Problem 3: Memo: Java [redacted]

The language that I am most likely to chose to use is Java. A lot of this is because it is one of the first languages that I learned and the one that I have used the most, but there are a number of reasons why I have stuck with it. It is easy to develop in coming from a well-defined syntax, type system, and excellent documentation and portability.

In my opinion, the syntax of Java is very straightforward. Rarely is there confusion in what the code means or which part of the language to use, making obfuscation more difficult (although not impossible). While the syntax is very similar to C, there are a number of simplifications, especially with respect to pointers, that reduce common errors. Even though this forfeits some of the control one has over how the program executes, possibly slowing the runtime down even more than the overhead of the JVM, I find that the benefit of having fewer obtuse runtime problems makes the act of coding a lot more efficient for me. If one needs to have a speedier program, which occurs less and less with the increasingly fast processors of today, converting to C or C++ is not too difficult. Conversely, anyone familiar with those languages should be able to rapidly pick up Java. Additionally, the sequential format of such an imperative language also translates smoothly from how I think of an algorithm: a list of steps. Of course, it is often useful to rework the problem into another paradigm such as functional programming, but when I just want to prototype it is often the fastest way.

Types in Java are also have great utility in avoiding errors and making the code extensible. Strictly enforcing type rules catches innumerable problems in my code or in my logic as opposed to a non-strict typed language like Ruby, where it is easy to lose track of what certain variables are over the course of the program. The object-oriented nature of the language also naturally harmonizes with how I visualize systems with the interconnectivity via inheritance and containment analogous to the ideas of subsets and parameters. Also, to me it is an easy way to think of how to reuse code.

Distributing Java code to users with differing setups is rarely a problem. There is a large amount of functionality already included in the basic Java language, like GUI code and networking code, meaning that much of the time, there is little need for any extra libraries. This means that just having Java installed lets many diverse applications run without complications. Even if one needs to add a specific library, there are many open source libraries such as Apache commons that can drive the cost down so that anyone with a computer can develop what they want. Using JavaDocs also enhances the usability of the code, as I have rarely had issues finding what I needed to know from the online Java API.

Improve title
Be descriptive
non technical
awk.
word choice
very important but not about syntax!
not just a matter of speed - memory safety eliminates whole class of errors & exploits!
contradicting
swap order to follow outline suggested by first P.

Be descriptive in title: why is it superior?

The Superiority of OCaml

(Compared to any other language I use)

[redacted]

January 29, 2014

you need a thesis that will guide the rest of the structure.

Bring across your preference through a technical description.

I know, and have programmed extensively in, C, C++, Java, Python, and OCaml. Of these languages, I prefer OCaml for implementing programming language tools. OCaml has features that are useful for PL tools specifically and also for large software projects. OCaml is strongly and statically typed, uses type inference, is functional, compiled, has a useful foreign function interface (FFI), is memory-safe, and has a moderately sane packaging and build system

means not ends

Large projects benefit from strong and static type checking. When interfaces or types change, compilation and type-checking will report all the locations in the program that a change in typing requires refactoring. In dynamically typed languages like Python, discovering these changes would often require invocation of unit or systems tests and might not discover all locations that need to be refactored.

OCaml's functional style allows for natural and expressive implementations of tree traversals and other recursive applications that show up in PL tasks. Data types for programming languages and representations of programs can be concisely defined and pattern-matched over using the "match X with ..." construct. This allows for a natural expression of logic that analyses or interprets code structures like abstract syntax trees. Achieving this construct in C, C++, or Java can be cumbersome and involves using language features like instanceof or dynamic cast, or using tagged structures in C. "match X with ..." also benefits from static type checking as unhandled cases are revealed during typechecking, so as data structures evolve, the projects maintainers are always aware of which locations in their program need to be extended to accommodate changes to their data structures.

Start sent. w/ English

The OCaml community is not as large as the Java or Python community, so its library ecosystem is also not as large. However, OCaml has FFI that allows for OCaml code to interface with C or C++ code. Using this FFI, OCaml projects can be linked into existing C or C++ libraries, like LLVM and ptrace.

OCaml is memory-safe, so no program written in OCaml should, assuming a correct OCaml bytecode interpreter or compiler, and assuming correct external C libraries, OCaml programs will not have memory bounds errors that could be present in C or C++ programs. These errors are difficult to identify, diagnose and fix, so their absence is beneficial to

non-grammatical

Summarily rejected for exceeding page limit.

software projects as debugging efforts can be focused on ensuring the program is correct, instead of identifying low-level memory errors.

Supports your "natural" traversal argument too

OCaml also can compile to native code, and its native code compilation process produces optimized and efficient binaries. The OCaml native compiler can optimize tail-recursive functions into iterative loops. In benchmarks, OCaml performs favourably against Haskell (http://benchmarksgame.alioth.debian.org/u64q/ocaml.php).

In summary, OCaml has the memory safety advantages of Java, with stronger static typechecking, performance comparable to C and compatibility with C code, and a functional style well suited for PL tasks. Programs written in OCaml will be fast, memory-safe, concise and for programming languages tasks, easier to write and maintain than programs written in an imperative style.

From: [redacted]

Subject: On the usefuless of Java

Date: Thursday, January 30, 2014

Make a claim that establishes the structure of the rest of the doc.

Overall, Java is my favorite programming language. Although I wouldn't use it in every situation, it has many good features. One obvious one is its portability; a compiled Java program may be run on any platform which has the Java virtual machine. There are a lot of features which I feel make debugging a lot easier than other languages.

avoid: make definitive technical claims.

Lead w/ Java since this is about Java, not scripting langs.

In some scripting languages it is entirely up to the programmer to ensure types are used correctly, and using the wrong types can produce puzzling error messages. In contrast, Java has static typing which enforces type correctness at compile time. When combined with an IDE such as Eclipse, this means you can fix a lot of bugs before ever running the program.

When it comes to run-time errors, Java has Exceptions built in to the language. When Java throws an uncaught exception, it automatically includes a stack trace which shows not only where in the code the error occurred, but also every line of code in each calling function of the current stack. Often the exception message together with simple inspection give sufficient information to fix the problem.

weak opening sentence for TP.

Another feature is automatic garbage collection. This rules out an entire class of bugs, and again helps to simplify the debugging process. This is well worth any extra overhead from the garbage collector, and such overhead is continually being reduced by newer versions of the JVM.

really about speed, not size?

The strict, static typing system may make the code a little slower to write and more bloated than a scripting language. However, the object-oriented structure it imposes helps to write well designed classes and methods. Although the language can't enforce good design, it encourages things like modularity, encapsulation, and reusability. This makes it scalable to large projects, with large numbers of developers.

I can write ← public static int main(...) { 100K LO "Fortran" }

contradicts prev. sentence, no?

How does Java impose OO?

Java is a well-established language, and time-tested. The native libraries are very well documented and it is easy to document your own classes in the same manner using JavaDoc. Due to it's popularity, there are many high-quality tools and libraries available, which goes along with reusability.

CMSC631 PS1 [redacted]

Title

[redacted]

Make a claim that establishes the structure of the rest of the doc.

Problem 3.
My favorite programming language is Java. The reasons are as follows.

First, Java has a natural and easy-to-learn syntax. Java's syntax is mostly derived from C and C++. A C or C++ programmer need not learn many new rules about identifiers, key words, primitive types or control-flow structures when learning Java. As a purely object-oriented programming language, Java adopts a clear syntax to define attributes and methods only within definitions of classes - this makes Java code clear and easy to understand.

Java is not purely OO.

what does this mean?

Second, Java has a strict and highly extensible type system. Java is a strongly typed language. Java compilers can check type errors during the compiling stage instead of the executing stage. This characteristic makes Java more robust, or reliable. In addition, Java has a good system for programmers to define new types by extending the built-in "Object" type that defines some very fundamental behaviors of all non-primitive data types. In this way, all Java's built-in data types and user-defined data types can cooperate seamlessly.

passive

subjective

passive

I have no idea what this phrase means.

Third, Java has a clear and unambiguous pragmatics. Java has precisely defined syntax and semantics systems. Supporting inheritance and polymorphisms of data types makes it possible for a Java system to use dynamically bound run-time. This should be considered as an advantage of Java and well-designed Java systems will benefit from greater flexibility without bringing in ambiguous of pragmatics.

?

passive

non grammatical

not = library

Forth, Java has a powerful tool suite from both the language library and 3rd parties. Java has a set of powerful built-in libraries to support graphical user interface, distributed environment communication, XML-binding etc. Also, open-source organizations (such as Apache) and software vendors (such as mysql) provide a comprehensive set of 3rd-party libraries for Java.

awk.

Fifth, Java is well-suited for cross-platform, mobile, distributed, multithreaded and many other enterprise level applications. Java runs on an internal level called "Java Runtime Environment", and this makes Java platform-independent. A Java program on one machine can be transplanted to another machine with different hardware or operating systems as long as it has a compatible JRE. Java programs are also widely used in small-sized mobile systems as Java is originally designed. Java supports distributed systems as well as multithreading. In addition, due to Java's advantages in OO structure, security, multi-threading, Java is also widely used in building enterprise level information systems.

compared to?

what is small sized?

Bad title

Program Analysis and Understanding - Homework 1

[redacted]

January 30, 2014

Make a claim that establishes the structure of the rest of the doc.

Here are a few reasons why I like Java - Java is portable i.e. it supports the "Write once, run anywhere" paradigm. This is because it compiles the high level code to an intermediate byte code ~~that can then be~~ Passive interpreted by the Java virtual machine(JVM). It is a statically typed language which strongly encourages use of the object-oriented paradigm. Java has a rich library providing large functionality including GUI programming, multi-threading, networking, dynamic class loading, XML manipulation, etc. The presence of such a large API allows for Passive creating large softwares with many functionalities in a ~~modularized~~ manner with few implementation dependencies. Java has a well maintained documentation *Javadoc*. Java ~~supports~~ uses automatic garbage collection and thus avoids memory leaks (in contrast to C++). Java is reflective, and hence allows metaprogramming and dynamic code generation at runtime.

avoid jargon in tech writing

non grammatical

Sentence seems to say "static typing encourages OO"

GC does not guarantee lack of memory leaks, but it has other technical benefits.

Nothing more to say?

Summarily rejected for being over page limit

Bad title

Problem Set 1 : Memo

[redacted]

Bring across your preference through technical description.

Haskell is a powerful purely functional language with a strong static type system, and lazy evaluation. I am fond of Haskell because of the elegance of its constructs, and the power that it can derive from a very simple set of starting assumptions. Haskell is built to look and work like mathematics, and to that end its syntax was designed to mimic mathematical constructs, and formal proofs.

← what does that mean? (a manager may not know)

mathematical formulas?

Don't see connection between data processing & functional.

Because of its functional nature Haskell is suited for many tasks which involve the analysis and processing of data, and because it's so syntactically similar to raw maths people can convert complex transformations into code with minimal effort. This is true for fields like bioinformatics, and more general scientific computation. Systems like compilers and parsers also become much more elegant in a purely functional context, and much of the management work that would be needed in an imperative language is swept away. Dynamic programming problems can see significant speedups with Haskell because laziness means that many unnecessary paths are never evaluated.

what management work?

A few code samples will help highlight why I like the language.

```
-- The Simple Canonical Implementation

fib  :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

This is the naive implementation of the Fibonacci function, in Haskell, and could have been copied character for character from a blackboard in a maths class. It's small but the use of pattern matching allows one to easily define the base cases of the function and then expand from there. More interesting is the infinite list version of Fibonacci.

```
-- Inifnite List Version
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
fib n = fibs!!n
```

← looks less like math to me...

This example wouldn't work without the lazy evaluation of values, the `fibs` value returns an infinite list of numbers, which are all only evaluated when you ask for them. This automatically memoizes the results of each intermediate stage, because the `fibs` list is unrolled in memory into a list of numbers, with only a thunk at the first unevaluated element in the list.

? (jargon)

Haskell has a strong typing system with all the guarantees that implies, and because it's easy to create new types, you can start encoding new constraints into the type system, and now those constraints are checked at compile time and are impossible to break at runtime. Because Haskell also has HM type inference, the strong static typing can be achieved without constant type annotations, making writing code a smoother experience.

Haskell's community has it well supplied with libraries and tools for all sorts of tasks. It's particularly good for compilers because of the strong library support and the syntax of thelanguage itself being well suited to the task.

Benefits of Racket ← Be descriptive

[redacted]

University of Maryland

Bring your preference across through technical description.

Racket is a functional programming language in the Lisp/Scheme family and the first programming language I ever learned. After having learned several other languages (including Java, Python, and C#) it has remained my favorite language because of the well-designed IDE, the way it can easily be extended and developed into a more customized language, and the way it is designed to be easy to pick up for a beginning programmer.

It's "DrRacket"

Doctor Racket, the developing environment for Racket, is easy to set up and easy to use, particularly for beginning programmers who might be intimidated by more complex environments such as Eclipse or Microsoft Visual Studio. Dr. Racket allows you to easily navigate your code by giving you a menu of all your definitions so far and allowing you to jump to them simply by selecting the name. In addition, the IDE provides a number of debugging features (such as the Debug button and the Macro Stepper) which are extremely helpful for locating and correcting errors in the code.

subjective: make the point with factual claims

Not sure what that means

Dr. Racket also comes equipped with some limited versions of Racket which can be used as stepping stones to learn how to program and these can be accessed simply by using the "Choose Language" menu. These many variants of Racket as well as the option to specify which language you are using in the actual code (via the #lang tag at the top of the definitions window) show that Racket is a very customizable language.

The Racket language allows a beginning programmer to understand key concepts such as recursion more easily because of its readable syntax and its lack of side effects. In addition the error messages provided by Racket provide users with a lot of helpful feedback about what type of mistake they may have made, which is crucial when one is just beginning to program, as a good error message tells a programmer not only where to look for a bug but also how they might be able to fix it.

1 sentence!

Racket is particularly useful when creating small games because of its list manipulation techniques which allow a user to keep track of and edit large amounts of data. The language is also useful when performing mathematical computations because its syntax so closely resembles the syntax mathematicians use when writing out computations by hand.

Really!?

In conclusion, Racket is a functional programming language with an intuitive syntax and an IDE that was designed to be easy to use. The language makes it easy for beginners to learn how to program, and is particularly useful for mathematics and for the creation of games, which helps beginning programmers to become interested in computer science and to understand key concepts such as recursion and data abstraction.

1 sent.

Racket - Flexibility from Minimalism

[redacted]

Make a claim that will establish the structure for the remainder of the doc.

Racket is my favorite programming language for several reasons besides common features in functional languages such as first class functions and pattern matching.

avoid in technical writing

First, the syntax is simple and consistent. Beginning programmers spend little time learning Racket's syntax and more on programming itself with language-independent topics such as abstraction, contracts, recursion, etc. Experienced programmers read Racket code with ease, as the first symbol in a form indicates that form's meaning. The unambiguous nature of the syntax makes Racket easy for humans to read just as it is for machines to parse.

subjective

Second, meta-programming is pleasant in Racket thanks to the uniform syntax. Because Racket macros deal with abstract syntax trees instead of raw strings, manipulation is less error-prone. Macros come in handy when programmers want to abstract away second-class language features, which helps reducing code duplication further and increasing productivity. For example, we can use macros to abstract away evaluation strategy, implement a type system, or create a new language with a different paradigm on top of Racket. In other languages, the same level of abstraction achievable from macros is not possible, or at least not as easily and reliably.

further than what?

Don't know what this means.

subjective

Third, it is easy to create new languages on top of Racket that interact with one another. In fact, there are already many of these in a standard Racket distribution, and a program can be composed of different modules written in different languages. Typed Racket implements a sophisticated type system allowing gradual program annotation for static checking without precluding common idioms in dynamic languages. Lazy Racket allows controlled laziness in selected components, avoiding unpredictable overheads in the entire program. Redex is a highly helpful domain specific language for modelling and testing language semantics. The environment makes it fun to experiment new language features, and cutting edge research results do not have to wait for years to be adopted.

Last but not least, Racket comes with well documented libraries and tools. The IDE is uncluttered yet powerful and provides tremendous help for debugging. The tool suite makes Racket appropriate for practical programming ranging from throw-away scripts to large software systems.

In conclusion, Racket ~~is great because it~~ liberates programmers from accidental complexities and gives them freedom to create.

Don't all languages?

make a descriptive title about C++

C++ [redacted] favorite Programming Language

January 29, 2014

My favorite PL is C++, which ranks the 4th by TIOBE index.

C++ has a flexible and concise syntax. With the introduction of C++0x, it now supports auto keyword, which means it can automatically detect the type for a variable when initially declared. This makes the syntax more concise than before, e.g., "map<int>::iterator iter = m.begin()" can be simply written as "auto iter = m.begin()". The syntax of C++ is also highly flexible. For example, C++ now supports Lambda functions and expressions. In addition, another experience that I recently had is that there is no direct way to create an array with generic types. However, C++ support it directly without applying any tricks.

swap order to be parallel

mismatch in number

C++ has (almost) the best performance. It is one of the philosophy of C++ that C++ ~~should be able to~~ compile to assembly language directly. This key point makes C++ very efficient, as long as the compiler is good. all code in Ruby is actually first compiled to C++ and then compiled to executable files, which means longer compilation time and more difficulties when doing compiler level optimization.

However, ~~there are some aspects that I do not like~~. Apart from STL and boost library, there are not many libraries and can be used and installed in a clean and easy manner. The package community of C++ is much smaller than the one for Python or Java. In addition, C++ is becoming more and more complex, especially after C++0x, and there are so many subtle techniques that need to be remembered if high efficiency is needed.

Why? Seems like you get all of the C++ opt. PLUS high level opt. at Ruby level.

Bring across preference with technical description

Confused: you had this experience w/ C++ or something else?

make a descriptive title about Ruby

Why I Use Ruby

CMSC 631 Spring 2014

[redacted]

~~Abstract~~

Every great craftsman boasts a wide array of tools from the general and multipurpose to the obscure and powerful, but ultimately leans toward one of choice. A computer scientist crafts code from languages of choice, begging the question: why use a certain language above all others? From experiences with Java, JavaScript, Python, Livecode, and other languages, I find Ruby the most convenient. Ruby is a dynamically-typed object-oriented programming language whose flexible syntax provides a versatile tool to model various concepts.

Incorrect usage— Look up "begging the question"

what does "flexible syntax" mean?

Bring preference across w/ tech. description.

1. Dynamic Type System

Ruby is not a type enforced language, thus allowing for polymorphic variables to capture type-changing functions, such as modeling limits or methods with nil-case returns. Loosely typed variables also serve as reliable sentinels with no overlap with functional values used. This ultimately allows for variables to carry two nontrivial properties—its current type and its value—allowing for more compact ~~coding~~ code.

what?

"coding" is an activity.

2. Object Oriented Model

Values in Ruby are represented as objects in system memory tied to their own states or environments. Variables store pointers, making the multithreaded access of a single value more convenient than in functional languages. Ruby, however, also handles pointers to functions, allowing higher order functions and quick replication of functional languages' powerful value definitions and recursive strengths.

what is a rec. strength?

subjective?

Also true in Schemme.

3. Metaprogramming Eigenclasses

Ruby separates compiled source code and runtime code, allowing dynamic modification of methods, types, and objects allowing for black-box modification and a plethora of domain specific tools. For example, values retain additional nontrivial information through the instance fields of both their type and their eigenclass. This information ~~storage~~ is nested through the chain of eigenclasses, providing more powerful variables that carry vast amounts of useful information in a scalable manner.

subjective

than what?

how, in what way?

Conclusion

What Ruby lacks compared to static environment that efficiently enforce type contracts, correctness checks, and the like, Ruby makes up for with its flexible and powerful process model. With interchangeable and lightweight syntax and excellent documentation on top of efficient modification and modeling capabilities, Ruby is a very powerful and convenient language for goals and users of all calibers.

sub.

has a specific meaning that doesn't fit here.

Avoid obscurity, inflated vocabulary, & convoluted construction.

[redacted]

these are (mostly) means, not ends

A very commonly used programming language for introductory computer science courses is C++ as it is a high-level language, which is easy to read and understand. It is an excellent programming language because it is object-oriented, has templates, is widely supported, has a static type system, and many of its important aspects are transparent to the programmer.

be more specific

C++ is an object-oriented programming language. It is an improvement upon the C programming language and maintains much of its syntax, with the addition of classes amongst other things. Object-oriented languages allow programs to be broken down into their most fundamental elements. Programs then become more modular, allowing for portions of them to be easily reused for different purposes. C++ allows users to define their own data types as classes. Classes in C++ allow for inheritance and multiple inheritance. Inheritance allows classes to build on each other. There is a base class and a derived class, which inherits from the base class. Multiple inheritance allows a class to inherit from multiple base classes, a feature not supported by other programming languages. C++ supports templates, which allow for programs to be implemented in a more generic way.

most

than what?

Developed and launched in the early 1980s, C++ is a very widely used programming language, and because of this there are a wide variety of tools available for use with C++. There are many software libraries ~~that were~~ built exclusively for use with the C++ programming language. One of the most popular libraries ~~that programmers use when writing in C++~~ is the Standard Template Library (STL). The STL provides a great deal of functionality through algorithms as well as containers. The STL contains many commonly used algorithms in computer science, including several different sorting techniques. In addition to the many algorithms offered by the STL, there is also a large set of containers. STL containers are commonly used objects such as a stack, queue, list, and many other objects, which make data management much easier. Since the introduction of the C++11 standard the STL has been expanded to include auto pointers, lambda functions, signals, regular expressions, and atomic operations just to name a few.

than what?

} means. Why do these matter?

C++ is one of the most supported languages currently on the market. Because of this, there is a significant set of supporting libraries that allow for the easy extension of program functionality. Some of the most notable libraries are OpenCV, Cuda, MPI, and Eigen.

Not sure what you mean.

With a static type system, type checking is performed at compile time for C++ programs. This helps the programmer to more easily identify bugs due to type mismatching. One of the biggest reasons that C++ is used in introductory computer science courses is due to its transparency. It is a high-level programming language, but it leaves much of the lower level program management up to the programmer. The programmer is responsible for most of the memory management. Variables can be declared dynamically using the "new" operator, in which case the programmer is responsible for both the allocation of the variable's memory and the garbage collection associated with that variable. Class member variables have to be declared by the programmer as public, protected, or private, and the type of inheritance also has to be specified by the programmer.

} means

C++ is a suitable language for any large, high-performance project as there is a great deal of community support and the language has been around for a long time allowing the compilers to be extremely optimized over time.

Over allowance on allow. Try a rewrite w/ ZERO "allows".

make a descriptive title

Haskell ~~is my favorite programming language~~

Include a "thesis" that guides the structure of the document

spell check!

Haskell has an expressive syntax Haskell comes with expressive syntaxs to achieve pattern matching and list comprehension, [1] which saves many lines of code but preserves the readability.

```
-- pattern matching: Fibonacci numbers
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- list comprehension: all Pythagorean triples ∈ Z^3_{>0}
[(a,b,c) | a <- [1..10] , b <- [1..10] , c <- [1..10]
         , a <= b , b <= c, a * a + b * b == c * c]
-- result: [(3,4,5),(6,8,10)]
```

Listing 1: Example of pattern matching and list comprehension

Haskell has a strong and static type system Haskell's type system is strong. In general the type of an operator and its operands are required to be exactly matched and there is no way of forcing type conversion in Haskell, thus the compiler will complain if you are doing something improper (e.g. trying to use integers as strings). Moreover, Haskell is equipped with static typing, and therefore is capable of type inference. Suppose you are writing a function `f` to sum up a list of integers, you can simply write `f = foldl' (+) 0`, which is enough for Haskell to work out `f`'s signature: `[Integer] -> Integer`. One can in addition gain some intuition about what a function is doing by looking at its name and type signature. Take `filter :: (a -> Bool) -> [a] -> [a]` as an example, we can tell from the type signature that a predicate and a list is requied, and we will get a list of the same type in return. By looking at its name, we can further tell that the function should work like a "filter" using the predicate given as a guide.

This implication is false. Not all static type systems can be inferred.

be specific

Haskell enables explicit control of side effects In many other languages, side effects can be created ~~everywhere~~ anywhere by doing I/O or mutating global variables, which make some procedures error-prone and hard to reason. By introducing the concept of monads, Haskell models side effects using its type system so that a value are tied to the side effect it causes. You can, for example, get user input using `getLine`, and in return you get a value of `IO String`, and there is no way for this string to live without `IO`. Therefore we have controls over side effects and functions are guaranteed to produce the same result as long as they are called with exactly the same arguments, which is good for unit testing and debugging.

Disagree in number.

What is it?

My Chosen Programming Language

[redacted]

To give you some context, I consider myself well-versed in C, Java, Python, and Matlab. I've also done a moderate amount of web programming (HTML, CSS, JavaScript, PHP, SQL) and I've written the occasional Linux shell script. Lastly, I've had some limited exposure to assembly language, C++, Scheme, Ruby, OCaml, and Prolog, mainly through course-work or teaching-related activity (the last three while TA-ing for CMSC 330).

Choosing one of these languages as "best" is very difficult, because I think that the best language for a task really depends on the task. For example, I would prefer a scripting language for some quick and dirty text processing, but a low-level language for a performance-sensitive application. That said, if I had to choose, it would come down to the following two languages. Choose one and give a technical description that demonstrates its qualities

non-tech.

First, C is the closest to my heart and probably my favorite. This is partly because it was the first language I used seriously and I have the most experience with it. But from an objective standpoint, the primary advantage of C is control. Of all the languages I listed above (other than assembly), C gives me the most direct control over the CPU and memory of my computer, primarily due to the explicit memory management. This is clearly important for optimizing performance, but also because building a program from the ground up gives me a better understanding of the program.

This could be a good "thesis"

On the other hand, the major drawback of a low-level language like C (not to mention assembly) is the increased difficulty in writing (and later, reading) the code. This is particularly problematic in a research setting where I want to try many different ideas on the fly. These days I use Python much more often, which is my second language of choice. I find Python to be much more readable and writable without prohibitive hits to performance. Some features that I find especially beneficial from the read/write-ability standpoint are dynamic typing, object-orientation, garbage collection, list comprehension, and versatile built-in data structures (e.g. dicts). The n-dimensional array object in the NumPy library is also extremely useful for any intensive data processing.

One more notable aspect of Python is the increased similarity to natural language: indentation instead of curly braces, "or" instead of "||", and so on. The resulting aesthetics of Python code can reinforce the sense of accomplishment felt by the programmer once their code is complete. Not everyone finds Python beautiful, for aesthetics are inherently subjective - but then again, computer programming is an art as well as a science.

Why Anaconda Python?

[redacted]

Why should our development team use Anaconda Python as a core part of our software stack? Anaconda[1] is a free, cross-platform distribution of Python that comes with a package manager, a plethora of best-of-breed libraries, and an integrated C compiler. The integrated C compiler allows the use of the optionally typed language Cython[2], which translates a superset of Python into a C extension module.

Python has gained increasing popularity as a glue language. While its dynamic dispatch and runtime introspection capabilities provide an excellent environment for rapid iteration and debugging, these capabilities come with an implementation that doesn't allow for maximizing memory or processor usage. As a result, Python is often used for those parts of an application that are not performance-critical. Cython allows the incremental optimization of Python-based code via type annotations, providing both the rapid iteration of Python and the potential for optimization via C where necessary.

On Windows and OSX platforms, a C compiler is often not a default part of the OS install. Anaconda is a simple way to install a significant portion of the needed development environment with one installer for those platforms. The Python standard library, augmented with Anaconda's built-in QT integration and scientific and numeric libraries, provides an excellent cross-platform baseline for ongoing development.

Anaconda is a pragmatic programming environment, allowing quick access to a wide variety of advanced algorithmic tools. Its focus on addressing deployment issues will allow more of our development team's time to be spent addressing the problem we are trying to solve, rather than the problem of getting our development systems into the right configuration.

---

[1] https://store.continuum.io/cshop/anaconda/

[2] http://cython.org/

Bad title Give a descriptive title about Python.

CMSC 631
Program Analysis and Understanding
Problem Set 1

[redacted]

January 30, 2014

Bring across your preference by a technical description.

As I move from using the computer as a tool to solve problems to using it as a framework for implementing ideas, my chosen language has been moving from perl to python. Python has also proven to be better than perl for working with a group of programmers on larger scale problems. Python serves both as a scripting language to perform small local tasks and as a language to develop larger systems with its object oriented paradigm.

Perl is well-suited for applications that manipulate text data. Perl's syntax uses punctuation symbology to make regular expressions easy to write. Although regular expressions are not given the priority they are given in perl, python's clean syntax makes it easy to convert concepts into code and vice-versa. So, whereas it is easier in perl to implement a program that performs a natural language processing task, it is easier in python to implement a program that *learns* to perform that same NLP task.

It is easier to read statements in python than in perl. Since it is strongly typed, there are not many ways to write a statement with the same meaning, and thus the program reader is not easily surprised or confused by new code. Perl, on the other hand, not being strongly typed, allows the programmer to express his own personal preferences in code. Usually these personal preferences do not coincide with the preferences of another programmer reading the code. For this reason python is a better language for use in a collaborative team effort.

why does the rest of the sentence follow from this?

Perl v Python: pick one and argue its technical merits.

Ruby: Fluent, Flexible, Extensible

[redacted]

Ruby is one of several popular scripting languages. Like others, Ruby has strong text-processing capabilities, an internal documentation system, and an active user community that has produced many extensions. It can knit together parts of a larger system or accomplish small tasks quickly. The key reasons to use Ruby in many environments, though, are its blend of object orientation and functional idioms, and its extensibility, arising from its flexible syntax and reflection API.

what does that mean?

Structure should be
1. Fluent
2. Flexible
3. Extensible

Object orientation and functional idioms

Ruby was designed with deep object orientation—everything is (or acts like) an object—and with functional influence, through the use of code blocks, symbols that can be interpreted as method names, and `Proc` and `Lambda` objects. Ruby programs can naturally take advantage of both paradigms. For instance, the `reduce` method implements the kind of list-folding capability that is common in functional languages. It is implemented in a module that can be easily mixed into new classes: an object that represents an enumerable sequence only has to implement an `each` method to obtain `reduce` and several other commonly available methods on sequences.

The `irb` shell is a read-eval-print loop for Ruby, allowing developers to incrementally implement and test ideas as they can in languages such as Clojure.

this feels out of place

Flexibility and extensibility

Ruby's syntax allows many choices. For instance, method calls can use or omit parentheses, making some methods seem like keywords, and code blocks can be delimited with braces for small stretches or `do...end` when they are larger. Method names are commonly aliased, so that the developer can use the name that is most natural in a particular situation. Operator overloading—including the availability of type-coercion methods that allow objects of different types to appear in either order—also make newly written classes seem like part of the language.

Of course, the flexibility of the language makes it important to have common agreement about style, both within a development shop, and in the wider community. With that understanding, Ruby lends itself to write-once-read-many programming.

Ruby also has a reflection API that allows programs to modify method dispatch, examine and adjust variable bindings, define and undefine methods at runtime, and hook into the object system. These allow developers to use Ruby for metaprogamming; indeed, commonly used parts of the language, such as `attr_reader`, are defined using this framework. These capabilities account for Ruby's popularity for writing domain-specific languages.

Perhaps because language extensions feel so natural, there are many of them available, making it easy to take advantage of the community's prior work.

Well written but the structure could be improved to support argument.

HW1 Descriptive title [redacted]

Bring across your preference by technical description:

~~C++ is my favorite language~~. Here I provide some reasons in support of C++:

write a thesis that guides the structure of the memo.

1 Cpp has a strong type system:

C++ has a static type checking system. Therefore the compiler reports most the type error problems at compile time and you don't need to worry about runtime problems. However, unlike Java, many of the variables can have different meanings when ~~we are using them~~ used in different contexts. For example, an integer variable, can be treated as a boolean variable when ~~we are using it~~ used in true false expressions. Moreover, we can add an integer variable with a boolean variable and still get an appropriate result.

of?

2 Cpp is well-suited for fast and algorithmic applications:

C++ is ~~a~~ well suited ~~language, when we intend~~ to implement algorithms in batch problems. It has a great library, supporting almost all the data structures and standard algorithms. Moreover, C++ is faster than most of the programming languages, hence, many of the application that need to be time efficient are written in C++ ~~language.~~ Furthermore, C++ uses less memory comparing to languages like Java and Python, and this is the other reason that many companies use C++.

sub-

ed

or another?

3 Cpp has a good syntax:

The syntax of C++ is used widely in other programming languages. This shows that it is very popular and easy to learn. It is very powerful since you can define functions ~~with~~ accepting ~~infinitely many~~ number of arguments and overload the operators. Also C++0x provides a new way of defining such functions with templates. If you are an expert in using C++, you can write java programs with fewer number of lines in C++.

an arbitrary

4 Cpp is a good programming language for small programs:?

Sometimes you just need to write a 10-15 line program to do an easy and small task for you. C++ is also a good choice for such programs. You don't need to implement classes, functions, or other sophisticated programming tools; you just write a main and do what you want in it. Programming languages such as Python or Perl are also good choices for such applications.

what?

deflates your argument.