COMP 348 NOTES

Procedural Programming with C

Functions

Synonyms for function exist in various languages (such as method, procedure, or subroutine). It is also important to note that some languages make a distinction between functions that return a result and those that do not, the latter ones being referred to as procedures. We will use the C programming language to discuss procedural programming. The general form of a function definition in C is:

```
return-type function-name ( parameter-list) { body}
```

where **return-type** is the type of the value that the function returns,
function-name is the name of the function,
and parameter-list is the list of parameters that the function takes, defined as (type! parameter!, type2 parameter2, ...)

If no type is in front of a variable in the parameter list, then int is assumed.

Variable and function modifiers

Two modifiers are used to explicitly indicate the visibility of a variable or function: The extern modifier indicates that a variable or function is defined outside the current file, whereas the static modifier indicates that the variable or function is visible only from within the file it is defined in. The default (i.e. no modifier) indicates that the variable or function is defined in the current file and it is visible in other files. A summary is given below:

MODIFIER	DESCRIPTION
extern	Variable/function is defined outside of current file.
<blank>	Variable/function is defined in current file and visible outside.
static	Variable/function is visible only in current file.

The C standard library

We can access this API (called the C standard library) by adding the #include directive at the top of our program file. Perhaps the most common is the group of functions that support input-output and are accessed by <stdio. h>. This and other common header files are listed in the table below:

#include	DESCRIPTION
<math.h>	Defines common mathematical functions.
<stdio.h>	Defines core input and output functions.
<stdlib.h>	Defines numerIC converSIOn functions, pseudo-random number generation functions, memory allocation, process control functions.
<string.h>	Defines string manipulation functions.

Data Types

Primitive data types

Int, float, double, char
You know this.

Optional specifiers: Short, long, signed and unsigned.

Type conversion

Type conversion is the transformation of one type into another. In C, implicit type conversion (or coercion) is the automatic type conversion done by the compiler.

Defining constants

A constant defines a data type whose value cannot be modified. We can define a constant either with the **const** keyword as in:

```
float const pi 3.14
```

or with **#define** as in"

```
#define TRUE 1
#define FALSE 0
```

Composite data types

A composite type is one that is composed by primitive types or other composite types. Normally a composite type is called a data structure: a way to organize and store data so that it can be accessed and manipulated efficiently. Common composite types include arrays and records.

Arrays

Example:
float numbers[5] = {1 , 2.5 , 9};

It's like java.

Pointers

A pointer is a type that references(points to) another value by storing that other value's address. Declare a pointer like this:

```
int c = 7;
int *pointer = c;
```

2 operators used with pointers

* ("dereference operator")	Given a pointer, obtain the value of the object referenced (pointed at).
& ("address of" operator)	Given an object, use & to point to it. The & operator returns the address of the object pointed to

```
#include <stdio.h>
int main() {
   int a = 42;
   int *p;    //creates a pointer "p" referencing an integer
   p = &a;   //the address of "a" is stored in the pointer "p"
   printf("p: %d\n", *p);
   return 0;
}
```

Read the handouts for more good examples and run through.

Aliasing

Aliasing is a situation where a single memory location can be accessed through different variables. Modifying the data through one name implicitly modifies the values associated to all aliased names.
Look at example 9 in handout.

Constant pointers and pointers to constants

If you declare a const variable, you need to initialize it in the same line.
const int aValue; //will give an error

If you declare a pointer that points to a const, you need to write it like this:
const int * pointer = aValue;
this can be left uninitialized because the pointer is not a const, it only points to a const int.

You declare a const pointer that points to an int like this:
int * const pointer2 = aValue; //must be initialized

You declare a const pointer that points to a const int like this:
const int * const pointer3 = aConstValue; //must be initialized

Pointer and arrays

Look at example 11 in handouts.

Function pointers

A pointer can also reference a function since functions have addresses.

Consider the following (rather cryptic) declaration:

long (*ptr)(int);

This declares a function pointer; It points to a function that takes an integer argument and returning a long integer. We could now initialize the pointer by making it point to an actual function as follows:

ptr = &factorial;

This makes ptr point to function factorial (..). The function can be invoked by dereferencing the pointer while passing arguments as any regular function call, only in this case we refer to this as an indirect call

See example 13.

Records

A record, or structure, is a collection of elements, fields (or members), which can possibly of different types. The syntax of declaring a structure in C is :

```
struct <name> {
   //field declarations
};

//now we can create a <name> variable with
struct <variable> p;
```

We can eliminate the need to write struct every time with typedef:

```
typedef struct {
   //field declarations example
   float x;
   float y;
} <name>;

//now we can create a <name> variable with
<name> p;
//or
<name> p = {0, 0};
//or
<name> p = {.x = 1, .y = 3};
//or
p.x = 2;
p.y = 7;
```

See example 14.

Records and pointers

See example 15.

Records and arrays

In example 15, we make use of an array of records. More specifically, line[ ] is an array of type coordinate, itself defined as a record. The elements of the array are initialized at the time of declaration. We use the dot (.) operator to access fields of individual records.

Unions

A union is a variant of a record (structure). Unlike a record where there exists a separate memory location for each of its fields, the union associates all of its fields to a single memory location. In other words, union fields share the same space. This implies that only one field of a union can be accessed at a time, and modifying the value of one field results in the modification of the values of the rest of the fields.

Enumerated data types

Can be made like this:

Enum enum-name { tag-1, tag-2, ... }

where the tags are normally in uppercase. It is important to note that even though tags look like strings, they are not. Tags constitute keywords that we define for our program.

Memory management

Function	Description
malloc	Allocates the specified number of bytes
realloc	Increases or decreases the size of the specified block of memory.
calloc	Allocates the specified number of bytes and initializes them to zero.
free	Releases the specified block of memory back to the system.

Data structures and abstract data types

ADTs vs. data structures

ADT:

An abstract data type (ADT) is a definition for a data type solely in terms of the set of values and a set of operations on that data type. The behavior of each operation is determined by its inputs and outputs. This implies that an ADT is implementation-independent.

Data structure
A data structure is a specific implementation of an ADT. The implementation details are hidden from the clients of the ADT. This is referred to as information hiding.

The choice of a data structure for the implementation of a particular ADT involves benefits and costs. Because of these trade-offs, rarely (if at all) one data structure is better than another in all situations.

Linked list.. We know this. I skipped.

File I/O

The general form to access a file is

```
file-pointer  = file-I/O-function (file-name, mode);
```

Assume that we need to specify that we want to **open a file, out. txt, in order to write**. We can do this as follows:

```
FILE *fp;
fp = fopen("out.txt", "w");
```

where fp is a pointer that will keep track of this file, fopen( ) is a function to open a file (from the stdio library), and w is the writing mode.

Function fopen( ) returns a pointer. It would return NULL if for some reason the system has been unable in creating the file. No matter how unlikely this may be, it is a good practice to handle abnormal conditions:

```
if(fp == NULL) {
   printf("Could not open file\n");
   return 1;
}
```

In order to **write to a file**, we use the function fprintf( ) *(part of stdio)* where the first argument is the file pointer, fp:

```
fprintf(fp,"%s","Sample text.");

/* we should not forget to close the file after writing, fclose( ) takes as argument a
file pointer and closes the file referenced by the pointer. */

fclose(fp);
```

Function	Description
fopen	Opens a text file
fclose	Closes a text file
feof	Detects end-of-file marker of file
fscanf	Reads formatted input from file
fprintf	Prints formatted output to file
fgets	Reads a string from a file
fputs	Prints a string from a file
fgetc	Reads a character from a file
fputc	Prints a character to a file

String literal	Mode
w	Open for writing(file need not exist)
r	Open for reading (file must exist)
a	Open for appending (file need not exist)
r+	Open for reading and writing, start at beginning
w+	Open for reading and writing (overwrite file)
a+	open for reading and writing (append if file exists)

Multiparadigm Programming with Ruby

Classes, objects and message passing

In Computer Science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state. The term pure object oriented programming implies that all of the data types in the language are objects and all operations on those objects can be invoked by message passing. Sending a message to an object invokes a method by the receiver object. A message contains the method's name along with any parameters. In this chapter we will adopt the Ruby language.

Variables and aliasing

Aliasing:
Multiple variables referencing the same object.

Example:
person 1 = "Tony"
person2 = person1

The assignment of person1 to person2 does not create an object, it just gives it its reference. They both point to the same object.

You can avoid aliasing with "dup" which creates a new object.

person3 = person1.dup

Chain and parallel assignment statements

Ruby supports chaining of assignments:

```
a = b = 1 + 2 + 3
puts a # 6
puts b #6

a = (b = 1 + 2) + 3
puts a # 6
puts b # 3
```

Ruby supports parallel assignment:

```
a = 1
b = 2
a, b = b, a
puts a #=> 2
puts b #=> 1

x = 0
a, b, c = x, (x += 1) , (x += 1)
puts a #=> 0
puts b #=> 1
puts c #=> 2
puts x #=> 2
```

Arrays

We can create arrays using literals. A literal array is simply a list of objects between square brackets. As **everything is an object, this implies that an array can hold objects of different types**, as in the example below:

```
a = [ "number", 1, 2, 3.14 ] # Array with four elements.
puts a[0] # Access and display the first element. #=> number
A[3] = nil # Set the last element to nil.
puts a # Access and display entire array. #=> number 1 2 nil
```

We can also create an array by explicitly creating an Array object. Ruby allows us to specify array ranges, as in the example below:

```
myarray = [ 1, 2, 3, 4, 5, 6 ]
puts myarray [0] #=> 1
puts myarray[1 ... 3] # Exclusive range. => 2 3.
puts myarray[1 .. 3] # Inclusive range. => 2 3 4.
puts myarray[1,3] # Range between 1st up to 3rd consecutive, inclusive.
                  #=> 2 3 4.
```

Ruby allows a negative index, forcing the array to count from the end.

```
a = [ "pi", 3.14, "prime", 17 ]

puts a.class    # Array
puts a.length   # 4
puts a[0]       # pi
puts a[ -1]     # 17
puts a [4]      # nil
```

```
b = Array.new

puts b.class      # Array
puts b.length     # 0
b [0] = "a"
b [1] = "new"
b [2] = "array"
puts b            # a new array
```

Associative Arrays

An associative array (or **hash**) is an unordered collection of elements. An element is a pair of two objects: a **value and a key** through which the value can be retrieved. The value can be an object of any type. To store an element in an associative array, we must supply both objects:

hashName = { "key" => "value", ... }

We can retrieve the value by supplying the appropriate key:

hashName[" key"] => value

See example 6.

Classes

Naming conventions

local_variable
CONSTANT_NAME / ConstantName / Constant_Name
: symbol_name
@instance_variable
@@class_variable
$global_variable
ClassName
method_name
ModuleName

Objects

Example 7
The class keyword defines a class.
• By defining a method inside this class, we are associating it with this class.
• The initialize method is what actually constructs the data structure. Every class must contain an initialize method.
• @x and @y are instance (object) variables.
• puts and print write each of their arguments. puts adds a new line, whereas print does not add a new line.

A class can be instantiated with new as in:

p1 = Coordinate.new(0, 0)

Inheritance

Look at example 8.
Why have we provided a new class variable, newtotal, in the subclass? Ruby does not support hiding and it would not have considered variable total in class XYZCoordinate as a new variable. As a result, the output on the last statement above would have been 4, not 2.

Object Extensions

Ruby allows us to extend specific instances with new behavior.

Consider the following:

```
def p1.whatIam
   return "the origins on the 3D system."
End

puts p1.whatIam #=> the origins on the 3D system.
puts p2.whatIam #=> Will cause error
```

Control Flow

Ruby provides a rich set of control flow constructs to support selection and repetition.

If statements:

```
#if statement
if 5 < 3
 puts "5 < 3"
else
  puts "5 >= 3"
end

#ternary
5<3 ? (puts "5 < 3") : (puts "5 >= 3")

#unless
unless 5 < 3
 puts "!5 < 3"
else
 puts "!5 >= 3"
end
```

Unless statements

"You must take this course, unless you have already taken an equivalent one."
Basically same as if(not boolean)
To write in ruby, replace “if“ with “unless“

Else if statements

```
#multiple selection
if 5 < 3
 puts "5 < 3"
elsif 5 == 3
 puts "5 = 3"
else
 puts "5 > 3"
end
```

Switch statements

```
target = 3
case target
when 2,4,6,8,10
 puts "target #@target is even"
when 1,3,5,7,9
 puts "target #@target is odd"
else
 puts "target #@target is out of range"
end
```

Repetitions

```
$i = 0
$num = 5

#while
while $i < $num do
 puts "inside the loop = #$i"
 $i += 1
end

#until
until $i < 0 do
 puts "inside the loop = #$i"
 $i -= 1
end

#loop do
loop do
 $i += 1
 next if $i < 3
 print $i
 break if $i > 4
end
```

```
#for loop
for $i in ['fee', 'fi', 'fo', 'fum']
 print $i, " "
end

for $i in 1..3
 print $i, " "
end
```

Output:

```
inside the loop = 0
inside the loop = 1
inside the loop = 2
inside the loop = 3
inside the loop = 4
inside the loop = 5
inside the loop = 4
inside the loop = 3
inside the loop = 2
inside the loop = 1
inside the loop = 0
345fee fi fo fum 1 2 3
Process finished with exit code 0
```

Iterators

The keyword **each** returns successive elements of its collection

```
a = ["3.14", "number", "pi"]
a.each {|el| print el + " "}
```

Output:

```
3.14 number pi
```

The keyword **collect** takes each element from a collection and passes it to a block. The code below takes each element from the collection and displays its successor.

```
print ["H", "A", "L"].collect{|x| x.succ}
```

Output:

```
["I", "B", "M"]
```

The keyword **find** returns the first element from a collection which meets a condition. Otherwise it returns nil. The code below displays the first even number from a collection.

```
print [1,3,7,8,9,10].find{|x| x % 2 == 0}
```

Output:
8

Iterator based loops

```
#iterator based loops
3.times {|count| print count}
puts ""
1.upto(10) {|count| print count}
puts ""
10.downto(1) {|count| print count}
puts ""
0.step(10, 2) {|count| print count}
puts ""

for element in ['a', 'b', 'c']
 print element + " "
end
```

Output:
012
12345678910
10987654321
0246810
a b c

Regular expressions

In Ruby this is done with /pattern/. In Ruby, regular expressions are objects and can thus be manipulated as such. Some common pattern descriptions are shown below:

Pattern	Description
/Lisp|Lava/	Matches: Lisp or Lava
/L(isp|ava)/	Matches: Lisp or Lava
/ab+c/	Matches: a, followed by at least 1 b, followed by c.
/ab*c/	Matches: a, followed by zero or more b, followed by c
.	Matches: any character
/[Colloqui[um|a]]/	Matches: Colloquium or Colloquia

In the following example we are looking to extract and display lines which contain the word "punish."

```
#regex
string = "you bought a guitar to punish your ma"
string =~ /punish/ ? (puts string) : (puts "String does not contain the word punish.")
```

Output:

```
you bought a guitar to punish your ma
```

Access control

We can define access rights for features as follows:
Public methods can be called by anyone. Methods are public by default (except for initialize, which is always private, see below).

Protected methods can be invoked only by objects of the defining class and its subclasses.

Private methods can be called only in the defining class.

How to do it?

First way to do it:

```
class MyClass
 #default Access control is public
 def method1
   #...
 end

 protected #subsequent methods will b protected
 def method2
   #...
 end

 private #subsequent methods will be private
 def method3
   #...
 end

 public  #subsequent methods will be public
 def method4
   #...
 end
end
```

Second way to do it:

```
class Movie

 #same methods as above(not written to save space)

 #add this after writing the methods
 public :method1, :method4
 protected :method2
 private :method3
end
```

Modules

In OOP, the class provides the predominant unit of modularization. Some languages, including Ruby, further support modules. A module in Ruby can encapsulate constants and methods. A module cannot be instantiated and cannot form part of any inheritance hierarchy (i.e. cannot inherit and cannot be subclassified.)

See example 13 in handout.

The Math module in Ruby's standard library provides a rich set of methods. As one example:

```
puts Math.sqrt(9)   #=> 3.0
```

If you use a module a lot, you can include it in the top of the file to make your life easier to avoid repeating the name of the module.

```
include Math
puts sqrt(9)  #=> 3.0
```

Modules as mixins

Though Ruby does not support multiple inheritance, classes can import modules as mixins. In object-oriented programming languages, a mixin is a class that provides a certain functionality to be inherited by a subclass, but is not meant to be instantiated. Unlike with inheritance, a class cannot claim an is-a relationship with a mixin module. Ruby resolves name collision based on the lexical ordering of the inclusion of a module. The last module to be included hides all previous possible name collisions.

In general, mixins are useful for encapsulating behavior that is common to many objects in the class hierarchy, but cannot be factored into a common superclass.

See example 14 and 15 in handout.

***Java* and *Ruby* similarities and differences**

1. Java provides the notion of ***class*** as the primary decomposition axis. Additionally it provides an ***interface***. Ruby's primary decomposition axis is the ***class*** and additionally it provides a ***module***.
2. Java's ***classes*** can be ***abstract***. Ruby's **classes** cannot be **abstract**.
3. ***Interfaces*** are a mechanism to reuse specification only; ***Modules*** are a way to reuse implementation only.

Introspection

Introspection is the process by which of a program can observe (but not modify) its own properties, including its structure and behavior. A related term, reflection, is the process by which a program can observe as well as modify its own properties, including its structure and behavior. In Ruby, we can obtain the following type of knowledge about a program:

- What object it contains.
- The contents and behaviors of objects.
- The current class hierarchy.

For the following code:

```
require "CoordinateV2.rb"
require "XYZCoordinate.rb"

p1 = Coordinate.new(0,0)
p2 = CoordinateXYZ.new(0,0,0)

def p2.whatIam
 return "The origin on the 3D system."
end
```

We can execute some reflective queries to obtain knowledge about the system.

We can iterate over all instances of Coordinate in the system, posing a reflective query about each one. Let us inspect the system for objects of type Coordinate:

```
ObjectSpace.each_object(Coordinate) { |p|
 puts p.inspect
}
```

Output:

```
#<XYZCoordinate: 0x28455d8 @y=0, @x=0, @x=0>
#<Coordinate:0x2846028 @y=0, @x=0>
```

Note that an instance of XYZCoordinate is_a Coordinate, hence the listing of p2 in the output.

Contents and behaviors of objects

Read reflection last page of handout.

Functional Programming with Common Lisp (CL)

Terms

Arity: used to describe the number of arguments or operands that a function takes.

Unary: function (arity 1) takes one argument.
Binary: function (arity 2) takes two arguments.
Ternary: ...
N-ary: n arguments
Variable arity: any number of arguments

Prohibiting expression evaluation

(/ (* 2 6) 3) ;returns 4.
'(/ (* 2 6) 3) ;returns (/ (* 2 6) 3)

Boolean operations

Lisp supports:

- **and** - variable arity
- **or** - variable arity
- **not** - unary arity

TRUE = t
FALSE = nil

Constructing a list

3 ways to construct a list:

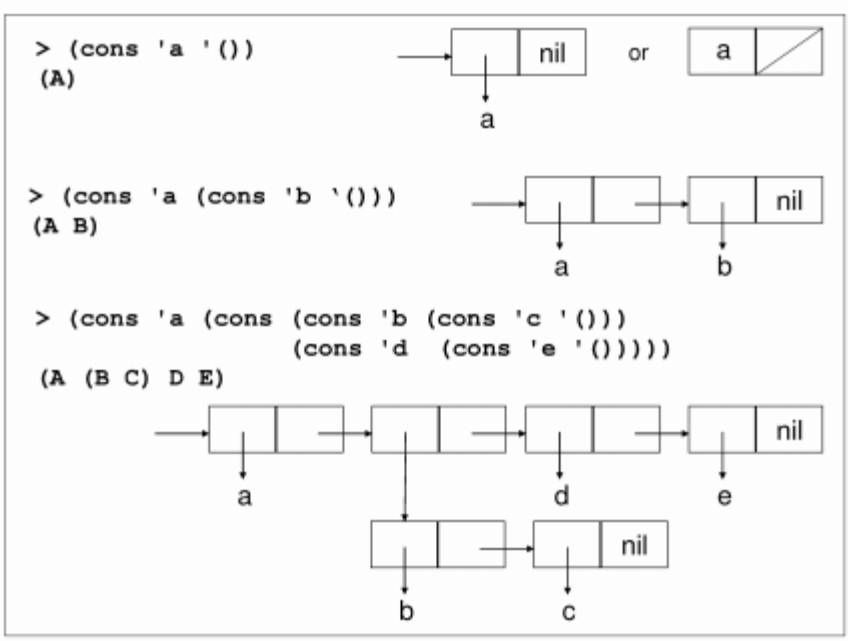
- cons: creates a list by adding to the head of an existing list
- list: creates a list comprised of arguments
- append: creates a list by concatenating existing lists.

cons

The function cons is a binary function: it expects two arguments, an element and a list. If an element is added to an empty list, then cons is essentially used to create a list, as in the first of the examples below:

(cons ‘a ‘( ))	Returns (a)
(cons 1 ‘(2 3))	Returns (1 2 3)
(cons ‘(1 2) ‘(3 4))	Returns ((1 2) 3 4)

A list in Lisp is singly-linked where each node is a pair of two pointers, the first one pointing to a data element and the second one pointing to the tail of the list with the last node's second pointer pointing to the empty list.



> (**cons** 'a (**cons** (**cons** 'b (**cons** 'c '( ))) (**cons** 'd (**cons** 'e 'C)))))
returns (A (B C) D E)

Mutability

Consider:
> (cons (+ 2 3) '(b c))
Is it syntactically correct? YES! Because it has 2 arguments, the second of which is a list.
returns (5 B C)

Now Consider:
> (cons ‘(+ 2 3) '(b c))
Since we added the ‘ before the parenthesis, it would process it as follows:
returns ((+ 2 3) B C)

Now consider:
>(cons a)
Is it syntactically correct? NO! Because of 2 things. (1) a cannot be evaluated and (2) there is only 1 argument.

Now consider:
>(cons ‘a)
Is it OK? NO! Needs 2 arguments.

Now consider:
>(cons ‘a ‘( ))
Is it OK? YES! We have an element and a list. The head will be A and the tail will be nil.
returns (A)

Now consider:
> (cons ‘a ‘(b c d))
Is it OK? YES! We have 1 element and a list.
returns (A B C D)

List

Function list takes any number of arguments and constructs a list comprised of these arguments. Function list has variable arity, i.e. it can take any number of arguments.

(list 1 2 ‘a 3)	(1 2 A 3)
(list 1 ‘(2 3) 4)	(1 (2 3) 4)
(list ‘(+ 2 1) (+ 2 1))	((+ 2 1) 3)
(list 1 2 3 (list 'a 'b 4) 5)	(1 2 3 (A B 4) 5)

Consider the following
> (list a 1)
Is is OK? NO! Because a cannot be evaluated. If we want to pass it as an element we need to add a quote.

Now consider:
> (list ‘a 1)
Is it OK? YES! Now we tell the compiler not to evaluate a.
returns (A 1)

Now consider:
> (list ‘a ‘( ))
Is it OK? YES! Lists can be nested and ( ) is just an empty list.
returns (A NIL)

> (list ‘a ‘( ) ‘( ) ‘( )) =>**returns** (A NIL NIL NIL)
>(length (list ‘a ‘( ) ‘( ) ‘( ))) =>**returns** 4

Now consider:
> (list ‘a)
This will create a singleton list (A)

Now consider:
> (list (a b) 3)
Is it OK? NO! Lisp will try to resolve (a b) but will fail.

Now consider:
> (list ‘(a b) 2)
Is it OK? YES! Will not attempt to evaluate a, b
returns ((A B) 2)

Now consider:
>(list (list ‘a ‘b) 2)
Is it OK? YES!
returns ((A B) 2)

Now consider:
> (list (cons ‘a (cons ‘b ‘( ))) 2)
Is it OK? YES!
returns ((A B) 2)

Append

Concatenation is the operation of joining two sequences of elements end to end. Concatenation can be applied to strings or lists. In the latter case, we can demonstrate the operation of concatenation with the following example:

concatenate((a,b),(c,d)) → (a,b,c,d)

Function append constructs a new list by concatenating any number of lists that are supplied as its arguments. Much like list, function append has variable arity, i.e. it can take any number of arguments. **There is a restriction on the types of its arguments: they must all be lists.**

(append ‘(1 2) ‘(3 4))	**returns** (1 2 3 4)
(append ‘(1 2 3) ‘( ) ‘(a) ‘(5 6))	**returns** (1 2 3 A 5 6)
(append ‘(1 2 3 ‘(a b c)) ‘( ) ‘(d) ‘(4 5))	**returns** (1 2 3 ‘(A B C) D 4 5)

Consider:
>(append 1 ‘(4 5 6))
This will give an error because 1 is not a list and append only takes lists as arguments.

We can to create list (1 2 3 4) with append. How?
We need to transform 1 to a list first. The following will work.
>(append (list 1) '(4 5 6)) → (1 4 5 6)

Consider:
>(append (cons 'a '()) (list 'b 'c))
Returns: (A B C)

NOw consider:
>(append (list 'a '(c d))(cons 'f (list 'g (cons 'k '()))))
Returns: (A (C D) F G (K))

Accessing a list

We can only access either the head of the list or the tail of the list. Hence, only 2 operations are available: car and cdr. car returns the first element and cdr returns the tail element.

car

Operation car takes a list as an argument and returns the head of the list. Note that the head of a list can be **either an atom or a list**. For example:
(car '(a s d f)) → returns a.
(car '((a s) d f)) → returns (a s)

cdr

Same goes for cdr as for car, but returns the tail(everything other than the head).
(cdr '(a s d f)) → returns (s d f)
(cdr '((as) d f) → retuns (d f)
(cde '((as)(df)) → returns ((d f))

So how do we access the second element?
(car (cdr('(1 (3 5) (7 11)))) → returns (3 5)

Now consider:
>(cae (list '() '(a b c)))
returns NIL.

Append vs list empty values:
(append '() '() '()) → NIL
(list '() '() '()) → (NIL NIL NIL)
Appends actually looks inside to concatenate the elemnds in its arguments, and sees there is nothing there o returns NIL. List creates a list with all the values entered even if NIL, therefore makes 3 NIL values in a list.

Consider:
(append (list 'b '(d e) (* 2 3)) (cons '(+ 2 3) (list 'f (cons 'g '( ))))))
Working from inside
We need to append (list ‘b ‘(d e) (* 2 3)) to
(cons ‘(+ 2 3) (list ‘f (cons ‘g ‘( ))) → ‘(+ 2 3)(f(g))
Where (list ‘f (cons ‘g ‘( )) → (f (g))

Return → (B (DE) 6 (+ 2 3) F (G)) with a length of 6.

Consider:
(list (append ‘(+ 1 4) ‘( ) (list ‘( ) ‘( ))) (cons (+ 1 4) (list ‘a (cons (+ 1 7) ‘( )))))
returns → ((+ 1 4 NIL NIL) (5 A (8))). Its length is 2.

Consider:
(car (cdr (cdr (append (list ‘( ) ‘(a)) (cons ‘b (list (+ 2 3 4)))))))
Returns → B

Consider:
(car (cdr (cdr (append (append '( ) '(a) '( )) (list 'b '( ) (cons (+ 3 4) '( )))))))
Returns → NIL

Predicate functions

A function whose return value is intended to be interpreted as truth or falsity is called a **predicate function**. The built-in function listp returns true if its argument is a list. For example,

(listp '(a b c)) → Returns true (T).
(listp 7) → Returns false (NIL).

Other common predicate functions include:

Predicate	Description
(numberp *argument*)	Returns true is argument is a number
(zerop *argument*)	Returns true is argument is zero
(evenp *argument*)	Returns true is argument is even
(oddp *argument*)	Returns true is argument is an odd number

Advanced mathematical operations

Lisp provides a number of built-in advanced mathematical operations. For example, (sqrt a), (expt a b) and (log a) returns the natural logarithm of a.

Control flow

The simplest single conditional is if:

```
(if testExpression
    thenExpression
    elseExpression)
```

```
(if (listp '(a b c)) ;If (a b c) is a list…
    (+ 3 7) ;…then evaluate this expression,
    (+ 1 3)) ;…else evaluate this one.
```

Multiple selection can be formed with a Gond expression which contains a list of clauses where each clause contains two expressions, called question (condition) and answer. Optionally, we can have an else clause.

```
(cond (question answer)
    ...
    (else answer)) ;Optional
```

Variable and binding

Binding is a mechanism for implementing lexical scope for variables.

let:
The let syntactic form takes two arguments: a list of bindings and an expression (the body of the binding) in which to use these bindings.

The let values are computed and bindings are done in parallel, which requires all of the definitions to be independent. In the example below, x and yare let-bound variables; they are only visible within the body of the let.

```
(let ((x 2) (y 3))
    (+ x y)) ;return 5.
```