LISP Basics

Much of this material lifted from Nick Parlante and Todd Feldman.

LISP and the functional paradigm

LISP is a stunning example of a language designed in the functional paradigm. Whereas imperative languages make heavy use of the assignment statement and route control through loops and branches to change the state of memory, the building blocks of a functional language are expressions that evaluate to values through function application. In a pure functional language, there is no assignment statement at all, functions get read-only access to their inputs and construct or synthesize the resulting answer from the input without modifying or updating any previous state. ML, for those of you who saw it in CS109, is another example of a functional language.

Some characteristics of LISP:

- LISP operates at a higher level of abstraction than imperative languages like C, C++, Fortran, and Pascal. In this sense, it takes care of many implementation details that C programmers are forced to deal with themselves (memory allocation and deallocation just to mention a big one).

- LISP supports an amazingly high level of generalization (e.g. code that is run-time dependent on the types of its parameters) and flexibility (e.g. data and programs can be modified or added during execution).

- LISP is highly recursive in terms of its data structures (lists, trees, etc.) and algorithms.

- LISP is driven almost entirely by functions— a function passes its result to another function, which passes the results to another function, etc.

- LISP maintains a striking equivalence of programs and data. We've already discussed that traditional compiled object code shares the same memory space as the data on which it operates. But in LISP it goes even further. The fundamental data structure is the list and LISP programs are themselves formulated as lists! Unlike in C, functions are truly first-class objects in LISP, you can create, modify, and evaluate functions at run-time in the same way you might create, modify, and evaluate a simple variable.

- LISP does not require variables to be declared and typed. As each piece of data is created, the LISP system associates a type with it. These types can be accessed at run-time, and the type information is always carried along with the data. This allows LISP to support generic functions and object-oriented programming quite easily since

functions can decide at run-time how to perform a specific computation if that computation depends upon the type of data involved.

- For all of these reasons, LISP is well-suited to problems whose solutions are unknown or problems which are not entirely understood. Code can be written to solve small pieces of the problem without worrying about many aspects of the rest of the problem (e.g. no need to deal with variable declarations and types, special cases, initialization, etc.). These are some of the reasons LISP is often thought of as an artificial intelligence (AI) programming language: AI problems are some of the most complex and least understood in all of computer science. Some common LISP applications include vision, programming language design, natural language, planning, computer-aided design (CAD), theorem proving, symbolic algebra, and text editors.

A Little History

LISP was originated by John McCarthy (now a professor at Stanford) at MIT in 1960 as language designed to meet the needs of the artificial intelligence community. LISP is the second oldest language still in active use (after our friend Fortran) and many of its good ideas can be seen reflected in other languages since developed. During the 70's and 80's a lot of splintering developed between the features and behaviors of various LISP implementations, much to this dismay of programmers trying to write portable code. In 1984, after two years of e-mail based discussions, LISP's major users and supporters finally agreed upon a new standard definition of LISP that was intended to serve as the common dialect spoken by all LISP programmers. Common LISP, as it was called, might best be described as the union of all the different LISP dialects rather than their intersection. As a result, Common LISP is an enormous language, containing over 700 built-in functions. With this strong standard now in place, Common LISP has become the most widely used version of LISP, making LISP programs some of the most portable code around. We will learn Common LISP in CS107, and from this point on I'll probably use the terms LISP and Common LISP interchangeably.

Building LISP Programs

In stark contrast to rigid languages like C and Pascal, which require the types of all variables to be completely specified in advance and can't execute anything until all of a program's components have been cleansed of compiler errors, LISP is extremely flexible. There's no need to commit in advance to what type of data will be bound to a particular variable or what form the rest (i.e. as yet unwritten or unfinished) of the code will take. Any reasonable LISP development system will allow you to add, modify, and debug your code at almost any time— even while it is running!

Because LISP operates at a relatively high level of abstraction, it is trivial to create complex data structures as lists and trees on the fly. Think of all of the C code you'd have to write just to set up and search a binary tree— defining a data type (and you'd have to decide what type of value will be stored at the nodes, or else go through the

extra work required by a polymorphic implementation), write (and debug) some code to allocate the nodes and build a sample binary tree, and then write (and debug) the code you're really interested in, the code that searches the tree. In LISP, with built-in support for creating trees, you don't have to worry about defining any new data types, you don't have to worry about what type of value will eventually be stored in the nodes of your tree, and you can create a sample tree on the fly which you will pass interactively to your tree searching code right away.

Many decisions can be made *dynamically* as opposed to *statically*, such as the type of data passed to a function or the particular comparison function to be applied to two or more pieces of data (and even the number of pieces of data being compared). Other details can also be left unspecified, such as the lengths of lists or arrays. LISP simply responds to the situations in which it finds itself at run-time.

As a result, the "LISP style of programming" has become known as *exploratory* programming. You investigate different aspects of a problem one at a time and eventually put all of your work together into a single unified collection of code to form a solution.

LISP is Interpreted

LISP is implemented by an interpreter— most broadly this means that there is not a strict compile-time / run-time distinction. Instead the source code is present in some form at run-time. The interpreter is itself a program which "interprets" the source code at run time to run the program. Note that LISP code *can* be compiled. Although code operates the same whether it's compiled or interpreted, compiled code runs faster. Since compiling code takes more time and destroys the interactive, fluid environment of using the interpreter, LISP code is usually only compiled when it has reached its final form.

You work with LISP by interacting with the LISP interpreter. The interpreter displays a prompt like ? or > to indicate that it is ready and waiting. You interact with the interpreter by giving it an expression to evaluate. The LISP interpreter runs a read-eval-print loop where it reads a symbolic expression, evaluates the expression, and prints the value computed.

Data Types

There are two basic categories of data in LISP: *atoms* and *lists*. Atoms are all the data types that are "atomic" or "basic". Atoms are the types which are not aggregates of other, smaller, constituent data. The list is the unified abstraction for collections of objects. LISP also supports arrays and structures for aggregating objects, but we are not going to cover them, since they are not used as much as lists.

Atoms

Number Examples of numbers are: `42` and `3.14`. Numbers evaluate to themselves. At run-time, there are four major sub-types of number: integers, floating

point, rationals, and complex. The number abstraction is implemented very seamlessly in LISP, so you do not need to worry about which subtype of number a particular value is. You can just think of all numeric values as just numbers, without worrying about whether they are integers or floats or whatever. The numeric functions `+`, `-`, `*`, `/`, `mod`, `floor`, `ceiling`, `round`, `sqrt`, and `expt` work correctly for all the different combinations of numeric subtype. This is an example of a language allowing you to express your computation in terms of the way you think, numbers, instead of how the computation needs to be implemented, short vs. long vs. float (think "expressiveness").

String Just like other languages, strings are sequences of characters used as data. Strings are bounded with double quotes ("). Examples of strings: `"Hi there"`, and `""`. Strings evaluate to themselves.

Symbol Symbols are sequences of characters that are used to name things such as constants, variables and functions. Examples of symbols are: `i`, `tempList`, and `+`. LISP is not case sensitive. The value of a symbol is its "binding" which is set by the operation of the program, possibly at run time. There are many built-in symbols: including all of the built-in functions and pre-bound variables such `pi`.

Lists

A list is a sequence of any LISP values. In ASCII form, a list is represented by a left parenthesis '`(`' followed by the ASCII forms of all the elements separated by spaces, followed by a right parenthesis '`)`'. The values inside a list are called "elements". There may be any number of elements in a list, and they may be of different types. In particular, lists may contain other lists. The following are all lists containing three elements.

```
(1 2 3)                      ; three numbers (all integers)
(1 2.5 22/7)                 ; three numbers (int, float, rational)
("hi" (1 2 3 4) 42)          ; a string, a list, and a number
("hi" ((((1 2 3 (4))))) 42)  ; a string, a list, and a number
```

Function Evaluation

The list is the unified aggregate data structure in LISP— it is used to store both code and data. When a list is passed to the evaluator, it interprets the list as a function call where the function is the first element and the rest of the elements are its arguments. The evaluator works by (1) recursively evaluating the arguments, and (2) evaluating the function using the values from step (1).

```
? (+ 3 7)
10
? (+ 3 (* 4 5))
23
? (+ 3 (* (+ 1 2 3) 4))
27
```

Note since LISP expressions are fully parenthesized, there is no need for precedence rules to be defined to disambiguate among expressions.

Quote

Since the evaluator interprets lists as function calls and tries to evaluate them, we need a way to express a list *without* evaluating it. When a quoted expression is given to the evaluator, the expression is passed through intact. In particular lists are passed through as lists instead of being interpreted as function calls.

```
? (1 2 3)
#!Err '1' is not a function
? '(1 2 3)
(1 2 3)
? a
#!Err symbol 'a' has not been given a value ("bound")
? 'a
a
? '(a b c)
(a b c)
```

The quote is your way of telling the interpreter to take what you said verbatim.

cons

cons is the basic list construction function. cons takes an element and a list and constructs a new list, one longer than the old one, which has the element pushed on the front of the list. Elements in lists may be of any type: numbers, strings, symbols, other lists, etc. and need not all be of the same type (a list can contain any combination of numbers, symbols, other lists, etc.) cons is the low-level memory allocator. Much of the memory allocation in a LISP program depends at some level on calling cons a lot.

```
? (cons 1 '(2 3))
(1 2 3)
? (cons "hi" '((2) 3))
("hi" (2) 3)
? (cons (+ 3 4) '( 2 3))
(7 2 3)
? (cons '(+ 3 4) '(2 3))
((+ 3 4) 2 3)
```

Empty list

The list with no elements, the "empty list", is a valid list. It can be represented with the normal parenthesis notation `'()`. Because the empty list is so common, the symbol `NIL` is a synonym.

car

While cons builds lists out of elements, `car` extracts elements from lists. `car` returns the first element of a list. The argument to `car` must be a list. The term car is historical—`first` is a synonym. It is a conceptual error if code tries to evaluate the `car` of the empty list `(car '())` since there is no first element to return. For backward

compatibility, Common Lisp doesn't report an error in that case and just returns `NIL`, but you should avoid it in your own code. The functions `second`, `third`, ...`tenth` are defined as convenient ways to refer to later elements in a list.

```
? (car '(1 2 3))
1
? (car '((+ 3 4) (2) ((3)))
(+ 3 4)
```

cdr

Takes a non-empty list and returns the "tail" or "rest" of the list—the balance of the list without the first element. rest is a synonym. As with car, taking the cdr of an empty list is conceptually vulgar, although Common Lisp will return the value NIL in that case.

```
? (cdr '(1 2 3))
(2 3)
? (cdr '((+ 3 4) (2) ((3)))
((2) ((3)))
? (cdr 1)
#! ERR '1' is not a list
```

Alternate List Constructors

Sometimes the cons configuration of combining an element and a list is not convenient. In that case the functions list and append may be what you need. list merges several elements into one list while append merges several lists together into a single list. Like cons, both append and list allocate memory to construct their results.

list

Takes any number of values of any type and binds them all together in a new list. The number of arguments to list will be the number of elements in the new list. Use List when you have a bunch of *elements* and you want to bind them together into a single list. If you have an element and list you wish to add to it, use Cons. Use List when all you have are elements.

```
? (list 1 2 (+ 3 4) '(* 5 6))
(1 2 7 (* 5 6))
? (list (list 1 2) "hi")
((1 2) "hi")
```

append

Takes any number of *lists* and merges their elements together into a new list. The length of the resulting list will be the sum of the lengths of the given lists. All of the arguments to append must be lists. You use append to splice together a sequence of lists into one combined list.

```
? (append '(1 2) '(3 4 (5) 6) '("hi" "there"))
(1 2 3 4 (5) 6 "hi" "there")
? (append (cons 4.5 '()) '() '(a b) '((("hi"))))
(4.5 A B (("hi")))
```

Variable arguments

As a side note, you'll notice that many of the built-in functions are flexible about handling any number of arguments of various types. This is not unlike the variable argument facility used by C functions such as `printf`, but in LISP, variable arguments functions aren't compromised by degradation in type-matching or potential for misinterpretation of parameters the way they are in C.

Boolean type

Common LISP does not have a distinct boolean type. This decision was purely for backward-compatibility with all the older dialects of LISP. Any expression can play the part of a boolean— `NIL` is false, any other non-nil value is true, similar to the way C defines it. The symbol `T` is used as a convenient, readable non-nil value.

Boolean Functions

LISP has the usual boolean operators as well as many built-in predicates which return a truth value to test some condition:

`and`	Takes any number of expressions and returns true if they are all true. It "short circuits" in that it evaluates from left to right and stops as soon as it finds a false expression.
`or`	Takes any number of expressions and returns true if any one is true. Short-circuits like and (stops as soon as it finds a true expression).
`not`	Boolean inverse.
`null`	True given an empty list. This is just a readability convenience since testing for the empty list comes up a lot.
`numberp`	True given a number.
`stringp`	True given at string.
`listp`	True given a list, possibly empty.
`atom`	True given an atomic value— anything but a non-empty list. (NIL is an atom, as well as a listp.)
`evenp, oddp, zerop`	These work only on numbers.

Some examples:

```
? (and (stringp "hello") (zerop 1.5) (not (listp 5)))
NIL
? (or (evenp 5) (oddp 5))
T
? (null '(a b "hi"))
NIL
? (or (atom '()) (numberp "hi"))
T
? (and T 0)      ;; note any number, even 0, is not NIL!
T
```

Comparators

`equal` Takes two arguments and returns true if they represent the same value. The arguments can be strings, number, atoms, lists, etc. This is a "deep" or O(n) version of equivalence— it traverses the two values to verify that they really are the same throughout.

`=, >, <=` (and other forms) These only work for numbers.

`string=, string>, string<=` (and other forms) These only work for strings.

`eq` This is a "shallow" variant of equal we will not have occasion to use. It tests whether two expressions are stored in the same memory. That is, the two arguments are essentially the same data-structure passed in twice. In some cases, whether two expressions are `eq` depends on non-portable properties of the particular LISP implementation.

Examples:

```
? (< 5 5.1)
T
? (= 4.0 (/ 8 2))
T
? (string= "hello" "HELLO")     ;; string= is case-sensitive
NIL
? (equal '(1 (a) "why") '(1 (a) "why"))
T
? (equal "who" (car '("who" "what" "when")))
T
? (equal 4 "4")
NIL
```

`if`

The `if` form in lisp is most similar to the ternary `?:` operator in C. The `if` is in the form of an expression. Syntactically it looks like.

(if *test-expr then-expr else-expr*)

It first evaluates the test expression to determine which value to return— the `then` value or the `else` value. The `else` value is not optional— the `if` expression needs a value to

return at run-time in the case where the test is false. By the way, the types of the `then` and `else` values do not have to match.

Examples:

```
? (if (< 3 5) "smaller" "larger")
"smaller"
? (if (null '()) '(a b) 6)
(A B)
```

Cond

`cond` is a generalized form of `if`. Its syntax is a little messy, so `if` is preferable where there are only two cases to distinguish.

```
(cond (test-1     consequent-1)
      (test-2     consequent-2)
      ...
      (test-n     consequent-n)
)
```

`cond` goes through the tests in order. As soon as one of the tests is true, it evaluates and returns the corresponding consequent expression. If no test expression returns true, then `cond` returns `NIL`. `if` and `cond` are not functions strictly speaking since they do not evaluate all of their arguments— they are known as "special forms". An `if` expression can be translated to the following `cond` expression.:

```
(cond (test-expr  then-expr)
      (T          else-expr)
)
```

Defun

`Defun` allows you to define new functions in LISP. Adding new functions to LISP is dynamic—when the `defun` is evaluated (potentially a run-time activity), the new function is added to the system.

```
(defun function-name(arguments)
   body-expr
)
```

The `defun` parses the text for the function, and binds the code to the given name. Once the function is defined, it can be called like any other function. Calling the function is equivalent to evaluating its body-expr. In the simplest case, there is one global name-space where all the function names and other globally defined identifiers are bound. It's possible to redefine system functions such as `append` since they are really just functions indistinguishable from your own, but it's not recommended. Functions can be defined in any order. The function name lookup happens at run-time, so it is only important that a function be defined before it is called at run-time.

```
? (defun double(num)
   (* 2 num))
DOUBLE            ;; the intepreter echos the function name just bound

? (double 3)
6

? (defun subtract-second(list)
   (cons (car list) (cdr (cdr list))))
SUBTRACT-SECOND

? (subtract-second '(a b c d e))
(A C D E)
```

cdring Down

Loops do not fit well within the functional paradigm. Loops depend on re-executing a sequence of statements using a loop variable whose value changes for each iteration. So the whole idea of a loop is very tied up with the idea of variables and sequences of statements. In the functional paradigm, any loop can be expressed using recursion or mapping (covered later).

One extremely common form of recursion in LISP is recursively traversing down a list. In this form, each call incorporates the value of the `car` into the answer combined with the `cdr` passed to a recursive call. The `cdr` always yields a list of shorter length, so the recursion must terminate eventually.

Here are a few simple examples which demonstrates the form of cdring down recursion.

```
;; Given a list of numbers, counts number of elements
;; (btw, the built-in "length" already does this...)
(defun my-length(list)
   (if (null list) 0                ;; the perennial base case
      (+ 1 (my-length (cdr list)))  ;; count car & recur on cdr
   )
)

;; Given a list of numbers, compute a list where all the numbers
;; have been doubled.
(defun double-list(nums)
   (if (null nums) '()              ;; (1) base case
      (cons (double (car nums))     ;; (2) computation on the car
         (double-list (cdr nums))   ;; (3) recursion on the cdr
      )
   )
)
```

Be sure to account for the base case on the empty list. Because both `car` and `cdr` of `NIL` return `NIL` (although it is vulgar to depend on that), forgetting the base case would lead to infinite recursion.

let

let is a convenience for binding temporary constants.

```
(let  ((var-1     expr-1)
       (var-2     expr-2)
      ...
       (var-n     expr-n)
      )
   body-expr
)
```

The Let establishes a binding for each of the variables in its binding list. Each variable is bound to the value of the corresponding expression. Once all the bindings are established, the body-expr is evaluated within the scope of all new bindings. The bindings are only visible inside the body-expr. In particular the variables may not refer to each other. The order that the bindings are established is not defined.

Let can help readability since it allows you to take some value midway through a computation and put it in a variable with a nice readable name. The variables in a Let are actually more like temporary constants since we will never change the value once it is bound.

Here's a function that recurs down a list of integers. It builds a new list incrementing any odd values so they are even.

```
(defun make-even(nums)
   (if (null nums) '()                    ; base case
      (cons (if (oddp (car nums)) (+ 1 (car nums)) (car nums)) ; do car
            (make-even (cdr nums))        ; cons with cdr
      )
   )
)

? (make-even '(1 2 3 4 5 6 7))
(2 2 4 4 6 6 8)
```

Here's the function rewritten with a Let which temporarily stores the even value in a variable.

```
(defun make-even(nums)
   (if (null nums) '()
     (let ((evenNum (if (oddp (car nums)) (+ 1 (car nums)) (car nums))))
        (cons evenNum (make-even (cdr nums)))
      )
   )
)
```

let*

There's a variant of let called let* where the binding clauses are guaranteed to be evaluated in order, and so the binding expressions may refer to earlier bound variables. Although at times, you may sparingly need to use this, beware of allowing a sequence of

`let*` bindings lull you back into the imperative / assignment statement programming style you are most familiar with.

Some simple examples

```
;; Compute the power of non-zero base with a positive integer exponent
(defun power1(base expt)
   (if (= expt 0)
       1
       (* base (power1 base (1- expt)))))


;; This variant includes the great optimization where for even exponents
;; it computes the power with half the exponent and the squares it.
;; Uses Let to compute the half-power once and use it twice.
(defun power2(base expt)
   (if (= expt 0) 1
      (if (evenp expt)
         (let ((half-power (power2 base (/ expt 2))))
            (* half-power half-power))
            (* base (power2 base (1- expt))))))


;; Insert a number into a sorted increasing list of numbers
(defun insert1(num nums)
   (if (null nums) (list num)
      (if (<= num (car nums))
         (cons num nums)
         (cons (car nums) (insert1 num (cdr nums))))))


;; This insert is more compact by unify the first two of the three
;; cases in insert1 (but it's probably even harder to read)
(defun insert2(num nums)
   (if (or (null nums) (<= num (car nums)))
      (cons num nums)
      (cons (car nums) (insert2 num (cdr nums)))))


;; Uses insert to recursively sort a list of numbers.
(defun insert-sort(nums)
   (if (null nums) '()
      (insert1 (car nums) (insert-sort (cdr nums)))))


;; Return a list with all the occurrences of elem deleted.
(defun delete1(elem list)
   (if (null list) '()
      (if (equal elem (car list)) (delete1 elem (cdr list))
         (cons (car list) (delete1 elem (cdr list))))))


; Returns the reverse of a list (not very efficient).
(defun reverse1(list)
   (if (null list) '()
      (append (reverse1 (cdr list)) (list (car list)))))
```