs, layers of languages) which is convenient enough to describe sophisticated processes.

6.15 Procedures and the processes they generate

Some quotations. The first one is by a poet or philosopher - I don't know the name.

> *To err is human, to forgive is divine.*

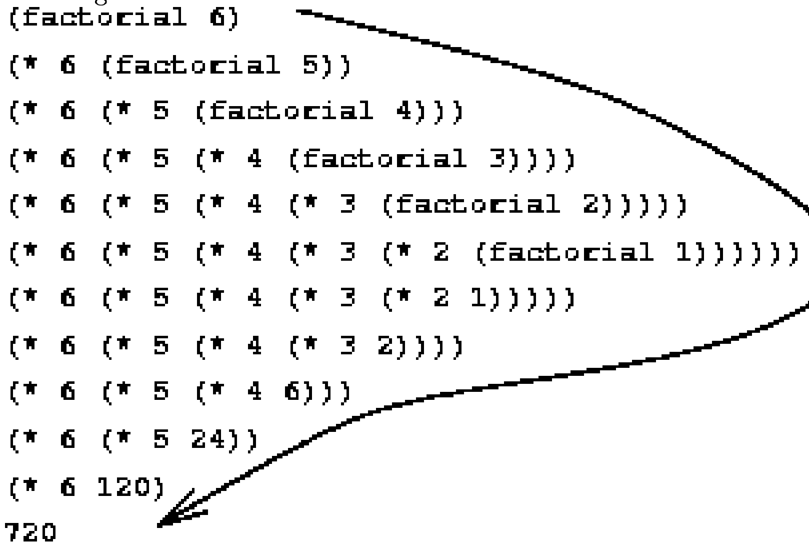
And the next one is by a Computer Scientist:

> *To iterate is human, to recurse is divine.*

We are going to examine some subtle issues here. If you find things flying above your head, relax, take a deep breath, and read again. If you still dont understand, skip the section - it won't stand in the way of your understanding LISP.

Look Figure 1 - it shows the way the evaluation of the classical factorial function takes place.

Figure 1: Factorial - the classical method



```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

You observe a sequence of expansions and contractions. We may loosely call this figure the 'process' generated by the linear recursive procedure for computing the factorial. Now let us look at an alternative method for computing the factorial, which is embodied in the program given below:

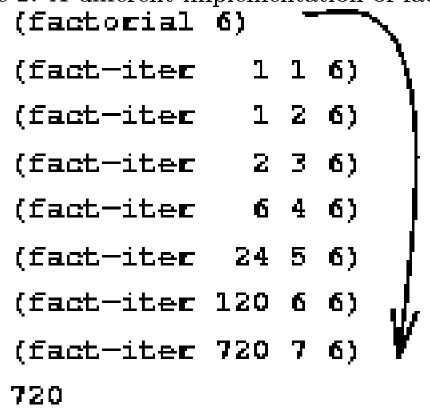
```
;;; Factorial - another implementation

(defun factorial (n)
  (fact-iter 1 1 n))

(defun fact-iter
(product counter max-count)
  (if (> counter max-count)
    product
    (fact-iter (* counter product)
               (+ counter 1)
               max-count)))
```

Let us view the 'process' generated by this procedure - look at Figure 2

Figure 2: A different implementation of factorial



```
(factorial 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

You observe that the expansion-contraction, which was clearly visible in the previous case, simply does not exist. Surely, this is something very interesting.

Consider the first process. The expansion occurs as the process builds up a chain of *deferred* operations (in this case, a chain of multiplications). The contraction occurs as the operations are actually performed. This type of process, characterized by a chain of deferred operations, is called a *recursive process.* Carrying out this process requires that the interpreter keep track of the operations to be performed later on.

By contrast, the second process does not grow or shrink. At each step, all we need to keep track of, for any *n,* are the current values of the variables product, counter and max-count. We call such a process an *iterative process.* In general, an iterative process is one whose state can be summarized by a fixed number of state variables together with a fixed rule that describes how the state variables should be updated as the process moves from state to state.

The contrast between the two processes can be seen in another way. In the iterative case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to resume the computation is to supply the interpreter with the values of the three variables. Not so with the recursive process. In this case, there is some additional 'hidden information' maintained by the interpreter and not contained in program variables, which indicates where the process is in negotiating the chain of deferred operations. The longer the chain, more information must be maintained.

In contrasting iteration and recursion, we must be careful not to confuse the notion of a *recursive process* with the notion of a *recursive procedure.* When we say that a procedure is recursive, we are referring to the syntactic fact that the procedure definition refers to itself. But when we describe a process as following a recursive pattern, we are describing how the process evolves - not how the procedure is defined. It may be disturbing that we describe a recursive procedure as generating an interative process - but its true. The state of the process is captured by the three variables - the interpreter need not keep track of anything else.

One reason that the distinction between process and procedure may be confusing is that most implementation of languages like Pascal and C are designed in such a way that the interpretation of any recursive procedure consumes an amount of memory that grows with the number of recursive calls - even when the process described is by nature iterative. As a consequence, these languages can describe iterative processes only by resorting to special purpose looping constructs like while and for. Dialects of LISP, like Scheme, are *tail recursive,* ie, Scheme always interprets an iterative process in constant space[24] - even when the iterative process is described by a recursive procedure. So looping constructs, even if they are available, are merely syntactic sugar.

Here are some exercises:

Each of the following two procedures (add1 and add2) defines a method for adding two positive integers in terms of the procedure *inc* which increments its argument by one and *dec* which decrements its argument by one. Your job is to clearly trace through the calls (add 4 5) in both cases and identify whether the processes are iterative or recursive.

```
;;; addition
```

[24]You note that for a tail recursive process, the recursion stack is not at all necessary - I shall try to explain this in the class - don't worry if you don't understand what I say.

```
(defun inc (x) (+ x 1))
(defun dec (x) (- x 1))
(defun add1 (a b)
  (if (= a 0)
        b
        (inc (add1 (dec a) b))))

(defun add2 (a b)
  (if (= a 0)
        b
        (add2 (dec a) (inc b))))
```

Define a function for computing the N'th fibonacci number. Write functions which generate linear recursive as well as iterative processes.

Define a 'power' function which computes 'a to the power of b'. Assume 'a' and 'b' are positive integers and 'a' is non-zero

Define a function for performing integer multiplication using repeated addition.

Define a function which computes the greatest common divisor[25] of two integers.

6.15.1 The Ackermann's function

Given below is a LISP function which computes the *Ackermann's function.*

```
;;; Ackermann's function

(defun A (x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (t (A (- x 1)
              (A x (- y 1))))))
```

Your job is to find out the values of the following expressions:

1. (A 1 10)
2. (A 2 4)
3. (A 3 3)

[25]Use the Euclid's algorithm, which says that GCD(a, b) is a if b is zero; else it is GCD(b, a mod b.

6.15.2 Testing for primality

Let us look the LISP implementation of a function which tests its argument for primality. Read the program carefully and try to understand the decomposition clearly. You should emulate this style in all the LISP programs which you write:

```
;;; Testing for primality

(defun square (x) (* x x))

(defun find-divisor (n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n)
                   test-divisor)
        (t (find-divisor
                 n
                 (+ test-divisor 1)))))

(defun divides? (a b)
  (= (mod b a) 0))

(defun smallest-divisor (n)
  (find-divisor n 2))

(defun prime? (n)
  (= (smallest-divisor n) n))
```

6.16 Formulating Abstractions with Higher order Procedures

We have seen that procedures are, in effect, abstractions which describe compound operations on numbers independent of the particular numbers. For example, when we write:

```
(defun cube (x) (* x x x))
```

we are not talking of the cube of a particular number, but rather about a method of obtaining cube of any number. We could always get along without ever defining this procedure. We could always write:

```
(* 3 3 3)
(* x x x)
(* y y y)
```

But this places us at a serious disadvantage, forcing us to always work at the level of the particular operations which happen to be primitives in the language rather than in terms of higher level operations. Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing. One of the things we would demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Procedures provide us this facility.

Yet, even in numerical processing, we would be severly limited in our ability to create abstractions if our procedures accept only numbers as arguments. Often, the same programming pattern will have to be used with a number of procedures. To express such patterns, we will have to define procedures which accept procedures as argument and return procedures as values. Procedures that manipulate procedures are called *higher order procedures.* They are used extensively in LISP - vastly increasing the expressive power of the language.

6.16.1 The 'funcall' function[26]

```
(defun sqr (x) (* x x))
(defun fun (x y) (funcall x y))
(fun #'sqr 20) ; prints 400
```

We will explore some new syntax elements. First, let us look at the builtin function *funcall.* It's first argument should always be a function. Funcall will execute the function supplied as its first argument, passing as arguments to that function all subsequent arguments which funcall itself had obtained from its caller. In this case, funcall calls the function 'x' with argument 'y'.

Now lets look at the way fun is called. The first argument to fun is *#'sqr.* Because the same Lisp symbol may be interpreted differently in different contexts, we write #'sqr to specify that we are passing to fun a symbol which should be interpreted as a function.

[26]You need not write #'sqr. 'sqr seems to be sufficient. I have no clear idea of the difference between the two notations. [This footnote added as part of ver 2.0 of my notes]

6.16.2 Formulating the higher order function *Sigma*

Consider the following three procedures. The first computes the sum of the integers a through b:

```
;;; Sum of numbers, a through b

(defun sum-integers (a b)
  (if (> a b)
        0
      (+ a (sum-integers(+ a 1)
                         b))))
```

The second computes sum of the cubes of integers in the given range:

```
;;; Sum of cubes of numbers a to b

(defun cube (x) (* x x x))
(defun sum-cubes (a b)
  (if (> a b)
    0
    (+ (cube a)
       (sum-cubes (+ a 1) b))))
```

The third computes the sum of terms of a series:

1/(1*3) + 1/(5*7) + 1/(9*11) + ...

which converges to $\Pi/8$ very slowly.

```
(defun pi-sum (a b)
  (if (> a b)
    0
    (+ (/ 1
          (* a (+ a 2)))
       (pi-sum (+ a 4) b))))
```

These three procedure share a common pattern - they are for the most part identical, differing only in the name of the procedure, the function of 'a' used to compute the term to be added and the function that provides the next value of 'a'. The prescence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Indeed, mathematicians long ago identified the abstraction of *summation of a series* and invented the *sigma* notation, for example:

> *Sigma{F(n)} = {F(a) + F(b) + ... F(n)}*
> *{n varies from a to b}*

The power of the sigma notation is that it allows the mathematician to deal with the concept of summation itself rather than with particular sums - for example, to formulate general results about sums which are independent of the series being summed.

Similarly, as program designers, we would like our language to be powerful enough so that we can write a procedure which expresses the concept of summation itself rather than only procedures which find out independent sums. We can do so very easily in LISP.

```
;;; Sigma - a higher order procedure

(defun sigma (term a next b)
  (if (> a b)
    0
    (+ (funcall term a)
       (sigma term
              (funcall next a)
              next
              b))))

(defun cube (x) (* x x x))
(defun inc (x) (+ x 1))
(defun pi-term (x)
  (/ 1.0 (* x (+ x 2))))
(defun pi-next (x)
  (+ x 4))

(defun sum-cubes (a b)
  (sigma #'cube a #'inc b))

(defun sum-integers (a b)
  (sigma #'identity a #'inc b))
;identity is a builtin function
;which simply returns the value
;of its argument.
```

```
(defun pi-sum (a b)
  (sigma #'pi-term a #'pi-next b))
```

Take some time and digest the example completely. You can try solving the following exercise if you are interested.

The sigma procedure is only the simplest of a vast number of similar abstractions that can be captured as higher order procedures. Analogous to sigma, write a 'product' procedure which would compute the product of a sequence of numbers. Use 'product' to define the factorial function.

Show that 'sigma' and 'product' are both special cases of a still more general notion called 'accumulate' that combines a collection of terms using some general accumulation function.

(accumulate combiner null-value term a next b)

Accumulate takes as argument the same term and range specifications like sigma or product, together with a combiner procedure (of two arguments) that specifies how the current term is to be combined with the other term - it also accepts an argument called null-value which specifies what base value to use when the terms run out.

6.16.3 Using Lambda to construct procedures

When you code in C, you use 'malloc' to get blocks of memory which are basically nameless (anonymous). The block is accessed using a pointer which stores its starting address. Can you imagine something like a 'malloc for functions'?[27]

LISP is sometimes quite beyond our imagination. So don't take pains to imagine! Again, let me tell you that we are going to see something subtle.

[27]There is no connection between malloc and lambda - I got the idea for this analogy by thinking this way - malloc gives you an anonymous block of memory and lambda gives you a nameless procedure.

```
;;; demonstrating lambda
(defun fun (x y) (funcall x y))
(print (fun #'(lambda (x) (* x x)) 20))
```

Fun is a higher order function. Instead of defining a function called 'sqr' and passing it as the first argument to fun, we use the notation *#'(lambda (x) (* x x))*. What does *lambda* do? It is generating, on the fly, a procedure which accepts one argument and returns its square. When you write LISP code, you will be required to write lots of little functions which are passed around as arguments to other functions. Lambda frees us from the burden of defining such functions - we can straight away write a lambda expression at a place which expects to get filled by a function. We will see some applications of lambda[28] later on.

Let us suppose we wish to write a function

$$f(x,y) = x(1+xy)^2+y(1-y)+(1+xy)(1-y)$$

It would be simpler to express this as:

$$a = (1 + xy)$$
$$b = (1 - y)$$
$$f(x,y) = xa^2 + yb + ab$$

Look at the way the following function is written:

```
(defun square (x) (* x x))
(defun fun (x y)
  (funcall #'(lambda (a b)
               (+ (* x (square a))
               (* y b)
               (* a b)))
           (+ 1 (* x y))
           (- 1 y)))
```

[28]The name 'Lambda' comes from Alonzo Churh's Lambda Calculus - a calculus of functions. Church proved that the notion of a function is extremely powerful - using it, one can express any computation.

Can you make out the way it is working? Withing fun, we are defining a nameless procedure using lambda - this procedure accepts two arguments 'a' and 'b'. We are calling this procedure with the values 1+xy and 1-y, which means within the body of the procedure, 'a' would get the value '1+xy' and 'b' would get the value '1-y'.

If you think a little bit, you realize that we are creating two 'local variables' *a* and *b* in a rather roundabout manner. There is a much more straightforward procedure, and it involves the use of the 'let' construct.

6.16.4 Using *LET* to create local variables

The format of the 'let' special form is:

```
(let ( (var1 expr1)
       (var2 expr2)
       ....
       ....
       (varN exprN) )
     body-of-let )
```

Please study the format carefully. Let us look at some examples:

```
(let ((a 1)) (+ a 1)) ; prints 2
(let ((a 1) (b 2)) (+ a b)) ; prints 3
(setf a 10) ; a global variable a
(let ((a 2)) (+ a 3)) ; prints 5.
Local 'a' shadows global a.
(let ((a 3)) (setf a 20)); changes
only local 'a'
(print a) ; Global a remains as 10.
```

Let us define a function using let:

```
;;; Use of let

(defun fun (x y)
  (let ((a (+ x y))
       (b (- x y)))
    (* a b)))
```

We are finding out (x+y)*(x-y). Try to rewrite the function in the previous section using let. Please note the 'let' special form requires a body, and only in that body are the variable bindings 'a' and 'b' visible. The function *fun* returns the value returned by the body of let.

6.16.5 Some more exercises

Define a function which returns the reverse of the number supplied as its argument.

Define a function which prints a number in binary.

Define a higher order function which performs numerical integration using the simple area-summing procedure.

7 The Common Lisp Programming Environment[29]

You have to know something about your programming environment. You are using CLISP, written mostly by *Dr.Bruno Haible*. Though this implementation lacks the polish of many commercial distributions, it is fairly sophisticated and quite usable. The CLISP interpreter is invoked by typing *clisp* at the Unix prompt.

You can type in simple programs at the interpreter prompt itself - CLISP will execute a read-eval-print cycle and the value returned by the form will be printed on the screen. Now suppose you have a set of function definitions (say, sqr, cube etc) stored in a file called 'first.lsp'. You type

(load "first.lsp")[30]

and all the functions defined in this file can now be used at the interpreter prompt. A complete CLISP program, which contains function definitions as well as invocations of those functions could be executed by typing

clisp myprog.lsp

at the Unix prompt.

I will not spend too much time discussing the environment. Here are some simple exercises which you can do. You may refer the Implementation Notes (/usr/doc/clisp/impnotes.html)[31].

Find out what the function *lisp-room* does. Try out (lisp-space (make-array 100))

Find out what the function *room* does. What is meant by GC?

Find out the purpose of the *trace* function. *Trace* is a very useful tool. Define a factorial function. Before calling it, just type (trace fact). Now call the function and see what happens.

Look at the CLISP prompt after entering the expression (/ 1 0). Any change to the prompt? Why are we seeing a different prompt?

Define a buggy procedure:

```
(defun bug (a b)
   (/ a b) (bug a (- b 1)))
```

Calling the function

(bug 2 3)

will yield a run time error (division by zero). Your interpreter prompt changes. Just type

backtrace-4

See what happens. You can also try out:

backtrace-5

[29]The interested student may read /usr/doc/clisp/impnotes.html for a thorough understanding of the CLISP environment.

[30]CLISP can compile your code into platform independent intermediate code. This is done using the compile-file function. Find out how (and why) this is sometimes done.

[31]Some of you may not be knowing how to read HTML documents. You can use a browser program like lynx to do it. On my machines, you can simply type 'ly a.html'. At home, you may type 'lynx a.html'.

8 The List Data Structure

LISP stands for List Processing. The List is a very powerful data structure (one may argue that everything else can be constructed out of it). This section, hopefully, will make you a master list manipulator - becoming one is essential for the success of a LISP programmer.

8.1 Atoms and Lists

LISP considers an indivisible entity to be an atom. Thus 1, 10, 23.4 are atoms. Also, the symbols FOO, Baz2, Ding-3, dong etc are also atoms.

Lists can contain atoms as well as other LISTS. The empty list is denoted by

```
()
```

It is also given another representation, which is the symbol NIL. The empty list is considered to be false while everything else is considered to be true.

Let us type an atom. We enter:

```
FOO
```

at the interpreter prompt. But the interpreter gets angry and says that FOO does not have a value. So we type

```
'FOO
```

The interpreter is happy. It simply prints back the symbol FOO. What is the quote symbol doing? It tells the interpreter to treat FOO merely as a symbol and not attempt evaluating it.

Now, how do you build a list. Let's try:

```
(1 2 cat dog 3) ; error!
```

The interpreter tells us that 1 is not a function name. This happens because the interpreter thinks that 1 is a function and is to be applied on the arguments 2, cat, dog and 3[32] - which is not what we meant. Now let's try:

```
'(1 2 cat dog 3) ; no trouble
```

The interpreter is happy and he echoes back the sequence which we have typed. What is happening here? Simple. The interpreter now understands that (1 2 cat dog 3) is a list - it is not a function and a sequence of arguments. We say that *quoting* prevents evaluation - the list remains as a list. We may also achieve the same effect by using the special form *quote.*

```
(quote (1 2 cat dog 3))
```

But the symbol is used more frequently.

We can assign a symbolic name to a list:

```
(setf colors (red green blue))
```

8.2 List Handling Functions

LISP has some weird sounding primitives to manipulate lists. The three basic functions are:

⊕ CAR

⊕ CDR

⊕ CONS[33]

Here are some examples:

```
(car '(1 2 3)) ; 1
(cdr '(1 2 3)) ; (2 3)
(cons 1 '(2 3)) ; (1 2 3)
(setf triple '(1 2 3))
(cons (car triple) (cdr triple))
;(1 2 3)
```

For the present[34] - we will assume that:

CAR accepts a list as argument and returns the first element in the list

[32]This is not applicable to the empty list. The empty list is always interpreted as an empty list.

[33]The names car and cdr came from 'contents of address register' and 'contents of data register'.

[34]The future holds more promise - we will see how LISP implements list structures - we will then change some of our old notions as to how the elementary list handling functions behave.

CDR accepts a list as argument and returns the part of the list excluding the first element.

CONS The second argument of CONS is a LIST[35] and the first argument can be a list or an atom. CONS simply returns a new list whose first element is the first argument to CONS and all other elements are those from the second argument.

Please note that all these list manipulation functions are non-destructive. That is, they return a new list but never modify their argument(s). Here are some other examples:

```
(cons 1 nil) ; we get (1)
(cons 1 ()) ; same effect
; nil and () are the same.
(cons nil nil) ; we get (NIL)
(cons 1 (cons 2 nil)) ; we get (1 2)
(cons '(1) '(2 3)) ; we get ((1) 2 3)
(car 1) ; error 1 is not a list
(cdr 2) ; error 2 is not a list
```

Another LIST manipulation function is APPEND. Here is APPEND in action:

```
(append '(1 2) '(3 4))
; we get (1 2 3 4)
```

The first argument of append should always be a list and the second argument, for the present, should also be a list. Please note the difference between

```
(append '(1 4) '(2 3)); we get (1 4 2 3)
```

and

```
(cons '(1 4) '(2 3)) ; we get ((1 4) 2 3)
```

very carefully.

[35]This need not be true. We will see why later. As of now, we will not try to understand what happens when the second argument is not a list.

8.2.1 Some more list handling functions

```
(setf p '(1 2 3 4))
(cadr p) ;we get 2
(car (cdr p)) ; we get 2
(first p) ; 1
(second p) ; 2
(rest p) ; (2 3 4)
(last p) ; (4)[36]
(cdar p) ; error. Why?
;because, (cdar p) is same as
(cdr (car p)) ; (car p) is not
; a list.
(reverse p) ; returns reverse of p
; but p is not changed.
(setf p (reverse p)) ; this will
;reverse p.
;or we can try
(nreverse p) ; this returns the
reverse of p, it also changes p.
(setf p '(1 2))
(setf q '(3 4))
(nconc p q) ; concatenation of p
; and q. But p is changed!
```

8.3 Equality

Checking whether two things are equal is a somewhat subtle issue in LISP. There are three predefined functions for testing equality:

- ⊕ EQ
- ⊕ EQUAL
- ⊕ EQL

Let us try

```
(eq 'foo 'foo)
```

Lisp returns T, which stands for true[37].. The symbol 'foo may be represented internally by a block of memory locations. The second symbol 'foo also may be represented by a block of memory locations. EQ returns true if and only if both these memory locations

[36]Be careful. It's a list.
[37]Any symbol other than nil or () is supposed to be true.

are the same - ie, the first symbol 'foo and the second symbol 'foo are represented by the same blocks[38] of memory[39]. If not, eq does not return true. Lets try

```
(eq '(1 2) '(1 2))
```

The interpreter returns NIL. Why? Whenever LISP sees a literal representation of a list, it simply creates a new list[40] and returns its address. So even though the lists (1 2) and (1 2) are equal in the sense both contain the same data, they do not refer to the same objects. Now let us try:

```
(equal'(1 2) '(1 2))
```

The interpreter says T. You can make use of *equal* to check whether two objects contain the same information - irrespective of whether they are one and the same or not.

Just to reinforce the idea - suppose we have a cloning machine - I create a clone of myself - now I try *equal* on myself and my clone, *equal* says that both are the same. But I try the same procedure with *eq* - it says that we are not the same!

Deciding when objects are equal in the eyes of eq may be difficult - I also suspect that the behaviour of eq may be implementation dependent. When you write code, make use of *equal.*

The third function, *EQL,* is again confusing. EQL returns true if both its arguments are equal in the eyes of EQ, or if both of them are the same NUMBER. This means that the Common Lisp standard does not guarantee that the same numbers are equal in the eyes of eq - even though both CLISP and CMU-CL do not behave that way. Again don't get confused. You won't be required to use eql in your programs. Just stick to equal.

[38]That is, internally, there are no two objects, but only one!

[39]What I am presenting here is a very simplistic explanation - the description is not entirely correct - but it is logically consistent. It is based on what is presented by Wade L Hennesy, Stanford graduate and LISP compiler writer.

[40]I am unable to verify this on the CLISP system. Somebody willing to read the CLISP source and find out exactly how LISP behaves when it sees a list literal?

9 Simple Programs which manipulate lists

We will now examine some simple programs which manipulate lists. You should try to rewrite all these functions in different ways (maybe, make them simpler or more difficult or change non tail recursive code to tail recursive code or whatever. Also, trace the execution manually and using the 'trace' facility available in clisp). I DO NOT GUARANTEEE THAT THE CODE IS BUG FREE. Find some bugs and win my admiration!

9.1 Length

There is a built-in function called length which computes the length of a list:

```
(length '(1 (2 3) 4)) ; we get 3
```

Here is a function my-length which does the same:

```
;;; compute length of a list

(defun my-len (a)
  (if (equal a nil)
    0
    (+ 1 (my-len (cdr a)))))
```

Checking for an empty list is so common that we can write:

```
(null a); true if a is nil
```

instead of (equal a nil).

9.2 Append

Let's define a my-append function.

```
;;; my-append

(defun my-append (a b)
  (if (null a)
    b
    (cons (car a)
          (my-append
           (cdr a)
            b))))
```

9.3 Reverse

```
;;; reverse

(defun my-rev (a)
  (if (null a)
    nil
    (append (my-rev (cdr a))
            (list[41] (car a)))))
```

9.4 Same Length

Write a function same-length? which expects two lists as arguments and returns T if both lists contain the same number of elements or nil otherwise. You should not make use of the *length* function in your program.

```
;;; same-length

(defun both-null (a b)
  (and (null a) (null b)))

(defun both-nonnull (a b)
  (and (not (null a))
       (not (null b))))

(defun same-length? (a b)
  (cond ((both-null a b) t)
  ((both-nonnull a b)
   (same-length? (cdr a) (cdr b)))
  (t nil)))
```

9.5 Count-occurrences

Write a function count-occurrences which counts[42] how many times an atom occurs in a list.

```
;;; count-occurrences

(defun my-count (a l)
  (cond ((null l) 0)
        ((equal a (car l))
         (+ 1 (my-count a (cdr l))))
        (t (my-count a (cdr l)))))
```

Is there any difference between my-count and the builtin function count? What would be the implication of using eq instead of equal?

9.6 Common Tail

Write a function called 'common-tail' which expects two lists of any length and returns the longest common sublist which comes at the end of both lists, for example:

```
(tail '(1 2) '(3 4 1 2))
; we get '(1 2)
(tail '(1 2 3) '(3 4 2 1 3))
; we get '(3)
```

Here is the code.

```
;;; common tail
(defun same? (a b)
  (equal (car a) (car b)))
(defun tail-1 (a b)
  (cond ((null a) nil)
        ((null b) nil)
        ((not (same? a b)) nil)
        (t (cons (car a) (tail-1 (cdr a) (cdr b))))))
(defun tail (a b)
  (reverse (tail-1 (reverse a)
                   (reverse b))))
```

9.7 Merge[43] two sorted lists

```
;;; merge

(defun larger (a b)
  (if (> (car a) (car b))
    a
    b))
```

[41] (list 1 2 3) is '(1 2 3)

[42] There is a standard function called which does this. Experment with it.

[43] There is a standard merge function in LISP. Experiment with it.

```
(defun smaller (a b)
  (if (< (car a) (car b))
    a
    b))
(defun same (a b)
  (equal (car a) (car b))
 (defun my-merge (a b)
  (cond ((null a) b)
        ((null b) a)
        ((same a b) (cons (car a)
                          (my-merge
                            (cdr a) b)))
        (t(cons (car (smaller a b))
                (my-merge
                   (cdr (smaller a b))
                   (larger a b))))))
```

9.8 Position[44]

Write a function which returns the position of an atom within a list, or NIL if the atom does not appear within the list.

```
;;; Position

(defun pos-1 (a l c)
  (cond ((null l) nil)
        ((equal a (car l)) c)
        (t (pos-1 a (cdr l) (+ c 1)))))

(defun pos (a l)
  (pos-1 a l 0))
```

9.9 Enumerate

Write a function enumerate which accepts a starting number, an increment and a count[45] as arguments and returns the corresponding list of numbers.

```
;;; Enumerate

(defun enum (a inc c)
  (if (= c 0)
      nil
      (cons a
      (enum (+ a inc) inc (- c 1)))))
```

[44]There is a builtin function called position. Try it.

[45]There is a standard function called 'count'. Try it out.

9.10 Flatten a list

Flattening is a 'deep operation'[46] on a list. For example:

```
(flatten '(1 (2 3) ((4))))
```

Result is

```
'(1 2 3 4)
```

Such deep operations may sometimes be tricky. So try to understand the code below clearly. Trace its working, both by hand and by using the CLISP trace function.

```
;;; Flatten a list

(defun flatten (a)
  (cond
    ((null a) nil)
    ((atom (car a))
     (cons (car a)
           (flatten (cdr a))))
    (t (append (flatten (car a))
               (flatten (cdr a))))))
```

9.11 Deep Reversal

I want a reverse function which works like this:

```
(rev '(1 (2 3) 4))
```

Should give the answer:

```
'(4 (3 2) 1)
```

Note that the whole outer list is reversed - also, each of the inner lists are also recursively reversed. The code is given below. Study it carefully. I will be explaining the code in the class using a modified recursion diagram - which should make it easy for you to analyze arbitarily complex lisp programs.

[46]As opposed to a 'shallow operation'

```
;;; deep reverse

(defun my-rev (a)
  (cond
    ((null a) nil)
    ((atom (car a)) (append
                      (my-rev (cdr a))
                      (list (car a))))
    (t (append
         (my-rev (cdr a))
         (list (my-rev (car a)))))))
```

You should attempt to understand the working of each and every function give above as thoroughly as possible. Note that what you gain is only a 'minimal' understanding of LISP.

9.12 Finding out nesting depth

An atom has depth zero:

```
(depth 1 0) ; we get 0
```

Other examples:

```
(depth '(2) 0) ; we get 1
(depth '(a (4)) 0) ; we get 2
```

And so on.

You should try tracing the code below - check whether it is working properly in all cases. You should spend some time drawing the recursion diagram - it will be time well spent.

```
;;; compute nesting depth

(defun depth (a d)
  (if (atom a)
    d
    (max (depth (car a) (+ d 1))
         (depth (cdr a) d))))
```

10 Structures

Creating a structure is a fairly easy job:

```
;;; building a structure

(defstruct point x y)
(setf p (make-point))
(setf (point-x p) 10)
(setf (point-y p) 20)
(print (point-x p))
(print (point-y p))
```

You have to understand the meaning of defstruct clearly. The line

```
(defstruct point x y)
```

results in three functions being created automatically:

1. make-point, which, when invoked builds a 'structure' which contains two fields 'x' and 'y', both fields having the value 'nil'
2. point-x, an accessor function, which can be used for extracting the value of the field 'x' of the structure.
3. point-y, another accessor function, which can be used for extracting the value of the field 'y' of the structure[47].

With this much of an understanding, you should be able to write a program which constructs a binary search tree. Note that you need not use structures to represent trees - lists are much more general mechanisms.

10.1 Building a binary search tree

Lets first create a structure to represent a node of the tree:

[47]When you write code in C, you write struct point p, which creates a variable p of type struct point. Instead, in LISP, you write (setf p (make-point)). In C, to access a field x, you will write p.x wheras in LISP, you will write (point-x p). You should realize that different languages employ different notations to do the same thing - an experienced programmer should be able to tide over this notational-difference barrier very easily - in fact, you will be called an experienced programmer only when you are able to do so!

```
(defstruct node dat left right)
```

Let's write a function *new*[48]:

```
(defun new (dat) (make-node :dat dat))
```

What is the meaning of (make-node :dat dat)? If we call (new 10), dat will be 10. The notation *:dat dat* simply initializes the dat field of the structure to the value 10. Similarly, the notation *:left nil* will initialize the field *left* of the structure to the value nil. But this initialization is not necessary because fields by default get initialized to nil.

Now we are ready to write a tree-append function which is modelled after our recursive tree building function in C. You will find the code to be simple and clear.

```
;;; Binary search tree

(defun tappend (h n)
  (cond
    ((equal h nil) n)
    ((< (node-dat n) (node-dat h))
     (setf (node-left h)
           (tappend (node-left h) n))
     h)
    ((> (node-dat n) (node-dat h))
     (setf (node-right h)
           (tappend (node-right h) n))
     h)
    (t (error[49]
         "duplicates not allowed"))))
```

Now let us write an inorder traversal function:

```
;;; inorder traversal

(defun inorder (tree)
  (if (null tree)
    nil
    (progn
      (inorder (node-left tree))
      (format
        t "data = ~A~%"
        (node-dat tree))
      (inorder (node-right tree)))))
```

[48]You observe that I am simply translating my standard C routines for tree building to LISP.

[49]The error function will immediately terminate execution of the program and print the message given as its argument.

The only thing worth mentioning about the program is the use of *format,* which is somewhat like *printf* in C. The first argument is t, which stands for the *terminal stream.* The second argument is a format string. Within the format string , ~A[50] gets replaced by the LISP representation of the next object given as argument while ~% gets replaced by a newline.

10.2 Deleting an element

First, read through the C program which performs deletion on a binary search tree. Then read the LISP code. You will not findy any great difference between the two.

Note that we have two functions, one called *deltree* and the other one called *rdeltree.* We will invoke *rdeltree* like this:

```
(setf h (rdeltree h 50))
```

rdeltree will first check whether we are requesting the deletion of the root node. If that is the case, this request will be handled as a special case. Otherwise, *deltree* will be called. *Deltree* always expects the node to be removed to be a non-root node.

```
;;; bst deletion

; first, get address of
; node containing dat
(defun getnode (h dat)
  (cond
   ((null h) nil)
   ((= (node-dat h) dat)
    h)
   ((< (node-dat h) dat)
    (getnode (node-right h) dat))
```

[50]If you want to print a number in hex, use ~X.

```
   ((> (node-dat h) dat)
    (getnode (node-left h) dat))))

(defun lchild? (p c)
  (eq (node-left p) c))
(defun rchild? (p c)
  (eq (node-right p) c))
(defun child? (p c)
  (or (lchild? p c) (rchild? p c)))

(defun largest (h)
  (if (null (node-right h))
   h
   (largest (node-right h))))

(defun node-type (h)
  (cond ((and
       (null (node-left h))
       (null (node-right h))
       )
      'LEAF)
       ((and
       (not
        (null (node-left h)))
       (not
        (null (node-right h)))
       )
      'BOTH)
      ((and
        (not
         (null (node-left h)))
        (null (node-right h)))
      'LSUB)
      ((and
        (null (node-left h))
        (not
         (null (node-right h))))
      'RSUB)))


(defun getparent (h c)
  (cond
   ((null h) nil)
   ((child? h c) h)
   ((< (node-dat h) (node-dat c))
    (getparent (node-right h) c))
   ((> (node-dat h) (node-dat c))
    (getparent (node-left h) c))))


(defun deltree (h dat)
  (let*[51]
   ((c (getnode h dat))
    (p (getparent h c)))
   (cond ((equal (node-type c)
            'LEAF)
        (if (lchild? p c)
         (setf (node-left p) nil)
         (setf (node-right p) nil)
         ))
       ((equal (node-type c)
            'LSUB)
        (if (lchild? p c)
         (setf (node-left p)
             (node-left c))
         (setf (node-right p)
             (node-left c))))
       ((equal (node-type c)
            'RSUB)
        (if (lchild? p c)
         (setf (node-left p)
             (node-right c))
         (setf (node-right p)
             (node-right c))))
       (t (let ((m (largest
                (node-left c))))
          (deltree h (node-dat m))
          (setf (node-dat c)
              (node-dat m)))))))


(defun rdeltree[52] (h dat)
  (cond ((= (node-dat h) dat)
      (cond
        ((equal (node-type h) 'LEAF)
       nil)
        ((equal (node-type h) 'LSUB)
```

[51] let* and let are similar - but there IS a small difference. Find it out.

[52] rdeltree stands for root-deltree. A lousy name. I can't think of anything better at this moment - it's half past midnight.

```
      (node-left h))
      ((equal (node-type h) 'RSUB)
      (node-right h))
      (t (let ((m (largest
                (node-left h))))
         (deltree h (node-dat m))
         (setf (node-dat h)
             (node-dat m))
         h))))

    (t (deltree h dat)
      h)))
```

Please note that *rdeltree* always returns the root of the newly created tree. So you have to always invoke it as:

```
(setf h (rdeltree h 54))
```

Also, we need a way to build the tree. Here is a *mktree* function which reads in numbers from the keyboard and constructs a search tree using the *tappend* function:

```
(defun mktree (h)
  (let ((a nil))
    (setf a (read[53]))
    (if (null a)
      h
     (mktree
      (tree-append h (new a))))))
```

The no-argument function *read* reads a number(or symbol, or list or whatever) from the keyboard and returns the read value, which is assigned to the local variable 'a'. In this case, we will be entering a sequence of numbers and we will terminate the sequence by typing *nil*.

You should spend some time working out the code given here. Any C code you write for manipulating tree structures can be easily mapped to LISP once you understand the structure of the above programs.

11 Quick sort and some nifty functional programming tricks

I will describe a simple Quicksort implementation - and show you some tricks used by functional masters.

11.1 Trick1 - a function can return a function - lambda revisited

```
;;; functional trick 1

(defun foo ()
  (lambda (x y) (+ x y)))
```

You might remember that *lambda* is used to create anonymous functions. Our function *foo* simply calls *lambda* and asks it to build a function of two variables (x and y) which returns the sum of those values. Thus, the value returned by *foo* is a *function* which accepts two arguments and returns their sum.

Now what does this invocation do?

```
(funcall (foo) 3 4)
```

The call

```
(foo)
```

returns a function of two variables - this function is called by *funcall* which passes it two values 3 and 4. This is simple[54].

11.2 Trick2 - Closures

The function returned by a function can remember the environment in which it was born. Got it?

[53]See what happens when you type a control-D while read tries to read something from the keyboard. Can you make read work in a saner manner? You will have to refer a text book to solve this problem - just like many other problems given in the footnotes.

[54]Make sure you understand this!

```
;;; trick2 - lexical closure

(defun fun (x)
   (lambda (y) (+ x y)))
```

The function fun accepts a variable x and returns a function which accepts a variable y and which returns the sum of y and the value of x at the time the anonymous function was created. The anonymous function 'captures' the state of its environment at the time it was created, or it is said to 'close over' its environment - and hence the name 'closure' is given to such functions.

```
(funcall (fun 2) 3)
; prints 5
```

The invocation

```
(fun 2)
```

returns an anonymous function within which the value of x is 2. Now this anonymous function is called with one argument 3. Hence the result is 5.

11.3 Trick3 - mapcan[55]

There is a higher order Lisp function called *mapcan* which can be used as a filter. In its simplest form, mapcan accepts a *one-argument function which returns a list* and a list as arguments.

```
;;; use of mapcan

(defun neg (x)
  (if (< x 0)
   (list x)
   nil))

(print (mapcan #'neg '(2 3 -2 -4 1)))
;prints (-2 -4)
```

[55]There is a standard function called mapcar. Find out the difference between mapcan and mapcar

In the example shown above, *neg* is a function which accepts a number, and if it is less than zero, returns the number as a list. This function is passed as the first argument to mapcan. The second argument is a list of numbers. Mapcan calls the function *neg* with each element in the list as argument and simply appends the value returned by each invocation to form a list, which is then returned. Mapcan thus acts as a filter, removing certain elements from the list based on a criterion which is specified as a function.

11.4 Trick4 - operators are functions too

```
;;; operators are functions

(defun fun (f a b)
  (funcall f a b))

(fun #'< 1 2); T
(fun #'> 1 2); NIL
```

I don't have to say more. When you do

```
(fun #'< 1 2)
```

you are passing to function fun the value of the symbol '<' as a function, and two other values 1 and 2.

11.5 Quicksort

Having understood all these tricks, you are now ready to implement Quicksort. The idea is simple. An empty or one element list is already sorted. Otherwise - read the code to find out! It is much simpler than the C implementation.

```
;;; Quick sort

(defun pivot (a)
  (car (last a)))

(defun remove-pivot (a)
  (reverse
    (rest (reverse a))))
```

```
(defun sorted? (a)
  (or (= (length a) 1)
      (null a)))

(defun check (a f)
  (lambda (x)
    (if (funcall f a x)
      (list x)
      nil)))

(defun qsort (a)
  (if (sorted? a)
    a
    (append
      (qsort (mapcan
              (check (pivot a) #'>)
              (remove-pivot a)))
      (list (pivot a))
      (qsort (mapcan
              (check (pivot a) #'<=)
              (remove-pivot a))))))
```

There is a standard *sort* function in Lisp. Learn to use it.

12 Currying

Lambda's are 'brain exploding' stuff. We will see one more trick called 'currying'[56] involving Lambda and closure.

The idea is that it is possible to define a function accepting N arguments as repeated applications of functions accepting just one argument. Because Lisp allows on-the-fly creation of functions and because functions can return functions, this is quite simple to do.

```
(defun foo (x)
(lambda (y) (+ x y)))
```

What happens when you call (foo 2)? Think about it. Think a lot.

Look at this beauty:

```
(defun foo (x)
(lambda (y) (lambda (z) (+ x y z))))
```

What does it do? We shall discuss further in the class.

13 The Metacircular Evaluator, or Lisp in Lisp[57]

Once again, don't mistake Lisp to be C/C++ - you can do far more interesting stuff far more easily sometimes for the sake of doing far more interesting stuff far more easily.

One of the big features of Lisp is that there is not much difference between Lisp programs and Lisp data, both are lists. Because Lists have such a nice, regular, recursive structure, and because programs are simply lists, parsing Lisp in Lisp is a far more trivial job than the parsing of other languages like C[58].. I shall try to explain more in the class.

The idea is this - the process of evaluation of a Lisp program itself can be expressed as a Lisp program - we will write an evaluator for Lisp in Lisp - such an evaluator is called 'metacircular'.

Let me quote from SICP:

> *Metalinguistic Abstraction,* establishing new languages, plays an important role in all branches of engineering design[59]. It is particularly important to computer programming, because in programming, not only can we formulate new languages

[56]There was a logician called Haskell Curry whose ideas the functional guys love very much!

[57]Or, I should say, Lisp in Python rewritten in Lisp, because I rewrote Chris Meyer's Python Lisp evaluator in Lisp!

[58]As a beginner, you might take some time to digest this fact. Don't worry, you can leave out this section and still understand Lisp - you will not realize its true nature - that's all. But then, how many of us are interested in knowing the true nature of anything?

[59]The electrical engineer use many different languages for describing circuits. Two of these are the language of electrical networks and electrical systems. The network language describes the circuit in terms of primitive elements like resistor, capacitors, inductors etc. In contrast, the primitive units of the systems language are signal processing elements like filters and amplifiers. The systems language is erected on top of the network language, but the concern is more with the large scale organization of complex systems.

> but we can also implement these languages by constructing evaluators. An evaluator (interpreter) for a programming language is a procedure that when applied to expressions of the language, performs the actions required to evaluate that expression.
>
> It is no exaggeration to regard the following as the most fundamental idea in programming - *The evaluator, which determines the meaning of expressions in a programming language, is just another program.*

I shall explain my implementation of the metacircular evaluator in the class (if possible - it's a bit complex, I am not sure whether I can give a coherent presentation - the interested reader can get some materials from me which he/she can read).

14 Alonzo Church Magic[60]

Logic is supposed to be the origin of mathematics - only lunatics[61] would study logic - its so terribly complex. Church's Lambda Calculus is a logical scheme for understanding computation - we find it difficult to understand because Church builds up tremendously complicated things from almost absurdly simple notations. Church tells us that just knowing how to define a function (note - we use nothing but 'lambda' and arbitrary symbols - no symbol except 'lambda' has any special meaning - no numbers, no addition, no conditions, no loops, nothing - absolutely nothing else - all these things can be built using 'lambda' - simply amazing) lets us do anything which a computer can do. I present a glimpse into this wonderful world through some 'watered-down' examples (I don't Curry - I don't explain Y - I use 'define' and traditional 'if'). **I use Scheme because it's notation is simpler.** More explanations in the class.

```
(define true
  (lambda (x y) x))

(define false
  (lambda (x y) y))

(define iff
  (lambda (p x y) (p x y)))

(define pair
  (lambda (x y)
    (lambda (f)
      (f x y))))

(define first
  (lambda (p)
    (p true)))

(define second
  (lambda (p)
    (p false)))

(define zero
  (pair true true))

 (define succ
   (lambda (z)
     (pair false z)))

 (define pred
   (lambda (z)
     (second z)))

 (define is-zero
   (lambda (z)
     (first z)))

 (define lam-int
   (lambda (z)
     (if ((is-zero z) #t #f)
         0
         (+ 1 (lam-int (pred z))))))
```

[60] I wrote this section by reading a 'Perl Journal' column on Lambda Calculus by a famous Perl hacker. I substituted Scheme with Perl. I find this to be the best introductory article on the Lambda Calculus on the web even though it talks little of the calculus itself. I got stuck up at the section on fixed points and the Y-Operator, which is simply mind boggling. What I present here is pretty elementary stuff - it's meant to be entertaining for those who like to be intellectually entertained - that's all. You can safely skip it if you have no inclination for mental gymnastics.

[61] But then, lunatics change the world, for the better or the worse.

```
(define int-lam
  (lambda (i)
    (if (= i 0)
        zero
        (succ (int-lam (- i 1))))))

(define add
  (lambda (a b)
    (if ((is-zero b) #t #f)
        a
        (add (succ a) (pred b)))))
```

15 Stream Processing

15.1 The Infinite is Finite?

Why do we think in a particular way? Mostly because we have been thinking that way for a long time. Ordinary human beings unconsciously fall into the trap of believing that the way they think is the ONLY way simply because they have not done anything else for a long time. Great scientists too sometimes fall into this trap, but we note that all throughout history, the greatest revolutions have been brought about by people who dared to think in a different way.

The programmer who codes in a language like C for a long time has difficulty in 'Thinking in Lisp' because he is sometimes forced to think in ways which are rather bizarre, considering his 'imperative' upbringing. You might feel such difficulties if you try to digest SICP all by yourself. I demonstrate a closure based implementation of inifinite lists as finite 'lazily-expanding' pairs. If you have by now acquired the knack of 'thinking in Lisp', the code should be clear. Otherwise, you can leave out this section, just like the other 'advanced' sections in this document.

15.2 What's a list?

A List is something which 'cons' generates and which can be taken apart by 'car' and 'cdr'. We can create 'interesting' new kinds of lists by defining the cons-car-cdr operations in innovative ways. Let's look at a step-by-step presentation (We change back to CLISP for this example)

15.2.1 Delaying evaluatoin

```
(defmacro delay (expr)
  `(lambda () ,expr))
```

What happens when we call (delay (+ 1 2))? The function returns a function whose body is (+ 1 2). Note that we are 'delaying' the evaluation of (+ 1 2) by packing it within the body of a function which can be invoked at a later time.

15.2.2 The 'force' operation

```
(defun force (delayed-object)
  (funcall delayed-object))
```

The 'force' operation simply performs the evaluation of the delayed object. An example sequence would be:

```
(setf p (delay (+ 1 2)))
(force p) ; return 3
```

15.2.3 stream-cons

```
(defmacro stream-cons (a b)
  `(cons ,a (delay ,b)))
```

The function simply creates a cons pair of its first argument with a 'delayed' version of its second argument.

15.2.4 stream-car

```
(defun stream-car (a)
  (car a))
```

The function simply extracts the first value of the pair 'a'.

15.2.5 stream-cdr

```
(defun stream-cdr (a)
  (force (cdr a)))
```

The function extracts the cdr of 'a' and forces its evaluation.

15.3 An infinite list represented finitely

We can now use the above operations to define an infinite list finitely. Lets look at this function:

```
(defun foo (a)
  (stream-cons a (foo (+ a 1))))
```

Can you understand it's magic? If so, you are own your way to being a Lisp Guru!!!

16 Set Implementation

There are builtin Lisp functions *union, intersection and member.* Learn to use these functions. We will write simple imitations. We start with member:

```
;;; checking for membership

(defun my-member (a l)
  (if (null l)
    nil
    (if (equal a (car l))
      t
      (my-member a (cdr l)))))
```

There is a difference between this function and the standard function *member.* Find out what it is and try to implement it.

Here is a simple implementation of set union:

```
;;; set union

(defun my-union (a b)
  (cond
    ((null b) a)
    ((not (member (car b) a))
     (my-union
      (append a (list (car b)))
      (cdr b)))
    (t (my-union
         a (cdr b)))))
```

A set is not supposed to contain duplicates. My-union adds to a only those elements in b which are not in a. This ensures that there will not be any duplicates if 'a' and 'b' do not contain duplicates. But if the set 'a' contains duplicates, then the union too will contain duplicates. The standard function *union* works this way, it does not attempt to eliminate duplicates. Maybe, you can modify the function to ensure that duplicates never occur. You can think of using the builtin function *remove* or you can develop your own version of *remove,* then you can develop a function called *remove-duplicates.* You **MUST** do this as an exercise.

Here is *intersection,* again having the same problem with duplicates:

```
;;; set intersection
(defun my-intersection (a b)
  (cond
    ((null b) nil)
    ((member (car b) a)
     (cons (car b)
           (my-intersection
              a
              (cdr b))))
    (t (my-intersection
         a
         (cdr b)))))
```

17 Looping[62]

17.1 dotimes

```
(dotimes (i 3) (print i))
```

The body of the code will execute three times - with value of i set to 0, 1 and 2.

17.2 dolist

```
(dolist (x '(a b c d))
  (print x))
```

The body of the loop executes 4 times - with value of x set to a b c and then d.

17.3 do

```
The general form is:
(do (
  (iteration-var-1 init-form-1 stepping-form-1)
  (iteration-var-2 init-form-2 stepping-form-2)
  .............
  .............
  (iteration-var-n init-form-n stepping-form-n))

 (termination-test result-form)
 body-form-1
 body-form-2
 ..........
 body-form-n)
```

A specific example is:

```
(do (
         (i 0 (1+ i))
         (j 0 (1+ j))
        )
         ((= i 4) nil)
         (format t "i=~A~%" i)
         (format t "j=~A~%" j))
```

[62] Remember, loops are secondary - learn to write Lisp in Lisp style (functions, recursion) and only use loops when absolutely necessary.

18 Macros

The Lisp macro facility is very powerful - we examine only one simple notation in the class.

Let's try to write a function which simulates 'if'.

```
(defun my-if (a b c)
   (if a b c))
```

We check whether it is working properly:

```
(my-if (> 1 0) 2 3) ; OK
(my-if (< 1 0) 2 3) ; OK
```

When will my-if fail? Think.

Let's rewrite as a macro:

(defmacro my-if (a b c) '(if ,a ,b ,c))

Where is the difference?

We look at how the following definitions:

(defun new (dat) (list nil dat nil))

(defmacro node-left (p) '(car ,p))
(defmacro node-right (p) '(caddr ,p))
(defmacro node-dat (p) '(cadr ,p))

Help us create a binary search tree. We note how the trick of hiding data representations with constructor and accessor functions let's us write code which is independent of the actual representation of data.

19 Associative Array

The Lisp implementation of an associative array is simple:

```
;;; associative array

(setf G '((0 (1 2) ) (a (b c d))
          (pce 330694)))
(print (assoc 0 G))
;prints (0 (1 2))
(print (assoc 'pce G))
;prints (pce 330694)
```

The associative array is basically a list of tuples of the form

```
( (key1 val) (key2 val2) ...
  (keyn valn) )
```

Assoc is a standard function, which, given a key, extracts a tuple which matches the given key.

Associative arrays can be used to represent graphs. As an exercise, try to implement a graph and write functions to do DFS and BFS. Here is a program which will display a path from one node to another - mod

20 Arrays

We now look at some other syntax elements - you can write code even without knowing all these stuff - but try to at least skim through the material presented here.

Lisp has a data structure which behaves like ordinary arrays.

```
;;; arrays

(setf a (make-array 10))
; a is 10 element array
; in the sense that you can
; index it.
(aref a 0); prints NIL
;clisp initializes all cells
;to NIL
(aref a 10); run time error
(setf (aref a 0) 20)
(print (aref a 0));prints 20

(setf a (make-array '(5 5)))
(aref a 0 0)
(setf (aref a 0 1) 20)
(print (aref a 0 1)) ; prints 20
```

Here is a program which accepts an array (which contains only numerical values) and returns the sum of all the elements in the array:

```
;;; sum elements of an array

(defun sum-1 (a i len)
  (if (= i len)
    0
    (+ (aref a i)
       (sum-1 a (+ i 1) len))))
(defun sum (a)
  (sum-1 a 0 (length a)))
```

Can you write a function to add and multiply two dimensional arrays?

20.1 Characters and Strings

[Note: It may be difficult for you to remeber a lot of syntax - all string manipulation problems can be rephrased as list manipulations - it would be better to do them that way. A string "hello" may be thought of as a list (h e l l o) - you can then write functions to check whether two strings are equal, whether one is a substring of the other, remove all substrings etc - that way, you strengthen your list manipulation skills].

Common Lisp treats characters as a separate data type. You represent the character 'A' by typing

```
#\A
```

you represent the character 'u' by typing

```
#\u
```

There is a function which checks whether its argument is a character.

```
(characterp #\A); returns T
```

Some other functions:

```
(char-code #\A) ; we get 65
(code-char 65) ; we get #\A
```

Strings are vectors of characters. Here are some functions:

```
(aref "hello" 0);we get #\h
(length "hell") ; we get 5
(parse-integer "123") ; we get 123
;converts string to integer.
(parse-integer "FFFF" :radix 16)
;we get 65535.
```

Can you write your own version of parse-integer?

```
(reverse "abc") ;we get "cba"
(concatenate 'string "abc" "def")
;we get "abcdef"
(concatenate 'list '(a b c) '(d e f)
;we get '(a b c d e f)
```

Try out

```
(concatenate 'list "abc" "def")
```

What happened?

```
(count #\a "abab")
(position #\a "haha")
(sort "pramode" #'char>)
(sort "pramode" #'char<)
```

What is this *char>* and *char<*. They are two functions which are used for comparing characters. The notation *#'char>* means treat 'char>' as a function. Why are we passing a function as argument to sort?

```
(search "one" "anyonehere")
;gives you index of first
;occurrence of "one"
```

21 File I/O

First, you have to open a file:

```
(setf fd (open "dat"))
```

Then you can read lines from it:

```
(read-line fd nil nil)
```

Don't bother with the two nil's in the end. The *read-line* function reads a line from the file pointed to by fd and returns it as a string. Upon end of file, the function returns 'nil'.

You can also use the function

```
(read-char fd nil nil)
```

The arguments fd, nil and nil are option. Simply invoking

```
(read-line)
```

will make the interpreter read a line from the standard input.

With this much of info, we can very easily write a function which will read lines from a file and display it on the screen.

```
;;; file I/O - 1

(defun print-file (fd)
  (let
    ((s (read-line fd nil nil)))
    (if (null s)
      nil
     (progn
       (print s)
       (print-file fd)))))

(print-file (open "dat"))
```
[63]

[63]Try this on a large file - /usr/share/dict/words. Does CLISP really do tail call optimization?