Languages Exam

Sam Auciello

20 March 2013

Question 1:

As a way to demonstrate your understanding of programming ideas, discuss the concepts behind following programming buzz words across at least three languages that you're familiar with that allow different programming styles, perhaps C, Python, and a Lisp.

(a) First organize the terms into groups of concepts, showing which are variations of the same concept or idea across or within languages, or closely related concepts, or opposites.

(b) Then for each of these concept groups, discuss the ideas behind that group, and give concrete code snips across these languages to illustrate them.

Be clear that I *don't* want you to just define each of these words; instead, I want you to use them as the starting point for a conversation with examples about some of the fundamential notions of how programming languages work, and how those notions vary from language to language.

In alphabetic order, the words are

API
argument
array
bind
callback
class

closure
collection
comment
concurrent
compiled
data structure
dynamic
exception
fork
function
functional
global
hash
immutable
imperitive
inheritance
interface
interpreted
iterate
lexical
link
list
lazy
lexical
macro
method
name
namespace
object
overload
parse
scope
package
pattern
pass by reference
pass by value
pointer
recursion
side effect
stack overflow

```
static
symbol
syntactic sugar
thread
test
throw
type
variable
vector
```

Control flow

Programming languages need mechanisms for designating which instructions are executed when. The most common forms this takes are conditionals and iteration. Conditionals are used to determine whether a section of code should execute and iteration is used to execute code multiple times. Control flow can also take the form of functions and function calls. Functions can be thought of like mathematical functions that map a set of inputs onto a set of outputs but in the context of control flow it is usually best to think of them as pieces of code that can be broken out and reused. It is often useful to use functions to break up code even when a given function will only be called once in the execution of the program simply because thinking in terms of smaller reusable, general purpose functions makes code easier to read and much easier to modify later. Compare the following two solutions to the same problem in Ruby:

```
print "what is your email address? "
email = gets.strip
if /.+@.+\..+/.match email
  puts "your email is #{email}"
else
  puts "that isn't an email"
end
```

and

```
def getEmail
  print "what is your email address? "
  return gets.strip
end
```

```
def validateEmail email
  return /.+@.+\..+/.match email
end

email = getEmail
if validateEmail email
  puts "your email is #{email}"
else
  puts "that isn't an email"
end
```

For something this simple the first is probably best but if any of the steps gets significantly more complex (for example we could imagine wanting to check that the email address ends in a real top-level domain) then the second style starts to become much nicer to work with.

Another way of breaking code out into smaller modular pieces is macros. Macro can refer to different things in different contexts but a macro generally is a way of making a sort of meta statement to the language of the form "when I say X what I really mean is Y". For example C macros can be used to intstruct the compiler to rewrite sections of code before the rest of compilation begins:

```
#define SWAP(a, b) (a)^=(b);(b)^=(a);(a)^=(b);

...

if (swap_needed(x, y)) {
  SWAP(x, y)
}
```

The compiler simply replaces the macro call in the source code with the code to be substituted in. In this case the code is far more legible if it simply says swap then if it spelled out the swapping process. This could of course have been done with a function call but in C a macro can sometimes have performance benefits in these cases because each function call requires additional memory allocation.

In lisp macros work somewhat differently. A lisp macro can be used to define a completely new syntax and unlike C macros which use a completely separate language to define them, lisp macros are written in lisp. Lisp macros also allow you to control when code is evaluated. In a normal function call in Common Lisp like

```
(defun foo (x) (+ 1 x))
(foo (* 2 4))
```

the arguments of the function are evaluated before the funciton call. So in the above example all that the function `foo` sees is the result of the multiplication: 8. Inside of a lisp macro, you can control exactly when things are evaluated for example:

```
(defmacro foo (x)
  `(progn
    ,(when (eq '* (car x))
      `(format t "argument was a multiplication"))
    (+ 1 ,x)))
```

The ` symbol tells the Common Lisp interpreter that what follows shouldn't be evaluated right away except for the inner pieces preceded by `,`. This macro takes an unevaluated expression x and returns the code to add one to the result of the expression being evaluated preceded by a print statement reporting that the expression began with an asterisk if it did. A more practical example of a macro might be a debugging statment:

```
(defmacro debug (x) `(format t "~a: ~a" ',x ,x))
(setq foo 1)
(debug foo) ; FOO: 1
```

Because the macro can control exactly when it's arguments are evaluated it has access the unevaluated form of the argument and can, in this case, print it out as a label. Delaying evaluation in this way is sometimes called lazy evaluation. In some languages, like Haskell, all evaluation is delayed as long as possible.

Another useful form of control flow is exception handling. Exception handling allows programs to handle problem situations gracefully. For example in Python:

```
def func_that_cant_handle_zero(arg):
  if arg == 0:
    raise Exception("Everything is terrible!")
  return "normal results"

try:
  func_that_cant_handle_zero(0)
```

```
except Exception:
  print "bad things happened"
```

Even though our function recieved input that it didn't know what to do with, we can account for the problem using a try/except statement rather than simply crash the program.

It many programming environments it is desireable and possible to have multiple lines of code running at the same time. This is called concurrency. One way of doing this is with a fork. A fork statement tells the currently running process to split into separate processes that have no straightforward way of communicating with one another and have access to all of the information that the parent process had prior to that point. One major advantage of this approach is that forked processes can have access to copies of the same data structures which they can then manipulate without worrying about how it effects the other process. Another common way of handling concurrency is called threads. Threads allow sections of code to be run at the same time within the same program. They can be much faster than forks because they don't require everything the process has access to to be duplicated and the have the advantage of sharing access to the same data (not just duplicates). They also have the disadvantage of sharing access to the same data. It becomes important to worry about the precise order inwhich things can happen and being very careful not to make any assumptions about what has happened already in another thread.

Assignment

Programs often keep track of many different sorts of data at once. It is vitally useful to be able to map different pieces of data to helpful names so that the data can be referred to later. Languages have many ways of doing this. The most common is simple variable assignement where some bit of data called a value gets bound to a variable name:

```
# Ruby or Python
name = "value"
// JavaScript
var name = "value"
// C
char* name = "value"
;; Scheme
(define name 'value)
```

The `var` in JavaScript is optional but without it the variable is treated as global which is usually wrong (more on that shortly). The `char*` in C specifies the type of variable. In this case a pointer to a character (the asterisk designates a pointer). In C strings are stored as sequencial characters terminated by the `NULL` character `'\0'` and stored in variables as a pointers to the first character. The `'` in Scheme designates that following symbol shouldn't be evaluated. In this case both `name` and `value` are symbols and second is bound to the first so that the first evaluates to the second.

Variable assignment can also take the form of argument passing. In this case the names are specified when a function is defined and the values are specified when the function is called:

```
// C
void my_function(int x, int y) {
  printf("x is %d and y is %d\n", x, y);
}

void main() {
  my_function(2, 3); // x is 2 and y is 3
}

;; Scheme
(define (my-function x y)
   (format #t "x is ~a and y is ~a" x y))

(my-function 2 3) ; x is 2 and y is 3
```

In both cases the arguments are treated just as bound variables for the purpose of that function call. The `int`s in C are required and specify the type of the argument which, because C is statically typed must be known beforehand. The special form of `define` seen here in Scheme is syntactic sugar for:

```
(define my-function (lambda (x y)
   (format #t "x is ~a and y is ~a" x y)))
```

In this case `my-function` is a symbol that is being bound to this lambda function.

In larger projects it is often necessary to limit which names are accessible in which contexts. These contexts are called scopes or namespaces. Scopes

are often nested so that names from the outer scope are accessible from the inner scope but names from the inner scope are hidden from the outer scope. For example in JavaScript:

```
var foo = 1, bar = 2;
(function() { // functions form scopes in JavaScript
  var foo = 3, baz = 4;
  console.log(foo, bar, baz); // 3 2 4
})()
console.log(foo, bar); // 1 2
console.log(baz); // ReferenceError: baz is not defined
```

If a JavaScript variable is set (e.g. `foo = 1`) without being initialized with the `var` keyword then it is put in the outermost global namespace. This is bad because it means that forgetting the word `var` in one place can cause variables to have unexpected values anywhere in your code. Not all languages define scopes this way. In Ruby method scopes don't nest this way (functions in ruby are called methods):

```
foo = 1
def bar
  print foo
end
foo() # NameError: undefined local variable or method ‘foo’

def foo
  bar = 2
end
foo()
bar # NameError: undefined local variable or method ‘bar’
```

Insted of having nested function scopes, ruby has nested class and objects scopes. Ruby makes use of th `@` sigil to denote instance and class variables so:

```
class Foo
  @@bar = 1 # these are class variables
  @@qux = 2
  def baz
    print @@bar
    @@qux = 7
```

```
  end
  def snap
    print @@qux
  end
end
Foo.snap() # 2
Foo.baz() # 1
Foo.snap() # 7
```

Ruby also uses the `\$` sigil to denote global variables. I find Ruby's approach to scope to be really nice. It assumes that all varaibles are only needed in the local scope unless a sigil specifies otherwise.

Sometimes it is useful to have a lot of names/value associations wrapped up in a specific isolated context that can be passed around as a data structure. This is precisely what a hash is; a set of key/value pairs that is treated as a single value. In Python it is called a Dictionary and in JavaScript it is synonymous with object.

Types

Computer programs handle and use data and data typically requires structure. Types are a way of classifying data so that the program knows how to interpret it. For example, in C, the series of four bytes:

```
00000000 11011110 11011110 1100110
```

could represent the integer 7303014 or the string `'foo'`. Types typically come in two categories. Atomic types like integers and booleans are just simply a data of that type. Structured types like arrays, instances, and hashes can contain other types of data. For example, you could have an array of booleans, a hash mapping strings to numbers, or even an array of arrays of hashes. Low level languages like C allow you to interact directly with the actual machine representations of these types in memory which has the advandage of allowing you to fully control the way that data is stored in memory. This also requires you to keep track of the way that the data is stored in memory. High level languages like Ruby often have complex dynamic structures for storing arbitrary data. For example, anywhere you can put a value in Ruby, you can put a value of any type and the language will figure out how to represent that in memory wihtout you needing to worry about it.

```
x = 10
```

is just as valid as

```
x = [10, 20, ["foo", true, nil]]
```

This makes Ruby a dynamically typed language. Dynamically typed languages have the advantage of flexibility. This flexibility can however cause bugs if for example a function is was designed to take an integer as an argument but instead is passed a null value. Considder the following in Ruby and in Java:

```
// Ruby
def doMath x
  return 10 * (x + 2) - (x / 3)
end

numbers = {
  :one => 1,
  :two => 2,
  :three => 3,
  :four => 4,
  :five => 5
}

doMath numbers[:six]
```

```
// Java
public int doMath(int x) {
  return 10 * (x + 2) - (x / 3);
}

public void main() {
  Map<string, int> numbers = new HashMap<string, int>();
  numbers.set("one", 1);
  numbers.set("two", 2);
  numbers.set("three", 3);
  numbers.set("four", 4);
  numbers.set("five", 5);

  doMath(numbers.get("six"));
}
```

In Ruby you would get a relatively unhelpful error message about there being no `'+'` method for `nil`. In Java you would actually get an error message for the `.get()` call when no entry is found for `'six'` but even if you simply called `domath(null)` you would get a compile time error about the wrong type being passed. This is what is meant when Java is referred to as type safe. The programmer has to specify the type of everything but the result is that the program won't compile unless all of the types are correct. The result is that there are fewer bugs and more reduntant text in the code.

Object-oriented Programming

There are several ways to think about object-oriented programming. One way is to think of classes as user defined types. Many languages embrace this idea. For example in Ruby, the built-in types are, themselves classes which can be modified just as easily as user defined ones:

```
class Integer
  def double; self*2; end
end

puts 10.double # prints "20"
```

This is called a monkey patch or sometimes, "duck punching". Python allows for a similar programming pattern but unlike Ruby, Python won't let you directly modify the built-in types instead requiring that a new class be defined that inherits from the built-in type:

```
class MyInt(int):
  def double(self):
    return self*2

print MyInt(10).double() # prints "20"
```

Inheritance here allows a subclass to take on all of the characteristics of it's superclass (methods, properties etc.) and then redefine or add new ones. An instance of the `MyInt` class here behaves in all ways just like a normal Python integer except that it also has a double method. When a method from the superclass is redefined this way it's sometimes called overloading. In Python all objects inherit from the default object which defines some methods like `__repr__` which is called when the string is printed out. Overloading allows Python classes to have custom representations:

```
class Foo:
  pass

class Bar:
  def __repr__(self):
    return "[BAR INSTANCE]"

print Foo() # <__main__.Foo instance at 0x100c658c0>
print Bar() # [BAR INSTANCE]
```

JavaScript has a fairly unusual approach to objects. Most modern scripting languages have some form of key/value association type. Ruby and Perl call them hashes, PHP calls them associative arrays, and Python calls them dictionaries. Javascript simply calls them objects. The can be created as literals like in Python or Ruby:

```
# Python
x = { "foo": 1, "bar": 2 }
# Ruby
x = { "foo" => 1, "bar" => 2 }
// JavaScript
var x = { foo: 1, bar: 2 }
```

Unlike Python or Ruby, JavaScript uses prototypal ineritance as opposed to classical inheritance. With classical inheritance classes can inherit from other classes and objects can be instances of classes. With prototypal inheritance objects simply inherit from other objects. For example:

```
# Ruby
class Person
  attr_accessor :first_name, :last_name
  def initialize *args
    @first_name, @last_name = args
  end
  def full_name
    "#{@first_name} #{@last_name}"
  end
end

joe = Person.new "Joe", "Smith"
puts joe.full_name # Joe Smith
```

```
// JavaScript
var defaultPerson = {
  full_name: function() {
    return this.first_name + " " + this.last_name;
  }
};

var joe = Object.create(defaultPerson);
joe.first_name = "Joe";
joe.last_name = "Smith;
console.log(joe.full_name());
```

The `Object.create()` call here returns a new object that inherits from the passed object. Because the object simply inheritted from another object the properties of that object can later be changed.

Funcitonal Programming

Functionaly programming is about writing functions that have no sides effects. This means that each function only interacts with the rest of the program via arguments passed in and return values. Such functions can be seen as mathematical functions that map a set of inputs onto a set of outputs.

Where an imperative program is a sequence of instructions to be followed in order, a functional program is a collection of well defined transformations built up from one another with a final outer function that transforms the program's input into the program's output. One of the greatest advantages of this approach is that small well defined functions are much easier to test and debug than large unweildy ones. Python's doctests can be very helpful for these sorts of tests:

```
def addOne(x):
  """
  Example:
  >>> addOne(2)
  3
  """
  return x + 1
```

It is common for recursion to be used in place of iteration in functional languages. For example, given the problem of determining whether a list contains a given element in Python one might do the following:

```
def contains(list, element):
  for e in list:
    if e == element:
      return True
  return False
```

Whereas in Scheme, it would be more common to see:

```
(define (contains l element)
  (cond
   ((null? l) #f)
   ((= element (car l)) #t)
   (#t (contains (cdr l) element))))
```

Callbacks are also a very common pattern in functional programmming. The idea behind a callback is that a function can take another function as one of it's arguments and call that function when it's done. Often passing the results of the first function to the second instead of returning them. Node.js uses this technique to guarantee that input and output operations are non-blocking. For example considder the common use case of querying a database for some data and sending it to the user as JSON. In a traditional web server environment like PHP, the process would be frozen while the database was processing the request:

```
// PHP
$sql = "SELECT stuff FROM tables";
$query_result = db_query($sql); // execution is stopped here
echo json_encode($query_result);

// go back to serving other requests
```

In Node.js this problem is solved using callbacks:

```
// Node.js
var sql = 'SELECT stuff FROM tables';
db_query(sql, function(query_result) {
  serve_request(JSON.stringify(query_result));
});

// go back to serving other requests
```

Because the `db\_query()` function returns immediately, serving other requests can resume immediately. Behind the scenes, Node.js has a pool of threads that it uses to handle the actual database querying. Because JavaScript functions have closures the callback will have access to the scope inwhich it was defined which makes Node.js work particularly well.

APIs

An api, or application interface is an interface to a section of code. The api hides irrelevant implementation details so that the progammer can focus on the parts that matter. For example in a Ruby program, if I need to sort a list of numbers I don't need to know what sorting algoritm is being used. I merely need to know how to interface to the built-in library that does sorting.

```
# implementation
class Array
  def sorted?
    (size - 1).times do |i|
      return false if self[i] > self[i + 1]
    end
    return true
  end
  def sort
    result = shuffle
    return result if result.sorted?
    return sort
  end
end

# api
numbers = [60, 99, 61, 26, 82, 19, 44, 76, 29, 23]
sorted = numbers.sort
```

I don't need to see the implementation to know how to use it. All I need to know how to use it is that Arrays have a `.sort` method that takes no arguments and returns a sorted copy of the Array. These pieces of information are the api. In this case it may also be worth knowing that the builtin quicksort implementation has been overloaded with the factorial time bogosort algorithm since this will be much slower than expected.

Apis are most useful in general purpose libraries. For example, the node.js package optimist is a useful library for parsing command line ar-

guments. If you needed to know how it worked in order to use it then it would hardly be worth using it at all as you could simply make your own. Instead it provides an api for it's use. It can then be treated like a black box. As long as you know which methods to call with which arguments and what they will do, you can ignore the details of how.

A more complex example of an api is the Google Maps API. Google Maps is a big, fully featured web application for using maps. The inner workings of the application are controled by Google but they expose the API as a system by which web developers can embed maps on web pages and manipulate them in JavaScript. The web developer needed understand all of the implementation details of the map application as long as they understand how to use the mechanisms provided to manipulate the resulting maps.

Question 2:

```
Write six programs implementing solutions to the following two
problems across the three languages with different styles.
(These may be the same three from question 1, but don't need
to be.)

In each case, include docs and tests appropriate to the style of that
language, including explicitly what verision of what language you ran,
in what environment, what steps compiled and/or ran the code, and what
the input and output looked like.

Use these programs to illustrate some different currently popular
programming paradigms, as well as your mastery of the vernacular
within these programming language communities.

As a postscipt, discuss which languages you found well suited
to which problem, and why.

The two problems are

A) the perfect squares crossword puzzle

   Replace the * below with twentyfive base 10 digits to form a
   crossword-like array of thirteen 3-digit perfect squares, with each
```

```
   3-digit number reading across or down.  (121 = 11^2 for example is
   a 3-digit perfect square.)

      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

B) family tree

   Write a program to generate a visual family tree from a .csv (comma
   separated value) file of people.

   Each line in the file should represent a person, and include at
   least (name, father, mother, date born, date died). The data format
   is up to you, but should be (a) well defined, and (b) allow for
   multiple people with the same name.  Generate some (fake) data to
   run your code on, which includes at least 10 people across at least
   3 generations.

   The family tree should be either ascii art or easily displayed
   image (e.g. .png, .pdf, .svg, .html, ...) as you choose. You may
   use an external graphics library appropriate to the language; if
   so as usual quote your sources explicitly.
```

Square Crosswords

I very quickly found an answer to problem by inspection assuming squares can occur multiple times in the solution. My solution relies on palindromic squares to create a highly symetrical solution using 11^2, 12^2, and 22^2:

```
1   1 2 1   1 2 1   1
4 8 4   4 8 4   4 8 4
4   4 8 4   4 8 4   4
```

I decided to try answering the more difficult quesiton of whether this can be solved using each square at most once. I was able to find the following solution using a recursive search in ruby:

```
$ cd crosswords
$ ruby recursive_search.rb
841
```

```
484
144
169
441
961
625
256
225
676
576
729
196
```

I transccribed this by hand to the folowing:

```
8   1 6 9   2 2 5   1
4 8 4   6 2 5   7 2 9
1   4 4 1   6 7 6   6
```

My ruby code makes use of Ruby classes to create scopes that fully encapsulate the calculation. This isn't really necessary for an application with so few moving parts but it seemed like the natural thing to do in classical object-oriented language like Ruby. This particular script has two major flaws. Firstly, the logic of when a given square is allowed to fit in a particular space is coded in ad hoc manner which is excessively verbose and somewhat hard to read as well as being inellegant and non-general. Secondly, not having any of the structure of the puzzle built into the program there was no obvious way to translate the solution from the form it is stored in (an array of strings in an arbitrary order) to the crossword format it appears in above. Seeing as by this point it was clearly a flawed first pass, I did the translation by hand and moved on to a better approach in Hot Cocoa Lisp.

My Hot Cocoa Lisp program has a hardcoded list of spaces that will need to contain a digit from two different squares. It then uses this information to automatically constrain the search. This makes it more general and legible than the previous iteration but it still leaves no clear way to translate the output to crossword form.

```
$ hcl recursive_search.hcl
$ node recursive_search.js
[ '841', '484', '144', '169', '441', '961', '625', '256', '225',
  '676', '576', '729', '196' ]
```

I wrote the final version in C and used a somewhat object oriented approach involving structs to organize the puzzle. In this version I kept track of the solution in a 3x11 grid of digits to make sure the output could straightforwardly be made to look like a crossword. I then hardcoded 13 space structs each of which contains the coordinates of the three digits in that space and a bit map denoting which of those digits will have been filled in before this space is assigned a square.

```
$ gcc -Wall recursive_search.c -o recursive_search
$ ./recursive_search
 8   1 6 9   2 2 5   1
 4 8 4   6 2 5   7 2 9
 1   4 4 1   6 7 6   6
```

Family Trees

I wrote a ruby script called *gen_ csv.rb* that generates a random family and stores it in *people.csv*:

```
$ cd family_trees
$ ruby gen_csv.rb
$ cat people.csv
id,first_name,last_name,father,mother,born,died
0,Christy,Jackson,7,8,1928,2010
1,Stacey,Taylor,5,6,1942,-1
2,Claudia,Jackson,1,0,1973,-1
3,Jason,Taylor,1,0,2005,-1
4,April,Jackson,1,0,1986,-1
5,Martha,Taylor,9,10,1903,1971
6,Thelma,Bass,-1,-1,1904,1965
7,Fred,Jackson,-1,-1,1888,1948
8,Joann,Beck,-1,-1,1897,1987
9,Wade,Taylor,17,18,1865,1926
10,Zachary,Currie,-1,-1,1871,1943
11,Patricia,Jackson,7,8,1935,2002
12,Anna,Haynes,-1,-1,1924,2013
13,Gail,Haynes,12,11,1991,-1
14,Valerie,Currie,9,10,1912,1996
15,Dan,Haynes,11,12,1997,-1
16,Pamela,Bass,6,5,1938,1990
```

```
17,Regina,Taylor,21,22,1829,1886
18,Sylvia,Jacobs,-1,-1,1829,1927
19,Barbara,Newton,-1,-1,1926,-1
20,Jeanne,Bass,19,16,1989,-1
21,Lewis,Taylor,-1,-1,1789,1840
22,Mark,Simon,27,28,1796,1858
23,Max,Taylor,17,18,1876,1967
24,Frederick,Taylor,1,0,1985,-1
25,Johnny,Currie,10,9,1911,1974
26,Renee,Bass,19,16,1973,-1
27,Katie,Simon,29,30,1757,1842
28,Joyce,Bond,-1,-1,1760,1844
29,Justin,Simon,-1,-1,1724,1786
30,Victor,Schneider,-1,-1,1718,1785
```

The first tree generating program I wrote was in Ruby. I used a simple scripting approach to generate a graphviz file and complile it to a .png file using `dot`.

```
$ ruby display_tree.rb
$ dot -Tpng tree.graphviz > tree.png
```

This approach seemed too easy so I decided to try making an ascii version in C. Unfortunately intelligently rendering a complex directed graph in two dimensions turns out to be a fairly non-trivial algorithmic problem and it seemed like an poor use of my time to learn and re-write the dot algorithm (or even worse invent my own) so instead I made a console based family tree explorer:

```
$ gcc -Wall display_tree.c -o display_tree
$ ./display_tree

Christy Jackson (1928 - 2010)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Stacey Taylor (1942 - present)
Children:
Frederick Taylor (1985 - present)
April Jackson (1986 - present)
```

```
Jason Taylor (2005 - present)
Claudia Jackson (1973 - present)

warning: this program uses gets(), which is unsafe.
Enter a relative to navigate to (mother, father, spouce, child_n): father

Fred Jackson (1888 - 1948)

Mother: N/A
Father: N/A
Spouce: Joann Beck (1897 - 1987)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): spouce

Joann Beck (1897 - 1987)

Mother: N/A
Father: N/A
Spouce: Fred Jackson (1888 - 1948)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): child_0

Patricia Jackson (1935 - 2002)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Anna Haynes (1924 - 2013)
Children:
Dan Haynes (1997 - present)
Gail Haynes (1991 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): mother

Joann Beck (1897 - 1987)
```

```
Mother: N/A
Father: N/A
Spouce: Fred Jackson (1888 - 1948)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): child_1

Christy Jackson (1928 - 2010)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Stacey Taylor (1942 - present)
Children:
Frederick Taylor (1985 - present)
April Jackson (1986 - present)
Jason Taylor (2005 - present)
Claudia Jackson (1973 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): child_3

Claudia Jackson (1973 - present)

Mother: Christy Jackson (1928 - 2010)
Father: Stacey Taylor (1942 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): quit
```

For my third version I simply re-wrote the tree explorer in Hot Cocoa Lisp with a more functional style.

```
$ hcl display_tree.hcl
$ node display_tree.js

Christy Jackson (1928 - 2010)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
```

```
Spouce: Stacey Taylor (1942 - present)
Children:
Claudia Jackson (1973 - present)
Jason Taylor (2005 - present)
April Jackson (1986 - present)
Frederick Taylor (1985 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): quit
```

Question 3:

Discuss the strengths and weaknesses of these programming languages as you see them. What sorts of problems or situations are good fits to these languages, and why? Which do you personally like, and why? Be specific, giving examples that justify your comparisons and conclusions. (This may well cover some ground you've already discussed in the previous two problems. If so, you don't have to repeat any of that, just refer back to it and bring up anything that you feel hasn't yet been brought forward.)

I feel that Python works quite well as a teaching language; even if for no other reason than that it forces new programmers to properly indent their code. It also has a better selection of good libraries to do INSERT THING COMPUTERS DO HERE than most modern scripting languages, making it a good choice for a lot of practical applications. In general I get tired of the little problems with Python. The one that irks me the most lately is the limitted nature of lamdas. This email: http://mail.python.org/pipermail/python-dev/2006-February/060621.html from Python's designer Guido van Rossum explains why he has no intension of changing this. He begins by claiming that there is no reasonable way to make the syntax work. This is clearly ridiculous:

```
lambda(arg1, arg2):
  statement one
  statement two
```

It basically seems to boil down to him not wanting Python to be like lisp. I don't know what he thinks makes Python better than lisp but either way I find that the more I program the more I want to be programming functionally and the less I like Python.

I rather like Ruby for object-oriented programming and general scripting. The block passing structure has a way of making simple tasks simpler and more complex tasks surprisingly manageable. It can be very terse which I like because it means less extraneous typing and more expressive power. The way that Ruby treats objects seems very much like the way Java treats objects to me. The two largest differences between the two languages seem to be a) that Java is strongly typed and Ruby is duck typed, and b) that Ruby is much newer and has a lot more helpful features. The biggest flaw I see with Ruby is that it doesn't really have functions. It has methods which are necessarily attached to classes and objects (and if you embrace this then the language can be quite powerful). It has code blocks which can be passed to functions but aren't really functions in that they can't be treated as values. It has procs and lambdas that effectively are functions but they are so far removed from the normal use case that their syntax is obscure and forgettable.

I find Lisp-like languages to have a syntax particularly conducive to functional programming. Since every piece of the language has a consistent syntax it's very easy to think in terms of function calls (in a way everything is). In some ways the entire point of monads is that every deterministic operation (inside a computer or otherwise) is just a transformation from one state of the universe to the other.