CS 131 (Eggert) Notes

W 1 M Lec 1-10-17

- Microsoft interview was this exact same question!
- This is one of the most famous problems in Computer Science
- NOT the halting problem or P = NP, but rather a programming language problem
- Solved by Donald Knuth, a Turing Award winner
- Knuth was a Professor at Stanford, formerly at CalTech.
- At CalTech, he decided to write a textbook to write about everything important about computer programming
- 1962 and the field was still young!
- CS 31 and 32 of the day and the last chapter would be about programming languages
- Knuth is a world-class computer science researcher and he is OCD about writing the best possible book

- NOT embarrassed to jump into calculus to explain why it runs efficiently.
- Published by Addison-Wesley in 1969 (first class academic publisher)
- The plates were all created by hand
- Knuth wasn't happy!
- The book looked okay but he thought we should be doing something better.
- Wrote a tool to help him generate the book.
- When a job is NOT done well, you write a program to automate the repeated part.
- TEX - source code that the TEX processor reads and generates something that looks nice on the screen.
- Knuth found images for a book that were better than the Addison-Wesley ones did by hand

Fundamental Algorithms is now done with TEX!

- The TEX output is so good that now, CS professionals use TEX!
- Word just isn't good at this.
- Knuth being a software engineer as well as an academic wanted to write this TEX program so that he could do his own book but other people could use the program as well.

Java now is the equivalent of Pascal the 1970s

- Used by a lot of practical application developers
- #1 language of **Apple** when **Apple** started off.

A lot of beautiful algorithms and he was worried that if he just shipped it out, people would never read it.

- He wanted to write a book about the program that explains it, and explains it well.

- You should write your book in TEX!

If you have a program and the documentation, they will diverge

- Knuth looked at the problem, and we wanted to come up with a different way of writing code.
- Attack the fact that documentation and code have a mismatch

Actual combination he used had the code extracted and then sorted

- Pascal insists on declaring things before you used them
- The book order was different from the code order, and he had a tool to shuffle the code around to make it good.
- We have now proved that this idea works to my own satisfaction, but if you are the academic, that is NOT good enough!
- You want to not only write code, but you realize most CS people are busy!
- They don't have time to read a 400 page book!
- You want a short blurb to convince other computer scientists to read the book
- Literate programming - you write a program as if you write a book!
- The idea is that you use this everyday for the world's most popular programming languages
- Knuth publicized this!
- He decided to write a short paper to illustrate this idea.
- Do a small problem and show how literate programming would work.
- The small problem was the Quiz question!

The story behind the demo is a little tricky!

- Knuth approached the editor of CACM (the world's leading organization for CS)
- Use a literate programming style, and they would output the literate program that would be extracted through LaTEX
- Big innovation and most people relied on the magazine to do that time of type-setting.
- Invented a new data structure called a Hash Trie!
- Like trees but not quite.
- Designed exactly for this problem.
- Feed a word into the Hash Trie and it will immediately find out if it is in a Hash Trie
- Update a counter sitting in the Hash Trie.
- Ph.D CS professor at Stanford.
- Wrote it using his Literate Programming technique so the actual code was in Pascal

McIlroy said there was an easier way to solve this problem

- Burned into Eggert's brain and McIlroy had a different solution

- Linux was invented at Bell Labs originally to do text processing and McIlroy was a competitor.
- Different kind of software technology that could process text better than Knuth
- McIlroy was Ritchie's (creator of C) boss
- McIlroy insisted on using the | (pipe) operator
- Why don't we use my contribution to Unix instead?

Transliterate nonalphabetic characters and we would have pulled the words out of the input.

- The simplest way of getting closer to this answer is to use McIlroy's pipe (|) to sort these

McIlroy created a one line Shell program that did the same thing as Knuth!

- The fact that these options are here is NOT an accident!
- They needed to write Shell scripts like this all the time to make this kind of application easier to do.
- Even so, there is a valuable point here that at this level of abstraction, this problem is trivial
- Something you can give to beginning undergraduates
- HW 2 in CS 35L but slightly revamped!
- Illustration of choosing an appropriate level of abstraction as long as you don't care about certain things.
- The downside of this approach is that it is way slower than Knuth's hash trie!

uniq only finds adjacent duplicates because it has a bounded amount of memory

- This is like a general purpose sort command that has all sorts of funky stuff in it.
- Sort a TB worth of data even if you have a small HD
- Heavy duty machinery under these simple machinery
- Knuth's thing was stripped down and we had to pick Knuth's solution
- A lot of things to like about the Pascal version
- This is naturally parallelizable
- Get parallelizablity for free
- Knuth is more sequential
- The way the Linux kernel will do this is that it will start running the main code, but they are interlocked.
- When sort does a read, it waits for the regex to start writing
- Each one of them is a program operating in parallel in its own mind.
- Do we really have good parallelism here?
- Maybe that's a win!
- If you are running a single CPU machine, it will pretend to be parallel
- If a machine has 4 cores (like our laptop), it can run in parallel!

- They can be running at exactly the same time but there is something bogus about this claim
- sort has to read all its input before it can output anything!
- sort can start reading before tr finished, but it cannot read until tr does its last write
- NOT enough parallelism to really help!
- All 10 GB cannot flow into sort and they start shipping and we are running sequential programs.

If you are writing a program and you want to use it right away, this is the way to go.

- Don't want to farm off something to a Ph.D holder.
- This programming is the opposite of what we taught you in CS 31 and 32.
- Hacking together a script by someone who doesn't even know that sort is O(n log n)
- Runs and runs well enough.
- You can do this programming without going to university.
- We have a conflict between regular Computer Science (disciplined and organized - Knuth) vs the hack together Computer Science (Python and McIlroy)
- Conflict that will run through this class and some other classes that you take.
- Do things the classic CS way via OCaml, which is in the well-engineered, a lots of compile time checking camp
- Later on, we will do more loosey-goosey scripting languages!

Was Microsoft looking for a really good C++ hacker or were they looking for someone who could think outside of the box?

- Both answers are right, and Eggert wasn't sure which question was Microsoft looking for.
- Figure out what rule the Professor wants!

C# is a wannabe Java
JavaScript - Python
Perl - Python
PHP - don't suffer through this!

Pascal - no longer on the list!

- Still people in Murphy Hall who write COBOL
- COBOL had much more reach
- Prolog is bizarre, so naturally we have to cover it in this course
- This is to show you how a reasonably large chunk of the world works that isn't a programming language world.
- OCaml

Intel engineer - why are you wasting people's time in Scheme?

- They use ML for chip verification
- LISP is so 1960s!

- Very persuasive guy, so Eggert started using ML
- ML is used at financial institutions
- Practical applications for ML
- Academic world vs practical world
- There are other favorites like LISP but we hate Perl and Visual Basic (VB)
- Prejudices of most institutions
- See the world in a nicer way by studying Python than Perl
- Eggert has talked to both designers and the Perl guy is a linguist, not a CS guy.
- A university should provide general principles underneath!

Software issues

- How do you write programs and change them efficiently from doing something to doing something else.
- Build a model in your head about how a language works
- What models are out there, how do they work, and how do they NOT work (where do they screw up?)
- If you read programming languages in the textbook, you see how great it is.
- Apple says that Swift is the greatest language you ever saw!
- Eggert gets paid to tell you about limitations.
- Write down ideas so they appear as code
- If you cannot write it down, it doesn't exist, so coming up with a good notation is key!
- How do we design these programming languages and have good support for these notations?
- Ancient Greeks couldn't use algebra because they didn't have notation, so it really hampered them.
- We want notations that explained the coding we want to do and the coding we do in a succinct way.
- This kind of evaluation happens all the time in the real world when choosing a programming language for a project.
- You have to know how those strengths and weaknesses work in order to make intelligent decisions.
- C++ is the world's worst programming language for many problems!

- With all this in mind, Eggert must confess that he is NOT a big picture guy.
- There is a famous story about hedgehogs vs foxes
- Hedgehogs want a single, unifying principle, while foxes just run around and keep searching for something cool.
- This is Eggert's attempt at being a hedgehog where he is going over bright and shiny objects that look cool.

- **Keep this info in the back of your mind!**

- Language is what you use to talk to people and it is remarkably flexible
- C++ is NOT a language! You cannot write a love poem in C++
- Rename this course as **Programming Notations** because that is what it really is.
- When you write a Python program, you are using the same sorts of ideas that mathematicians use when they write calculus.
- Languages is just salesmanship, but we are bloating the importance of this course beyond what it really is.

Prereqs:

- Assume basic issues involving software instructions - CS 35L: sh, make, git
- Frontload the assignments where the earlier assignments take up more work while later ones are a bit less.

Exams:
Anything Eggert says can be on exams

- HW assignments could be on exams because Eggert may ask them on the exam and how to improve upon them
- Practice exam is given

Projects are solo

- Please don't use solutions on the Internet and they do look for copying and please don't make Eggert's life more miserable than it already is
- Don't waste time!
- Do your own stuff and it is okay to talk bout general ideas and debugging strategies.
- All the work you submit has to be your own work.
- If you find yourself emails of code snippets, you have crossed the line.
- Gray areas and improving an assignment that someone else did.
- If there is any doubt whether you can use something, ask the TAs or Eggert some other time.

syntax - deep form of the language
semantics - what the language actually means

First two programming languages where you cannot do assignment statements whatsoever.

- No ++ or changing anything, but you can still get work done with functions

names - they are important because the way we name objects is crucial

types - description of values used

- Deciphering the type message from the OCaml compiler

control - who gets the program counter next or the instruction pointer

concurrency - the central unsolved problem of programming languages

- Fast, reliable, and easy to use

TAs are in charge of the class and you learn by doing

- You have to look at that and we need all this other stuff to hang together.

Other faculty members ask Eggert about programming languages

- Once you have a Swiss Army knife, it is hard to specialize.
- A recurring dream of PL is a way of creating broad-spectrum languages.
- It would be great to have one language, but if you build a truly broad spectrum language, the complexity of that language is so great that people will go nuts.

- Tradeoff between speed and exception handling

W 1 W Lec 1-12-17

Suppose you are in a design meeting, how would you decide which language to use?

- If everyone knows one language, inertia builds up and it explains why we are still using Java (1994) and C (1971)
- Similar to a language you have already used
- A simpler language is easier to implement
- C programs run faster than Java and Swift, and performance is important
- Memory usage
- CPU

Scalability

- How well does the language scale to very large programs?
- How well does it scale to users?

Convenience

- How easy is it to do things?

- Some of these qualities collide with each other!
- Convenience and simplicity compete with each other!

Orthogonality

- Independent axes that do not constrain values of the other axes
- People can choose one feature without worrying about other features
- A place where a language should be orthogonal but it is not

- Cannot return function in C, but you can return a pointer to a function

C says arrays are 2nd class objects but all you get is a pointer to a 0th element

- They don't want you returning arrays because it will be confusing even though arrays and pointers are two things!
- Suppose you have a type t that is implemented by a header
- You want to return a value of t as a function
- Orthogonality doesn't immediately jump out at people but we really will miss it.

Never use array types because of its lack of modularity.

- We don't tell you this in 31 and 32 because we don't want to scare you.
- Can you return an array if you make it a pointer
- Put it in a struct since there is a no restriction
- The problem with pointers is that people think they are copying object values, but they are actually copying pointer values.

Programs are safe when you know that your program won't fail

- Safety is a subset of reliability but it is an important subset!
- You want it to be safe as well as reliable

Don't use Python for multithreading!

- Use C!

Mutability

- Successful languages like C++ evolve!
- What we told you in CS 31 and 32 about C++ could be obsolete!
- They are rolling out a new version of C++
- Subset of this year's version of the language.
- See evolution in hindsight.
- Eggert learned a programming language back in the 1970s that had the following properties:
- ran on PDP-11
- memory access is considerably faster than the CPU
- The programming language was designed with this limitation in mind
- Took more symbols to add numbers than memory accessing!

Memory cycle time now - what is the ratio now?

- CPU is much faster than memory now (100 to 1 is NOT uncommon)
- It can take 100 CPUs to load something from RAM or you could have added a number in each cycle
- A programming language designed for this way of computing where memory is fast and CPU is slow has evolved with time.
- Use this language where these performance characteristics are wildly different.
- The language we call C now is radically different from 1975!
- Language's evolution has put programmers in a tough state.
- Your reflexes are designed for this type of machine and this leads to a lot of performance problems.

- With student code, it doesn't matter, but in production code, this is going to matter!

Wrote a program in C that could take varying # of arguments

- Arrays decay into a pointer at the 0th element
- The code Eggert was dealing with this had this all over the place and you called them this way because that was the convention in this piece of software.
- All Eggert tried to do is make Foo(x, y, z, w, u, v)

- Foo can call something with 3 arguments
- You cannot do exactly this but I can write something as close to this as I can.
- Use macros!
- Frowned upon in CS 31 and 32
- Take advantage and figure out there are 7 guys there.
- Particular use of C being extended to use this particular calling convention
- Extended by using macros (which is highly dangerous!)

Steve Bourne (creator of Shell): Hated parentheses and solved the problem using macros

Two kinds of mutability

- Helps you tell if language succeeds

Syntax - first programming language is talking about syntax

- Formal way of telling whether a program is valid
- If we define grammar, we define syntax so it is circular reason
- **Draw the line between form and meaning**
- We don't care what it means, we just care what it looks like.
- In this lecture, we are focusing on **form**
- Talk about human civilization and talking about what clothes people wear
- NOT talking about the important parts, but rather the surface aspects
- Save meaning for later! **(hard part)**

Computer Science until recently was not considered a dignified profession

- People considered CS as a hacker profession
- In Japan, software hackers (t) don't have the same prestige here
- If you are a good engineer, you want to work for Sony but you don't want to write software because that isn't very dignified.
- **Computer people convinced other professors that CS was for real is with syntax**
- In the 1960s, we showed you can describe the syntax for programming language in the same way as natural languages
- There were things you can do more efficiently

- The rest of the university were impressed and thought there really was a field of Computer Science that was important.
- Badge of honor for our field and makes us stick together
- Syntax is where we got our reputation and don't forget it.

English syntax
- Possible to have good syntax, nonsense semantics
- "Colorless green ideas sleep furiously"
- When England conquered Ireland and stole everything in sight, they stole Irish grammar
- Galore would have been stuck in the same place as if they spoke Gaellic
- This sort of process of having good syntax but nonsense can also happen in programming language
- C program was perfectly good syntax but it was going to dump core if we ran it.
- Can we have a C program and we can run it, but it is syntactically bogus?
- NO, you cannot!
- Syntax has one kind of meaning in natural language, and natural languages are very flexible and supple and people will figure it out.
- It was just a long string of rambling stuff if you keep track of every word I said.
- It has got to work because we have automaton that are reading your program and figuring it out.

- "Time flies" - busy biology professor and ask grad students to time flies outside OR time goes by fast
- Tell our robot to time flies even and watch out for **ambiguity**
- Huge problem that we deal with in programming language syntax.
- It is going to be really easy to write down an ambiguous grammar

Use a syntax people are used to when designing your language
- Which syntax is better for adding?
- a + b vs. a b +
- Inertia makes people choose one syntax over another
- To some extent, parentheses are a sign of weakness
- Contribute nothing to the semantics of your language, but an indication that the people didn't know what they were doing.

Look at entire program structure and look at the form of your program
- It consists of a file system!
- Understanding how comments works is a sub-syntax

HW 1 is a syntax HW and will it worry about tokenization?
- No!
- What is a token?
- Tokenizers read programs left to right and greedily

- If you can form a bigger token, good! Absorb and it make that bigger token
- When you see a language for the first time, you have to deal with the issue of what is a token?

Eggert's colleague was complaining because a crucial part of the algorithm didn't work the way he wanted it to.

- Horrible business thing calculating taxes in northern Japan
- Blame Denmark for this issue of tokenization!
- ??/ - Dane's were using terminals that read Danish, not English
- They changed the key caps on their terminals and instead of seeing a #, they would put A with a ring over it!
- Tokenization problems will bite you somewhere and there will be issues

Q. Did you know about the Danish type thing before?
A. Yes, but Richard Stallman turned it off in gcc, but he supported it if you specify an option called "brain damage"

- You should never put two question marks next to you in a C program

You can memorize the Unicode code point, but files are sequences of bytes

- Here is a byte representing the t and space, and it works if it is ASCII
- You will need more than one byte and it uses an encoding called UTF-8

Suppose you can put any 17-bit payload

- Put all 0's, which is the same character as the null byte - you would think you are at the end of the string
- Make these all 0's except these bottom 6 bits

Cyrillic o - does something completely different

- If you are NOT worried about this, wait for someone to break into iOS and then it is a tokenization issue where we break into systems.

Languages without reserved words

- IF (IF = B): More flexibility for the programmer
- Virtue of mutability!
- Suppose you want to add "until" for C
- Argument of that in C is that you will break programs and make existing valid code into invalid ones.
- The problem with the C approach is that it makes additions to the C language more difficult to implement
- Keywords get broken
- C folks did not do that because they were on a 16 bit computer and they didn't have space

Never use > or >= in your programming!

- Discussion will talk about the first homework assignment!

W 1 Dis 1-13-17
http://web.cs.ucla.edu/classes/winter17/cs131/
https://piazza.com/ucla/winter2017/cs131

TA slides at https://goo.gl/YatvLg
Email: shyoo1st@cs.ucla.edu (Seunghyun Yoo)

SEASnet account: https://www.seas.ucla.edu.edu/acctapp/ - HW 3

Homeworks 40%
- OCaml, Java, Prolog, Scheme, … (75%)
- Python (25%)
- Midterm 20% / Final 40%

High correlation between HW Score and final letter grade

OCaml
- We can avoid synchronization issues and for Java, we have a garbage collector so we don't have to derive runtimes of the variables
- Functional programming - avoids changing state and mutable data

Factorial in C
- It does NOT follow a functional programming paradigm because it uses assignment and iteration
- No side effects.
- static makes the variable global and each time we modify the variable, there will be side effects

Factorial in OCaml
- Differences
- Syntax is different from C
- OCaml follows the functional programming paradigm, but C uses a loop
- OCaml doesn't have explicit type declarations

Side-effect
- Depends on if the function is side-effect free.

OCaml: Function Definition and Call(1)
```
# let square x = x * x;;
```

OCaml: Function Definition and Call(2)
```
# let add x y = x + y;;

# add 1 2;;
```

- **fun** keyword
- Hidden variable x and it returns a square of x

```
# let eval op v1 v2 = match op with
  | "+" -> v1 + v2
  | "-" -> v1 - v2 ;;
```

Sample code:

```
(* utop -> eval `opam config env` *)

(* Let's define a function "is_even"
which returns true only if x is an even integer *)

let is_even x = match (x mod 2) with
| 0 -> true
| 1 -> false;;
```

OCaml: Pattern Matching (2)

```
# let imply v = match v with
          (true, true) -> true
|         (true, false) -> false
|         (false, true) -> true
|         (false, false) -> true;;

# 1::2::3::[];;
- : int list = [1; 2; 3]
# 1::2;;
```

Error: This expression has type int but an expression was expected of type
int list

OCaml: List Manipulation (1)

- List.append
- List.append [1; 2; 3] [4; 5; 6];;
- [1; 2; 3]

```
let head lst = match lst with
  | [] -> failwith("there is no element")
  | h::_ -> h;;
```

OCaml: Comparison

- (=) : equality check
- (==): same object

Q) what will be the return values?

```
let a = [1 ; 2];;
a = [1; 2];;
```

a == [1; 2];;

Q) What about 0. == 0. ? 0 == 0?

Q) Is (=) a function?

HW #1: Context-free Grammar

- The terminal symbols are generated by the grammar

W 2 T Lec 1-17-17
Syntax

- Think of a sentence as being a finite sequence of tokens (terminal symbols)
- Identifiers stand for a set of letters, but from the POV of the grammar, we don't care how the identifier is spelled.
- We are just doing syntax here, so we don't care about semantics (we are ignoring semantic information, at least for now)

Q. How would you compile an infinitely long program?
A. Doesn't make as much sense when we are talking about other sorts of languages.

- You might want to allow potentially infinite sequences
- Stay in a formalization here.

Q. Can you imagine a language with an infinite number of tokens?
A. English has a finite number of words in the dictionary, but this is fuzzy because we can make up words

- We want to think of it as a finite set, but we want to analyze a language.

You can write down as many a's as you want, and then write down the same # of b's

- Any other string is outside this language
- Infinite language even though each member of the language is finite
- **Languages don't scale very well here, but if you try to specify C or Java with a notation like this, it isn't going to scale very well!**
- Need a specified way of setting all the tokens
- In addition to all this stuff about tokens, you invent a new idea about nonterminal symbols
- Strings that are part of a sentence but do not comprise the entire sentence.
- **Breaks things down into manageable pieces!**
- Carve out a little piece of it and say that this part is a noun for this.

Grammar is described by a set of tokens or terminal symbols as well as finite set of nonterminal symbols

- It could be a noun phrase like "the" or something more complicated!

You could do simple languages in a more complicated way

Grammars

- Describes what the syntax of the language is
- If anyone else wants to learn C, they have to talk to one of us.
- We don't do that with programming languages because we write down exact rules for syntax.
- You have to mentally program in your head and prove that this is a valid sentence in my grammar.
- Apply rules over and over again
- For really small stuff, grammars are overkill, but it helps explain syntax of larger programming languages like Java and C

Whenever a Java compiler compiles your program, it has to run through the steps of this proof.

- Look at the grammar and figure out if this is a valid program in the Java language and mentally run through the steps in this proof.
- Construct a data structure that looks like this syntax tree to figure out if this was a valid Java tree
- If our first assignment was to write a C program to parse this language, you could have called get char and then start counting b's
- It is harder though if we are trying to write a program to recognize Java or C
- For that program, you take a look at the arbitrary rules and check them against a particular input and see if it works.

Recursive grammars - interesting ones!

- Grammars with no recursion are kind of boring

Internet RFC - standard for email defined by John Postal in the 1970s

- Specify every byte and every email has to conform to this standard
- Although it was written in a polite way, they have to conform to the RFC standard
- You could have written it differently!
- You could interchange those two rules and it wouldn't matter.

Mailing list server says that we receive two copies of the message, and the two copies have the same message ID so we know it is the same message and I only need to show one of these messages to the recipient.

- Up to the email client to put things in here.
- Can you put any characters you want in there? What are the patterns of characters that are left?

Internet RFC does NOT use -> because the terminal John Postal had did NOT have a key that looks like this on the terminal (->) so he uses an = instead

- This is also why he used ASCII for his other stuff
- In Postal's syntax, you put quotes around terminal symbols i.e. "<"

- Minus sign acts like a letter here "-"
- Meta-notations: This is not strictly BNF, so we call this meta-notation as EBNF (extended BNF)
- This extra power is simply a notational convenience
- Doesn't let us define new languages that we haven't defined before.
- Turn it into one or more BNF rules

EBNF is BNF stripped down to its basics

- EBNF is better for writing programming language, BNF is better for higher level understanding and overviews

Breaking into someone running email clients and we are trying to find bugs in email clients

- Construct a message ID that is as tricky as possible
- Could you construct a message ID that is trouble with this grammar?
- You could insert JavaScript like a XSS (put a lot of JavaScript code into a message ID)
- Quoted string that contains your JavaScript program, but how is that going to break into your system?
- Treat this as a big string that they look up to see it is a duplicate of another string
- **Use a really long string!**
- **Message ID that is a GB long**
- Unfortunately, there is a separate rule in RFC that prevents you from doing this.
- **Grammar is doing stuff that it is good at, and other part of RFC focusing on other things**

qtext can have any character, so in particular, you can put a null byte here!

- Look at a grammar because it can tell you whether or NOT a message ID is valid or not!
- Throw out email whose message IDs don't conform to this grammar
- When spam was first becoming popular, Eggert had a spam filter to check if the message ID was valid or not.
- You would be surprised to see how many spammers don't generate valid message IDs
- Filter out invalid message IDs using a regular expression
- **Wrote down a regex that recognized valid message IDs**
- Given you a grammar here but this particular grammar has an interesting property.
- Convert this grammar to a regular expression!

Eggert's regular expression for email

It will see metanotation, terminal symbols, but NO non-terminals!

- Is there something you can do with BNF that you cannot do with grep

- a^n b^n is one of the simplest regular expressions you cannot generate a regular expression for!
- **Regular expressions cannot count (are memoryless)**
- **Can you look at a grammar and easily tell if you can convert it to a regular expression?**

Q. What is a regular expression?
A. This reads from standard input, finds every instance of this pattern, and then prints this line.

A grammar for REs
- Any BNF grammar where you cannot do it this way

Troublesome grammars generally involve recursion
- If all the recursion is **tail-recursion**, you can convert it to regex
- If the recursion is at the middle, you probably **CANNOT** convert it to regex!
- Extremely useful things here!
- Intrusion detection - based on regex looking for incoming packets (bytes inside) to check if the packet is trouble or not.
- Based on regex since regular expressions are handled more easily than arbitrary BNF
- You can implement them without using recursion!
- Intrusion detection rules are pretty rare in practice because hackers can attack a rule like this by sending lots of a's
- Blow your program's stack
- Getting the intuition by a regular expression and the most popular use of grammars is the usage of regular expression

There has been an effort to come up with a standard for EBNF
- If we see it here, you probably won't see it elsewhere
- You can use either single quotes or double quotes
- You can put anything in square brackets and that makes it optional
- X - Y: instance of X that is NOT an instance of Y
- Exceptions is very common in grammars
- X, Y: concatenation
- In ISO standard, they decided to use comma to concatenate
- X | Y: OR
- We cannot just hand wave our definition; we need to define EBNF formally!
- What formalism should they use to formally define the syntax for ISO EBNF?
- **Apple is pushing this language called Swift**
- **Swift is their compiler!**
- **Any compiler is written in itself**

- Use own formalism to describe itself in its formalism

syntax rule = meta id (non terminal), '=', definitions list, ';';

- There is something philosophically wrong about this.
- Defined a formalism formally, using the form itself

Blizelfix(?) - circular definition and you are NOT allowed to define something in terms of itself

- How would you understand it afterwards?
- Philosophically unappealing!
- We are always doing stuff recursively ever since kindergarten!
- When people said to look up words in a dictionary, what does this mean?
- the - the definite article

Problem c grammars: As languages get more complicated, they get increasingly large and hard to understand

- If you start reading C++ rules, you will forget the first half of the grammar as soon as you start reading the 2nd half of the grammar
- Eggert has run into this problem when using SQL
- The problem is that every database vendor has their own SQL extension using grammar rules
- Some vendors have used a different technology using syntax charts or racetrack diagrams

Apply this principle to a relatively simple grammar and one in which we ask what is the point?

- We are talking about very large grammars and we are trying to illustrate it with a small example

Scheme

- Variant of Lisp and it is a terminal symbol
- Condense things into smaller diagrams
- Then you got to put in all these arrows, but you want something that is easier to read than the original!
- They are a hint on how to implement a parser for a language
- Think about writing code that goes through this chart.
- When you are here, you see the next symbol is cond and there is an arrow
- You look ahead by looking ahead!
- If you think about writing a program to parse a language, a good way to do that is writing a syntax chart.
- Sometimes you won't be able to do this lookahead but it is a good first cut.

Apply a certain grammar rule, but you won't be able to go any further because there is no T on the left-hand side

- Never produce a sentence from this grammar rule and just ignore it
- Suppose you are trying to parse a sequence of tokens, which grammar rules should I use to prove it is in a language?
- No way to show any sequence is a T because T is never on the left-hand side of anything
- Bug in the grammar because it is pointless

W 2 R Lec 1-19-17

- too much detail - you can be too ambitious in a grammar, and this can lead to problems
- ambiguity - this is is the problem of too little detail
- We have to be careful to avoid both of these problems

Ambiguity

- Comes from multiple ways of parsing the same sentence
- When you write your own grammar, you will have to worry because you can get different answers depending on what compiler you are using
- It sounds like we have cut ourselves a problem that is too big to solve, but it really isn't that bad.
- When you write a program that has some bugs in it, will it loop forever? You have to find out!
- You gain some confidence after you run and debug your programs!
- Look at grammars and decide whether they are ambiguous by reading them carefully.
- Don't solve undecidable problems in general, but do it for practical grammars when you write actual programming language grammars

Change the grammar to prohibit the parse you don't want

- Too generous and allows parses that you don't want
- Make the grammar picky and more specific
- You can't put a + and - after a - because you can have parses you don't want!

Associativity matters even for addition
Q. What about string concatenation?
A. I hope it is associative!

- Some languages have this as a problem!

This example shows how to resolve ambiguity of associativity

- Another thing to consider is precedence
- Infix operator has to be your rule of thumb
- NOT always true, but true so often that we can make assumptions it is TRUE
- If we solve the associativity ambiguity and precedence ambiguity, we will typically solve a good chunk of the real world problems.

The problem of ambiguity is undecidable, but it is hard to tell.

Example in C:

- Putting semicolons and nothing else
- When I have an assignment statement in C, that is a misnomer. It is an assignment expression followed by a semicolon

Q. Difference between expression and statement?
A. expression: a = b[0]
b + 5
statement: b + 5;

Let's make a new language called C+:

- Every C program is a valid C+ program because you have to put parentheses in C grammar and it will be a valid expression
- This doesn't create an ambiguity because you cannot put a semicolon in an expression in C and you keep reading until you find an expression.

No token to tell you when a statement stops

- When parsing C+ program, you are reading your program left to right and we are done with the while part and there is something wrong with it.
- This grammar is ambiguous, but we want to know why is it ambiguous?
- You come up with a program that can be parsed in two different ways.
- You have to have parentheses to resolve ambiguities between infix and postfix operators, which can have multiple meanings

These parentheses are NOT necessary to prevent ambiguity

- They just put the parentheses in the do while because of consistency.

- **Optional expressions in C are EBNF**

In CS 31, they mentioned about the dangling else problem!

- One of the rules is too generous and is allowing parses we don't want.

Solutions to the ambiguity problem can cause too many details to pop up!

- Concrete syntax: Run through the simplification process by showing every rule expansion
- Abstract syntax: Simplify it down by showing the start and end result of all the rule expansions
- Compilers are written this way by reading a string of tokens, and not even bothering to create this data structure because nobody cares
- Tools available to take ambiguous grammars like this and deal with ambiguity in the usual way.
- Generate parsers in the usual way.
- To some extent, this is a solved problem for associativity and precedence.

- A lot of these tools can handle dangling if's, but ambiguity in general is undecidable.
- The usual advice for this is to NOT do it!
- If your language is so complicated, what can you do?

Capturing important notions but it made your grammar explode

- Attempts to cover too much!
- Why don't people do this in real world programming?
- **There is an infinite number of types in C**
- There will be an infinitely long grammar by creating more types in C, so this kind of approach, which attempts to cover extra detail in the language is doomed to failure in any practical language.
- Even then, it is problematic because let's suppose we have a function call here.
- If you call an integer function (IF) and specify some args, then that is valid

Binary attribute will grow by a factor of 2

- Type vs. storage class -> then the grammar would grow exponentially based on the attributes you have.
- This kind of approach isn't going to work.
- **Grammars work really well to cover syntax (form of a language) but they are NOT good to cover semantics (meaning of a language)**

Prolog Notes:

- C decided these operators are left-associative so this is equivalent to comparing a to b
- You can't ever introduce ambiguity because functions prevent ambiguous calls

0. Side effects in expression

- Behavior can be undefined and C compiler and do whatever it wants to.
- You might think, I would never write stupid code like this!
- Java designers hate this, so you evaluate expressions left to right.
- Avoid ambiguity there, and this assigns 0 to a in Java (all the time)
- C doesn't have this rule because they want the compiler to generate more efficient code
- Can be 10-20% faster and this is a big deal for millennium dollar cloud computing
- This is why they have rules for competing side effects

Real world example with Xerox Park

- Competing side effects means behavior is undefined, and gcc decided to generate inefficient code
- The problem is that compilers will try to make different optimizations, which can lead to errors!

- Each compiler optimizes differently so there is no universal way to interpret the code!

- One of the long-standing dreams is to operate programs more efficiently than C programs do.
- Good at what it does, low-level and close to the machine
- Sometimes, its performance is quite bad because you are forced to do things in a very narrow way.
- If these two goals, you have to start off with clarity.

You are used to that and it is hardwired into your brain.
- Something with side effects in it.
- Run into trouble doing ordinary things.
- Order of evaluation and side effects are two problems that can hurt you greatly!

Computer scientists have a reputation of being stubbornly individualistic and hackers and they argued for clarity.
- Build on mathematical traditions rather than throwing it out the window and we got some horrible junk
- People would try to debug their programs and they won't work and we don't know why.

Performance - languages like Java and C are basically designed for the Von Neumann machine

- There are some computations in which you can do a lot of work in the registers like cracking someone's password, so there are some calculations where the von Neumann bottleneck do not apply.
- However, this will generally lead to a performance disaster.
- Talk to memory with more than one load or store.
- Performance motivation is to avoid the von Neumann architecture
- Allow parallelism in a useful way to prevent us from tearing our hair out.
- Multithreaded programs in C++ and Java are a way to avoid this bottleneck, but they have **race conditions** galore, and there is no way to avoid that.
- We need a better way of taking advantage of multicore or cloud computing.
- It promises a way to build these systems of the future that have good performance, easy to understand, and don't suffocate the user.

Success stories of functional programming
- There is a pair of functions in LISP and OCaml where you give it a list of items and you can apply to a function to every item in a list to return another list

W 3 T Lec 1-24-17
Functional programming leads to cleaner and faster code!

What is a function?

- Walk away from the CS definition which is a chunk of code with a name, and head towards a mathematical definition, which maps a domain to a range

Math view: No notion of executing one function and executing another function.

- Evaluation order is NOT imposed by sequences
- No series of function calls

The only prerequisites here is that the composite function call the inner functions before the outer function.

- You can call g or h in either order and it doesn't matter

Functional programming avoids over sequencing options and removing dependencies so that the system can generate more efficient code and it becomes more clear!

- Problem of side effects is something to consider
- It is possible for x to modify a variable that g uses.
- In that case, you canNOT run the functions in parallel
- By writing code in this style, you are imposing order in computation because the compiler won't necessarily know.
- Sequencing operation can make things more efficient but they can get you into trouble.

You want optimization to be okay at all times and remove semicolons

- Let the compiler do the optimizations that it can
- A corollary of this is that the compiler should reason about variables without worrying about these side effects.

You cannot do optimization in Java like this because the P.L. evaluates left to right in order.

- Name should refer to a value in a way that is transparent and works like math
- Should NOT have tricky names and should NOT have tricky values!
- Avoid side effects like i++ which will change its values.
- Value will depend on context of what you look at, and if you are not careful, it can change its value underneath.

Let's give up sequencing - where is the good stuff?

- Use **higher order functions**
- Take in other functions as parameters!
- lambdas - passing in functions
- **Composite functions are a good example**
- Think of a reasonably high level of abstraction and write computations in terms of building bigger functions out of smaller functions

- These do some things and you want to build something bigger out of them.

Many functional programming languages

- Compile-time type checking (like Java, C++ …)
- In C, C++, you always write down the types of things which is super annoying!
- Foo x = new Foo();
- In OCaml you usually don't have to write down types at all: **Type inference**
- OCaml is like Python in this sense
- OCaml assumes you have a garbage collector unlike C++, but similar to Java
- Create objects to your heart's content without worrying about deletion.
- Higher-order functions are emphasized

Programming Portion:

- Names and types are **HEAVILY** emphasized!
- When faced with a new software system, try taking a hammer to it and see what would cause it to screw up.
- Static type checking and conditional expressions
- Computes types for then and else statements and **they MUST be consistent!**
- Throw error messages if they are **NOT** consistent!

(1, 2) -> throws away 1 and keeps 2 in C
(1, 2);;

- : int * int = (1, 2) - **tuple**

"*" - Cross product

- Less flexibility of tuples, but more flexibility in terms of the type of items that can be in them
- Lists are more uniform but rigid

What we have here are NOT just types, but rather generic types!

- You don't know all the details and these are crucial to make our higher order functions work.
- We want it to be as general as possible by using these generic types
- By the time the function runs, we have to know by then

One of the things that got Eggert confused about OCaml at first is that every function has just one argument

- This is super janky!
- Standard function **cons** used for building list items

- In OCaml, functions only have one argument

You have defined a symbol called “cons” which prints out a function value

- You can print out the machine code that implements the function and this is a consistent way of printing out a function value
- The interesting part here is the type of the function!
- Cannot call cons in certain ways that would break the rule of list building!
- NOT the real version of what is going on, but rather syntactic shorthand

Pattern matching

- First item will be matched to x, second item will be matched to y
- Output is x::y

let n = 37 * 37

- You can write down functions via lambda expressions if you haven’t already!

Match expression: evaluate some expression and give possible patterns that it might match

- Take two components and call a built-in operation
- n is only one pattern in certain cases

Q. Is the process of determining the type of a function similar to the process of parsing a grammar?
A. You have the entire expression in your head and infer types as you go.

- a has to be a tuple with two components, so mentally, it involves something crossed with something
- Whatever type x is means that y has to be the same type of list
- This is the only thing the :: operator works with

‘a has to be a ‘a list type!

- It needs to have the entire function definition in its head before it does type inference
- What if the match is a totally different pattern? How does type inference infer type of a

Every constant in OCaml has a type - match themselves (type is known)

Series of commas with patterns will match a tuple!

- P_1::P_2 matches a list with a head of P_1 and tail of P_2 (length >= 1)

x is like a local variable and we just get the value and use x later on because we have bound it to whatever it matches
_ is like x in the sense that it binds to anything but it throws away a variable after

- If I give this list an empty list, there will be a runtime error so the compiler warned that I defined a partial function
- Some elements in its domain might NOT work!
- Partial functions indicate your program is fragile and incomplete!

Advice for OCaml hw: Pay attention to gibberish and look for uncaught cases

0. Prevents errors
0. You will know what you are doing better

- A lot of programmers brought over bad habits from C and Lisp
- If you need to calculate the first item in the list, you must use pattern matching
- If you absolutely know that a list is non-empty, why do you have a list of length 1 or more
- The idea that you consciously keep track of a list of at least length n, it is problematic!
- Write reliable code and try it out for these two assignments

Q. Do you get a performance increase for using wildcards "_"?
A. At the scale of our hw it doesn't make a difference, but it does in real world OCaml applications

- fun x -> match x with
| 0 -> fun x -> -x
| _ -> fun x -> x * 2

- **Arrow operator is right associative**
- **function application is left associative**

Currying - take functions and make it into a chain of operations

- Writing papers on this back in the 1940s before computers existed.
- His paper had too many arguments and after a while, he would write papers about one argument.
- If someone gives me a hard time, "curry" it!

In OCaml, the addition function can be curry'ed! - tuples are less elegant!

- All those functions took exactly one argument; you just didn't know it used currying!
- To some extent, it feels a little weird
- Given a list of integers, give me the largest integer in the list
- The main thing I want is to work on any type!

- Take the troublesome part of the code, throw it away, and error handle it!

Q. Are you supposed to send in a list? Is it supposed to have 3 arguments?
A. Call a high order function and use it to compute a lower order function.

Function knows what to match to and the argument is nameless.

- All we know is the argument matches these patterns but we don't have a name for the arguments
- Here, we are using syntactic sugar and we are always writing this way.

Reversing a list

- Takes a list apart and looks at it.
- Look at types and make sure they are what you want them to be!
- Don't waste time running it if you know there is an error!
- Fix the bugs or warn you about bugs before you run them
- Eggert's submission was super slow because it is O(N^2) complexity
- You have to traverse the entire list and append at the end using this way

W 3 R Lec 1-26-17

- hw2 has a parser but you also have a program that is a generic parser
- Another way of thinking of it is a program that takes a grammar and generates a parser
- Finite state automaton (FSA): A DFA or NFA can represent a regex
- hw2 you are asked to read a parser for any context free grammar

Turing Machines are too powerful in general and there are some issues accessing parts of the infinite tape

Pushdown Automaton - you cannot wander around the stack, but you have an infinite memory because of the stack

- By insisting you can only look at the stack top, you give up a lot of theoretical power but you gain a lot of understanding on what pushdown automaton will do
- With a pushdown automaton, you can parse any grammar!

Build a function that would represent push and pop operations explicitly as it parses the input

- With hw2, what we are trying to do is figure out the way of getting this pushdown automaton without having to actually write all the code
- Solving the problem more abstractly and **a lot of software engineering is a problem of scale**
  - A scheduling at Ronald Reagan is done by hand
  - **It should be done automatically!**
  - Write programs that solve stuff like this but we need to take these big problems and simplify them down

- Break things down into subproblems in order to make it practical
- We need to think of a way of solving hw2 that doesn't involve writing so much code.

HW 2 tips

- Parsing
- 1) recursion (goes with a territory)
- 2) concatenation - we want a function to parse instances of S
- In reality, a parser will take an input string and return a Boolean saying whether or not it parsed it successfully
- C or Java program can be recursive
- You want to pass in an arg of char** because you want to move the pointer forward
- This solves concatenation because it approaches the input in **sequence**

Suppose this is what our rule for S looks like:

- Look at the next input symbol and you could still handle OR using a traditional programming language as long as you have nice rules where you can look ahead to the next symbol and figure out which branch of the symbol to take.
- **Lookahead:** Good preliminary step but not enough to solve the problem!

- Don't write this grammar - write it in such a way that there is no ambiguity
- As long as T and U don't start with the same symbol, you can take advantage of lookahead to tell which path you will go down

Problem in hw2 is when you have two alternatives, you don't know which alternative to take even with lookahead

- Why not just try both?
- **In order to implement a parser for these roles, try this first and if it works, fine! Otherwise, try the other one.**
- At runtime, when you encounter this situation, if you don't know which of two rules to take, you start parsing according to the first rule but you create a memory of where to go next.

Use a stack in hw2 to figure this out!

- Whether writing a program in C, Java, or OCaml, it doesn't really matter.
- **Whenever you call a function in C or Java, you are pushing the return address onto a stack! When the function call ends, it gets popped off the stack**
- **Use the built-in stack built into C, Java, or OCaml.**
- You could build your own stack or finite state machine, but obviously a lot more work!
- We want to write a small amount of code instead.
- The 2nd assignment when it talks about matchers and acceptors is really saying there is a skeleton idea of handling the problem of disjunction of using the stack you already got.

acceptor - takes a string of tokens and tells you whether or not it likes it

- When trying to build a matcher for s and you don't know which two alternatives to take, you use the acceptor to tell you whether or not the match you found is what you wanted.
- Otherwise, you may have to backtrack and try another alternative.
- The matcher for this set of rules will build up its own acceptors that it will pass to subsidiaries
- To parse an instance of S, you will try things and if the acceptor doesn't like it, you will backtrack until it finally finds something that is acceptable.

Inside a parser is a packaged up version of your entire input

- Acceptor: **a bucket list; the list of things the acceptor wants to do before it dies**

High-level: Each function call passes in a different acceptor based on what you pass in

- Glue together obvious little bits of code
- The bits are obvious, the glue is NOT obvious
- **Recursion and concatenation come in fairly easily, but OR (disjunction) is a lot harder!**
- In order to do functional programming, you have to compute with functions (building more complicated functions out of simpler functions)

As we do concatenation, we will get enough confidence/experience to tackle OR later on

- Build a recursive system and break down larger problems into smaller problems
- How do we build a matcher for S?
- We want to narrow down the component at the top (S)
- We need to construct a matcher for S
- What is the matcher for S given the matchers for T and the matchers for U
- Whatever the complicated acceptor is, we have to pass l at the end

Whatever remaining token list that you have, it has to be something U matches, and when U is done, the result has to be acceptable to a

- The matcher for U is passed in a and you pass in a list

Value of l gets passed implicitly through the function calls

- Be able to write sophisticated function calls without having to name the functions or their arguments
- A lot of times, you don't have to think about it because the functions just work.
- That is how to handle concatenation and that is how we solved problem 2

Problem 3: Compute the matcher for S, and assume we have the matchers for T and the matchers for U

- What is the matcher for S going to look like?

I already know the matchers for T and U.

- The matcher is a function that takes an acceptor and does something with that acceptor
- Our function should try to match T

- Constantly compute acceptor functions on the fly and during parsing, construct new functions

Let's say we are trying to parse S and we have a very generous acceptor function that always returns true

- The problem is that when we start parsing, we don't know where the boundaries are that separate parts of the string
- We can have an inefficient algorithm that takes a look at all N symbols and finds all N^2 ways of dividing this into two different pieces and solve them all.
- The problem is that it is too slow and it is O(N^2) at each level, so O(k * N^2) at all levels == O(N^3)
- We want something faster than O(N^3) for typical grammars
- Keep trying other solutions until the acceptors keep saying no
- T could keep trying different approaches and instead of all O(N^2) possibilities, try the possibilities that T works and U works

Acceptors

- S has two functions for the matcher of T and the matcher of U
- The acceptor for S is the same as the acceptor for U
- When U is done, whatever is left over and U's acceptor is the same as S's acceptor.
- We just need a separate acceptor for T

Every time you call a matcher, you give it 1 acceptor!

- How do you know what the acceptor of U is or do you keep recomputing the acceptors in every sub matcher?
- This has to do with how functions are built in OCaml and ML. This differs from Java and C

In languages with C and Java, we work around this by providing extra parameters since we cannot do currying other than non-trivial methods

- When add returns, we are calling add but its local variable is still there.
- The upside of OCaml is the we can do currying but there will be more pressure in the implementation
- Sometimes, function calls have information that survives the call!
- The technical term for this is a closure!
- The code is pretty simple: add two numbers and the context is x = 1

- Closure can be an instruction pointer and the environment pointer is what context we pass it in
- If it sees you pulling stunts like this, it says “WHOA, don’t use a stack for this particular function”
- It will probably put that closure onto the heap and do something like that.
- You can have combinations of heaps and stacks
- When functions return, they don’t pop everything off the stack.

If you try to write this code in C or Java, it won’t work!

Q. In the last recursive call, when you pass an acceptor and it goes to the top, how does it remember the correct value?
A. You call the acceptor that was created for your context, which may have been a while ago!

From Ch. 2 in the book

- Talk about this more from a programming language viewpoint

The first step of translation is the preprocessor!

- If this is notln.c, the output of preprocessor takes this, removes the #include, substitutes the content of stdio.h, and expands the macros of what we have
- For efficiency, getchar is implemented as a macro
- Take input program and turn it into hw1
- Your hw file is a sequence of tokens
- Take a sequence of bytes in the rest of your compiler, and it will get slow and confusing

Tokens are going to be one per input symbol

- To some extent, we have thrown away too much info!
- Doesn’t tel us all this extra info (**lexenes**)

Parsing

- The more common representation inside a compiler is a parse tree:
- Program is a series of definitions

Scope checking

- Define getchar
- Use getchar
- Keep track of all identifiers that are visible at every point of the tree
- Most of the time, print out an error message to a code
- We have to go on and then we typically do intermediate code generation
- The output of this step isn’t machine code or any particular machine. Instead, it is code for an abstract machine
- Push either 0 or 1 depending on if the input word was 0 or non-zero

The intention of gcc is to import it to other processors easily!

- Afterwards, people have to write other sections of code and making it independent of machines

IDE is written in itself, and you can use it to inspect itself

War between IDEs and software tools
- Both sides are winning this war
- If you run Eclipse, it is a frontend for these tools
- You can also use Eclipse code which is a frontend for itself
- These were assumed to be oppositions to itself but they ago alongside each other

W 3 Dis 1-27-17
- More of OCaml List Module
- Do an example of user-defined types i.e. binary trees
- Homework 2
- Java Introduction
- Be aware of what is going on
- Might be shorter today

List.module
- We have popular API functions like List.rev, List.flatten, List.map

List.map
- You have a list and you want to apply a function to every element of that list
- It is annoying to write for loops, so you can do it in a nice clean way with List.map

List.filter
- You have a boolean function and you check if element is even or not
- You have some boolean function and you want to filter the list

List.fold_left
- The thing about map is the every function is aware of its current element that it is working on
- You have some accumulator to remember what is going on i.e. summing up the list or doing other helpful things
- Performs an operation on a list!

List.map2
- Lit.map2;;

```
List.map2;;
- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>

# List.map2;;
```

```
- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
# List.map2 (fun x y -> x+y) [1; 2] [3; 4];;
- : int list = [4; 6]
# List.ap2 (fun x y-> x + y) [1.; 2.] [3.; 4.];;
Error: Unbound value List.ap2
Hint: Did you mean map2?
# List.map2 (fun x y-> x + y) [1.; 2.] [3.; 4.];;
Error: This expression has type float but an expression was expected of type
         int
# List.map2 (fun x y -> x +. y) [1.; 2.] [3.; 4.];;
- : float list = [4.; 6.]
# List.map2 (fun x y -> x *. y) [1.1; 2.2] [3.3; 4.4];;
- : float list = [3.63; 9.68000000000000149]
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold (fun acc x -> x + acc) 0 [1; 2; 3];;
Error: Unbound value List.fold
# List.fold_left (fun acc x -> x + acc) 0 [1; 2; 3];;
- : int = 6
# List.fold_left (fun acc x -> x +. acc) 0. (List.map2 (fun x y -> x *. y) [1.1;# List.fold_left
(fun acc x -> x +. acc) 0. (List.map2 (fun x y -> x *. y) [1.1.1; 2.2] [3.3; 4.4]);;
Error: Syntax error: operator expected.
# List.fold_left (fun acc x -> x +. acc) 0. (List.map2 (fun x y -> x *. y) [1.1; 2.2] [3.3; 4.4]);;
- : float = 13.3100000000000023
```

Common List Operations

```
# List.rev [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_left (fun acc x -> x::acc) [] [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
# List.fold_left (fun acc x -> acc @ [x]) [] [1; 2; 3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]

# let map f l = List.fold_left(fun acc x -> acc @ [(f x)] ) [] L;;
Error: This variant expression is expected to have type 'a list
       The constructor L does not belong to type list
# let map f l = List.fold_left(fun acc x -> acc @ [(f x)] ) [] l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun x -> x + 5) [1; 2; 3; 4; 5];;
Error: Syntax error
# map (fun x -> 2 * x) [1; 2; 3; 4; 5];;
- : int list = [2; 4; 6; 8; 10]
```

Midterm

- Given a function, what are their types

Binary Search Tree

- Make sure that for the current node, the elements on the left are smaller while the elements on the right are larger

HW 2

- High-level hints at HW 2
- There is an older hw you can look at if you still don't understand how it works
  - HW 1 is represented as tuples and it is a set of tokens

Derivation of 3 + 4

- Parser has a list of rules and we try to apply rule to a string to get an expression that makes sense
- Apply rules to get an expression and keep applying rules

Matching Prefix

Recursive program that tries all these prefixes and applies derivation rules to see if anything makes sense
Ex. Find all matching prefixes of 3+2-6 in order

Answer:
1st: 3+2-6
2nd: 3+2
3rd: 3

Acceptor

- Determines whether a given input is **accepted or not**
- Eggert provides the acceptor, so do NOT worry about that function!
- **Given top-level acceptor, but we have to write the internal acceptor**

Pure Object Oriented Languages

- Everything is an object
- A program is a set of objects telling each other what to do by sending messages
- …
  - Java breaks these rules

The Object Concept

- An object is an encapsulation of data
- Has state so we can change the state of the object
- Behavior, given a function - define the behavior of the object and an instance of an *abstract* data type

- You can extract info from it and do stuff with other objects.
- Running object on other fields and so on and so forth.
- String and we write stuff out to our hard disk.
- Objects are implemented via things called classes
- C++

ADT - Data with methods
Java - the outside world is NOT aware of the data in all functions, so you can only use some part of these functions and robustness

Encapsulation and information hiding
- What the “outside world” cannot see it cannot depend on it.

Class vs Object
- An object is an instantiation of a class
- Object is an entity you can talk to and it is in memory where you have a list of objects
- Class is a blueprint of what the object has

Type and Interface
- Every object has a type
- Account with functions you can use.
- If you want to instantiate a new object, you can have the name of a class and same as in C++
- If you want to do something with that object, do <instance>.<method>

Aggregation and Decomposition
- You can make it more modular and change a small class one at a time.

Inheritance
- A way of deriving a new class from an existing class
- You can jus make the class more specialized and make a diagram of how extensions should work
- See a few important things about discussion sections

- If you try to assign a shape to a square, things can fall or break.
- Everything inherits from an **Object**
- This is the parent type of everything

W 4 T Lec 1-31-17
Compilers vs interpreters
- Compilers improve performance
- They are harder to debug!
- It can be mysterious what happens to the compiler even if you are an expert in the machine code!

- Interpreters have byte codes that act similar to the source code (better debug ability)

- **For speed and debug ability, use compiler for production/release and interpreter for debugging!**
- Problem was that some bugs would appear in production that weren't caught in debug mode
- Nowadays, we see more hybrid approaches here

Hybrid approach e.g. JVM
- Debuggability was good but speed was bad at first.
- Java developers then added the concept of a profiler
- Art of capturing efficiency information about your software system so you could tune it later

Profiling
- Compile programs with "gcc -p" to figure out the hot spots of your program
- The other thing you could do is insert counting code
- Compiler inserts and we have to increment some counters
- Involves statistical guesswork since we don't know what gets executed exactly
- Modify the program and will run more slowly, so this is a heavier duty approach
- Java folks said we had an interpreter so our code is already slow
- Nobody would want it if we added profiling
- When you add profiling, that will tell you where all the hotspots are!
- Sections of the interpreted code that get executed.
- Execute the machine code, so in fact, we have two compilers in our system (NOT just one!)
- Break down this second part of compilation to go at runtime.
- This optimization tends to be relatively expensive so you don't want to bother with it if code gets executed once.
- **If code gets executed 1 million times, this is better!**
- Under this approach, you have both an interpreter and a compiler!
- If you run it with a flag, it is a classical interpreter approach

Q. What kind of problems do you see with this approach?
A. You don't get consistent performance because the decision of when to translate is a **heuristic**.
- Non-deterministic process
- In real time, this makes my program non-deterministic so you are at the whim of the system
- How will you debug and it places a great source of pressure on the reliability of this translation.

- Please translate this source code to the object code and send off a bug report.
- Over here, you will NOT see the machine code and there are some downsides to this approach.

Standard browsers i.e. Chrome, Firefox, etc. use JavaScript that uses this same sort of notion

- Compile them into byte code and selectively translate the JavaScript byte code into machine code for ARM or x86
- Done dynamically
- This kind of approach may explain why the traditional compiling people would rethink this
- Flexibility
- Don't think of compiler as a monolithic tool of software development
- Think of it like a module that you can plug in at runtime.
- The same idea applies to linking!

Dynamic Linking

- More common nowadays
- Libraries are linked through the initialization process
- Adds flexibility and makes your system easier to change

Weaknesses

- DLL Hell - multiple versions of a library (which version should you use?)
- Some modules want Version X, others want Version Y. You may want to use both!
- Performance is a bit slower!
- Takes some work to find this symbol
- A call to a dynamically linked function has to go through a level of indirection
- Untrusted code?
- In the static model, a programmer has more control over modules and linking
- Once you allow dynamic linking, your programmer may run code out of a directory that is write-accessible by other people
- Do you trust the JS code that is running in your browser?
- FUCK no!
- This is the biggest problem of dynamic linking; how do you deal with large collections of software of untrusted origin?

A lot of OCaml is about type checking, and we are a bit sick of type checking by now!

Types?

- Clarity - if you know the type of x, it will tell you something about the operation
- help humans understand the program

- safety, redundancy - helps you catch stupid mistake when you are writing the program
- performance - helps the compiler generate better code

Q. Are we referring about type inference?
A. Either way, you get all of these advantages and you have partisans who say ML went too far and you cannot figure out the types without running the compiler

- You should get them all whether you use heavy type inference or not much.

“Strongly typed” - all operations are type checked

- You can escape in C/C++ by type casting

Primitive vs constructed types - builtin to language

- A good amount of work used to be done in primitive types!
- float
- Is a type a data structure or something else?

Q. How exposed is the implementation of a type? How much does the user of a type know about the internal details of its representation?
A. This represents all the floating point values you can have or can you leave some of them out.

Q. Does the abstract type let you deal with all this information somehow?
A. NaNs are not numbers and they can never compare equal to any number. They don’t even compare equal to themselves!

- We are saying x != x could possibly NOT be true, which is quite scary!
- The C definition of equality doesn’t match intuition’s definition of equality

Tiny numbers if you put them on the number line are all in the middle

- Can you tell whether something is in that range?
- Do a range check!
- There is a level of indirection here!

Q. Can you write code that tells if a number is infinity?
A. Check if the value changes after either adding or subtracting.

- Doesn’t work on very large numbers.

Review CS 33 :(

- What is this business with +/- 0?
- Some people despised this idea!
- Another controversial area involved tiny numbers, which wasn’t enough for this.
- Underflow from positive or negative information
- Give you extra information about a computation gone bad.
- Should there be three-way 0?

- If you do a floating point computation and the result is tiny, the hardware traps and software is supposed to reach in and simulate those tiny numbers
- Why do we have these?
- Equality checks (Booleans)
- Tiny numbers guarantee that if two numbers differ, subtraction cannot make them 0
- The result will either be 0 or a non-zero tiny number
- Tiny numbers is the result of subtraction working the way your intuition is supposed to work

Every computer scientist should know how floating points work

- Notion of types is kind of a tricky one. NOT as simple as what the book says
- When you create a class in Java or C++, you are going to be dealing with issues like this.
- Decide how exposed to make the implementation. How much extra work to put into this implementation
- Important design decisions i.e. CPU cycles.
- Getting that right is NOT an easy thing and requires practice, work, thinking, etc.
- Arguing and coming up with a good type design

Type equivalence

- When are two types equivalent?
- **Structural equivalence: machine code generated for one is the same as the machine code generated for another (when it looks at typedefs)**
- **Name equivalence: structures with different names (when comparing structure types)**

Algol 60:

- Hard to implement so not too many languages do it this way

Subtypes:

- Types are associated with operations on those values
- Arguably, every float is a double, but NOT every double is a float.
- Every float operation can be done with a double

- Read C types from **right-to-left**

T has more operations that U does.

- char const * has less operations (read-only), while char * can read and write
- **It feels backward at first, but this is important to get right!**
- NOT only when we deal with pointers but when we deal with more complicated points in Java

Polymorphism

- Single function with a lot of type interpretations, so how do you pick the right one?
- Been with programming languages since the beginning
- This sort of ad hoc part of polymorphism

overloading: Pick the function symbol based on the context of which it appears

- Is this an integer "+" or a floating point "+"?
- Don't decide based on operator symbol, look at the types!
- Pick different operations at the machine level and figure out which is these it is based on the type
- With all of those built-in machine types, use x + y or x < y
- What are the rules used to overload the built-in operators?

coercion: Implicit type conversion to do type checking for you

- If the number that you write down is so large that it doesn't fit as an int, it tries a larger type

Coercion along with overloading has given us an interpretation that was NOT intended

Multiple arguments

- Creates an error message because it is semantically ambiguous
- The usual solution to this is to come up with other implementations that come up with a variant you want.

1970s

- When I start to define my own types, I run into coercion and too many problems

Parametric polymorphism

- The idea is that types can contain parameters
- This kind of polymorphism is crafted into C++ and now it has been added to a lot of languages
- Applying parametric polymorphism to Java next time.

W 4 R Lec 2-2-17

- Iterators let you know where the cursor is pointing to and you can go to the next object in the loop
- Each time through the iterator, you can say i.next() to get the next guy
- Side effect on the iterator because it gives the next guy, but it returns the item you want to go to
- You have a collection of strings and remove each string that is really short.
- Keep the long string.

- This doesn't work because Collection can contain any object you like
- Object is the generic object that is the root of the type hierarchy in Java
- **You cannot assign a subtype i.e. String to a generic Object!**
- Let's insert a cast for a runtime, type checking conversion
- Although this code works, it is deeply unsatistfying
- It is bad because we remove type safety because type checking doesn't work in this particular case.
- **NOT simply an elegance thing, but also a performance thing**
- Java interpreter will make this a string and we know that it will always be a collection of strings
- Given up on static type checking and doing extra runtime work.
- This is like what Python code does and we don't want to do that!
- Python has a lot of problems with runtime checks going bad.

Java Generics

- We have to say what the type of the iterator is over but we have gained static type safety
- We have also gained performance because we don't need to insert a runtime check here, so we don't know the value is of the proper type
- Seen this in C++ templates as well but slightly different
- Write generic looking code
- Take template T, instantiate it
- Instead of a T, you know what the data type is i.e. double and everywhere you saw a T before, you say double
- Similar to macro substitution, textually replace your templates with the type
- Java is different!
- Compile the generic version
- If you visualize it works, compile it!
- If NOT sure, print out a message and don't compile it for safety.
- **Compile one copy of the code instead of N copies**
- Since you are compiling each copy separately, you can generate special code depending on that type that could work just for a double or Boolean, for example.
- Can generate efficient code based on the fact if its a constant (Template)
- Creates a variable so we don't know what the size is! (Generic) - much slower
- Generic code - how do we do an assignment?
- Java and ML do generics and run pretty efficiently. How do they do assignments?
- If you were implementing generics and had to compile one copy, how would you do it?
- Each type would have a method called assign that takes two pointers (source and destination)
- To do assignment, do assign method.
- **This is too slow!**

- Java should be within 20%-40% of C's speed
- **Most of the objects are reference types!**
- There is no sizeof operator in Java!
- Whenever you do an assignment of an Object in Java, do you copy all the objects values?
- NO! You just copy a pointer.
- Every Object in Java is represented by a pointer to the Object.
- We just copy a pointer and on x86-64, we just use movq no matter what.
- Idea of using generics is for efficiency reasons
- Closely coupled to representing all objects as a pointer
- All objects smell the same - single word quantities and you just pass the words around
- Generics tend to be used for newer languages where you try to be more abstract, high-level, and easier to write garbage collectors for

Subtypes & generics

- Pointer assignment here in example
- Not copying everything in the list, rather we have a pointer to a list of objects!
- We can call the add method that is appending one item to our list of objects
- lo and ls are the same Object with different names
- This is a problem!
- get() grabs the most recently added Object and it will actually be a Thread instead of String
- By a process of elimination (Sherlock Holme's rule hahaha), the problem is the line:
- Line <Object> lo = ls;
- Assumes List<String> is a subtype of List<Object>
- This is bad! It subverts type checking
- Try it out in C++ and see what happens!

Given this idea that we get into trouble, let's try to write some more generic code.

- Let's try to write something that prints out every item in a collection.
- System is a standard class, out is stream available to everybody, println() prints everything and adds a new line.
- This combination is so common that people now write it in a more compact way.

Suppose we have a method that only works on Strings

- Instead of Object o, we have String s here.
- There is a problem with this code.
- Suppose we declare a subtype of String called BlueString
- Whenever you do a method call in Java, there is an implicit assignment from the parameter to the argument
- Assign a collection of BlueString to a collection of String

- **We cannot do things like this because it would subvert type checking!**
- So how do we fix it?
- We need a constrained type variable! Either a String or a subtype of String
- Use extends to create a **bounded wildcard**
- ? is a wildcard that is constrained to String
- Once you start down this path, you have to be careful
- These features were added to Java in looking at how people wrote code and what kind of error messages they would get.
- Supplying reasonable code that the language would NOT allow us to do!
- Language designers allowed bounded wildcards because of this
- Try a little bit more code and see if bounded wildcards are enough for it
- Probably NOT!

Suppose we don't simply want to print out an Object, but rather convert one collection from one format to another.

- Suppose we have local variables and we try to do convert(a, c), the compiler will yell at us.
- **BlueString does NOT have to be the same as c**
- **Array of BlueString is NOT a subtype of Array of String**

This gets used all the time in Java!

- This set of rules is very complicated and they go on and on for pages
- No one in this room knows them all either
- If you look at C++ type rules, how many of you would know all these rules.
- **There is a school of thought that says all this stuff of parametric polymorphism is too complicated**
- Keep it simpler!
- Mention a language to you that uses a much simpler approach (Python!)

Python & subtypes?

- Python doesn't have generics - so how can you tell if one type is a subtype of another
- Python doesn't have a strong distinction between compile time types and runtime types
- They are all kind of the same thing in Python
- t is going to be a data structure and u will be a data structure, but these describe the types rather than giving the layout of the Object
- Create an Object of type t and create another Object of type u
- Instead of talking about the types, we are talking about the values
- If it works, it is good enough.
- If it doesn't, then it doesn't work.
- This is known as **Duck typing**

Runtime philosophy but the advantage of Duck typing is that it is super easy to explain

- A lot of people don't even bother with the exception handlers - you need to catch the exception!
- Keep the code simple and create classes and subclasses
- If you want to create something that acts like an existing Duck, create another class with all the same methods
- What we are observing in modern programming language design is a long-term struggle between compile time type checking (Java, ML, theory) and the hackers who just want to write flexible, easy-to-understand code
- Theorists come up with more reliable software
- For your project, you have to decide which part of the divide you live on and this decision is an important one
- Can't really have flexibility and all that in this case.

Java - history

- Go back 23 years (I wasn't even born yet!)
- A bunch of engineers in a large computer company that were trying to think ahead to the future
- Take this technology and make it work in the Internet of Things
- If you assume you have work stations and servers, there is something else you can assume.
- **The reliability constraints were more important!**
- Code needed to be more reliable than CS 31 student code that dumps core 20% of the time
- Machine code runs fast but there would be trouble fitting it into these tiny machines
- C is so 1970s i.e. NOT object-oriented and NOT flexible enough
- They wanted something more flexible because they foresaw a future where there would be slight nuances with modules that plug into each other
- **C is good when you know what you want, NOT so good for classes, subclasses, and special cases**
- In the embedded world, people have MIPS, x86-64, etc.
- They wanted something that was architecture-agnostic
- C++ gave them flexibility but it didn't handle any of these other things very well
- Classes and subclasses worked very well but all of this other stuff didn't work very well
- Every time we use C++ on a different type, it compiles a different copy
- C++ Is a disaster from size of program POV
- New problem is that the C++ is way too complicated!
- Almost nobody knows all the C++ rules
- Professor at Texas A&M and Cornell know, but that's about it!
- C+- had problems with reliability since programs dump core, and the architecture problem hadn't really gone away.

- They went back to the drawing board and did what every good Computer Scientist would do.
- **Drive down the freeway to your neighbor and steal their ideas!**
- Sun Microsystem Lab -> Xerox PARC
- SmallTalk -> Object Oriented in a more flexible way than C++
- SmallTalk uses something close to generics (closer to what Python dos)
- C++ had a complicated system that kept crashing and bloating
- bytecodes - took program and translated it into a machine-independent set of bytecodes and threw away irrelevant details
- Runtime checking like *p a[i]
- Your program will run on any of these guys because you have an interpreter for it
- Turn these ideas into great papers that nobody bought
- The Sun Microsystem guys would steal these ideas and made them cheap.
- Some things they hated about SmallTalk though.
- Syntax is janky (ask Professor Kay)
- SmallTalk interpreter kept byte code to itself - did NOT expose to the user
- Secret byte codes
- Download a new program to the Internet toaster - send byte codes rather than machine code
- Scrapped the bad ideas and built a successor to C+-, which they called **Oak**
- Their manager said they were doing great work but they needed money, so the project got cancelled
- Engineers wanted to write it up and publish some paper.
- Back to the servers for you, what the engineers did as part of their writeup for their system was to come up with this technology that they built to create a Browser
- Roughly around 1995
- Browser had been invented and the only one in widespread use was **Mosaic (UIUC)** (Andreessen…)
- C++
- Notoriously buggy if you go to a wrong website, you get a core dump
- Instead of writing it in C++, let's do it in Java (Oak was trademarked)

From Day 1, Java had thread support that actually worked unlike the thread we see in C/C++ compilers that sort of works but it is a pain to use.

Java types

- primitive types
- Java nails down type sizes
- Java doesn't care if you have a weird machine with a long word, they want **long** to be 64 bits
- Handling overflow
- In C, it is undefined

- In Java, if you have overflow, it wraps around
- Trade performance for portability
- reference types
- You cannot see the pointers, but everything is referred to indirectly
- Including arrays
- In Java, you create reference types using new
- **You cannot allocate an array on the stack in Java, it is always on the heap**
- The size of an array is dynamically chosen when allocated
- However, you cannot change the size of an array after!
- Java doesn't have a delete operator for Objects, which is wonderful! (one less thing to do)
- What's the catch?
- Java doesn't have this problem. If nothing in the rest of your program points to an Object, the Java garbage collector will figure it out.
- You can still have a memory leak in Java even with a garbage collector
- How would you cause trouble in Java?
- Declare a static variable and somewhere in your program, do x = new int[1000000000000000000];
- Never use x again!
- If no other code segment uses x, the garbage collector **might** be able to figure out that x is garbage, but not guaranteed
- static global variables are so 1960s, don't do it!

Single inheritance

- Java is a single inheritance language
- **Only one parent in Java, which simplifies a lot of rules**
- Simple and elegant, but NOT flexible enough
- Use **interfaces** instead
- BlueString inherits money from String
- All of these methods are available for BlueString to call
- BlueString inherits a debt with APIs but NO implementation
- Inherit positive code from your parent and you implement negative code supplied by your interfaces
- There is also some sort of programming reasons to prefer this way
- Some language designers say that **extends** was a mistake
- You should never inherit money, you should only inherit obligations.
- It is better to start off poor and gain wealth on your own rather than living off your parents' code
- Inherited an apartment building in disrepair that people won't pay up.

W 4 Dis 2-3-17
Midterm - everything up to HW 3

- If you get started today on HW 3, it won't take too long.
- Bring everything

- Printouts of homework and solutions
- Notes from classes, discussion sections
- Class textbook
- Important to go over all class notes
- Go over HW problems again
- Everything covered until Midterm is fair game

Sample Midterm
- Similar format to the exam
- Whatever chapters Eggert covered in class will be on the midterm

Today - A lot to cover
Java development
- Eclipse, IntelliJ, etc.
- You could also use your favorite text editor
- Compile from terminal with javac
- javac main.java
- java -ea Main
- Test everything on SEASNet!

Object Concept in Java
- An object is an encapsulation of data.
- An object is like a struct in C, which is an encapsulation of data, or a class that has abstract data types and fields
- Objects can inherit from other classes through inheritance
- An abstract data type is implemented via a class
- Instantiate based on that class

Encapsulation and Information Hiding
- Procedural - C programming
- ADT - C++ programming

Type and Interface
- Object is Account and you have a couple of methods
- Account a = new Account()
- A couple of parameters and define according to the Constructor
- implicate one of these methods with node at a and pointers.
- Say you have all these different objects. You will have all these parameters and divide your code and make it more efficient
- Everything is encapsulated and neatly divided.

Generalization and Specialization
- Java has either **Inheritance or Interface**
- You can have a shape and a subtype i.e. Line, Rectangle, etc.
- Shape defines methods draw() and resize()

- Circle, Line, and Rectangle define their own implementations of draw() and resize()

Object Oriented Design Example - Point2d

- We have a point with functions negate and toPolar
- Convert from one point to another point in Java

//abstract class

- negate and toPolar
- abstract data types use the abstract keyword to indicate they aren't intended to be used
- Everything that works with Point will work with PolarPoint because PolarPoint is a subtype of Point
- Modularity involved here.

Subtyping

- interface vs implements
- If we have extends, a class in Java can only extend to one class (single level inheritance)
- interface lets you outline the function definition but cannot borrow exact statistics rom an interface.

Class Specialization Example

- Keep hierarchy in mind when doing something like this.

Interface

- A collection of method declarations
- Class-like concept
- No variable declarations or method bodies

Interface vs. Abstract Class

Interface

- This is sort of the hack that Java lets you do

Abstract class

- Java doesn't support multiple inheritance

implements is interface and extends is for abstract class!

Memory Model & Object Oriented Style

- Analogous to pointers and references

Parameter Passing

- In Java, everything is passed in by value

Homework 3 - Background

- Much easier compared to the previous 2 homework
- Concurrency issue
- You have an array and the only thing you can do is swap
- Check if it is larger or smaller than the maximum value
- Keep all these integers in the array between 0 and maxval (e.g. 127)
- Otherwise, you just increment j by 1 or decrement i by 1.
- You can split the work up to different threads
- Sum of the integers should remain constant

Problems with Concurrency

- static - variable that is the same variable wherever you use it

synchronized keyword

- Tells Java that you can "only let one thread in here at a time"
- Makes each operation atomic and this is how you keep things safe
- Blocking methods so you cannot use **any of the other synchronized** methods for other threads
- You can only work with one thread per instance of the class, so you are blocking access to the class.

volatile keyword

- This variable's value will be modified by different threads
- Value of thread will never be cached thread-locally
- Primitive variables may be declared as volatile

volatile is not blocking

- Unlike a synchronized block, we will NOT hold on to any lock
- volatile variable can be null

volatile Example

- Run code and make changes to MY_INT
- If we don't make MY_INT volatile, we cannot guarantee change is detected

AtomicIntegerArray

- This array will be an atomic integer array
- Everything happens in a single step
- get(i)
- set(i)

Example Code - overview

- Parse arguments and write some input here

For each thread - you will do each loop that swaps two random indices, and join at the end

- Test and see the output array

Input on command line will be number of threads

- Initialize state and the function we implement is called dowork
- Check if the sum is equal to whatever the sum was in the beginning
- Run this hw and get different timings on different machines
- If you are expecting a better time and it doesn't happen, it is fine
- Synchronized has a lot of blocks and mutexes, but sometimes, it is a natural thing

Make sure you don't have a bug or something like that.

- Run it on the same SEASNet machine to make sure it is consistent.

SwapTest.java

- You could copy and add run() to your code for testing purposes.

Sample of midterm related questions

0. I do think that it is reasonable to throw multiple exceptions as long as you clearly signify the type of error.

2a. s = int -> int = <fun>
x = int
b.

W 4 Dis (Theresa) 2-3-17

- Midterm from 2008
- Definitely review the sample midterm so you know the format
- Don't bother trying to memorize questions, it won't help you at all!
- TAs are scrambling to make copies because he prints it so late
- Take what you learn and apply it to real world problems

Java - won't be using fancy function calls or anything

- If you don't know Java, you need a crash course for the basics
- This hw is not as technically hard as the last two
- Research and experimental

Java Memory Model handles any access to shared memory in the cache

- This allows you to optimize a lot of things but it can slow down some things as well

Midterm

- Bring extra scratch paper!

0. "Ireland has leprechauns galore" has meaning, but English has a very flexible grammatical structure.

- I could have said “Ireland has a lot of leprechauns”

Go back to hw1 and hw2 and try looking at the types of each function

- Read that carefully because that will be tested for sure!

Any functions you have coded are fair game in the exam!

W 5 T Lec 2-7-17
Recap

- Object hierarchy
- Interfaces vs classes

Abstract classes

- Combinations of classes and interfaces
- Can define a method
- Think of abstract class as an ordinary class bolted to an interface
- You CANNOT invoke new on an interface because there are no methods that can run there.
- We can see this idea in use in the Java Standard Library
- This is a class on programming languages but if you look at the Java library it is fully extensive.
- Collections framework is an organized group of classes dealing with classes in many contexts.
- This grows out of control if you use all the diagrams

- Lists are ordered, whereas Sets are unordered.
- Lists also allow duplicates, while Sets do NOT!
- HashSet is the first class where we can say “new”
- Built out of a HashTable
- You also have LinkedHashSets out of it
- TreeSet
- Easier to get sorted items out of trees
- Key out of a balanced tree to see where to go
- ArrayList vs Vector
- Vector is **synchronized**, but ArrayList is NOT
- Useful in multithreaded applications for safety involving data accesses

Key factors
Clone-able: copies of object
Serializable: turns into a byte string so others can get copies
Random Access:: Gets ith element easily

- If you are to be an experienced Java programmer, you should know this!
- This is the next level of abstraction above classes and interfaces
- You can think of this framework as being this whole thing and it is a way of thinking of how to connect classes and interfaces to each other in a coherent way

- How do you hook things together?

Q. What does the implemented part of AbstractCollection do?
A. Let's say you had a Set membership operation and you don't have an implementation for it. Is X an element of Y? Suppose you wanted to test if X is a subset of Y and determine if it is an element of the 2nd set.

- Abstract Set has a code for subset and that subset code calls elementof, which is just an API
- The code for subset is generic but will work for whatever the implementation is.
- If you have some sort of class down here that doesn't care about efficiency, just rely on parent code.
- **Abstract classes let you bootstrap higher-level operations!**

Object

- **Anything part of Object class was something Java designers decided was so important that every Object should have these elements**
- new Object() creates an object and a pointer to the Object
- Most boring Object imaginable
- You can use this Object as a test case for you and build the most boring possible Object!
- Mostly used this in test code that should work on anything!
- Use it as a placeholder and sometimes, it is helpful to have a unique Object that is NOT a null value

Q. Is there no default constructor in Object?
A. You can define all constructors as private and no one can invoke them directly.

- In that case, you have to use static methods that you supply!

boolean takes one argument and you are comparing this argument with the Object this class is being declared on

- Defaults to "==": the built-in implementation of the equals method
- Which is reference comparisons!
- Can override this and it is useful for **string comparison**
- Can't do this in general for things like cyclic data structures

Companion method to equals

- hashCode - used for hash tables
- If o1.equals(o2) this implies o1.hashCode() == o2.hashCode

If your class overrides equals, then it should override hashCode as well!

- Defaults to (object low order 32 bits)
- Objects could be aligned so the bottom 4 bits might be 0
- You want to find the most random part of the address and typically, it is a little bit away
- How could you break things by using these methods.

- Define a subclass and define some sort of methods for equals and hashCode that cause trouble
- Have hashCode always return 0.
- **Performance will suck ass!**
- Can you make the code stop working?
- Don't always return right values for hashCode
- All things with hashCode won't run correctly!
- The penalty for violating this rule is that your program doesn't work!
- Won't be caught by a compiler because it is a runtime property that the compiler won't check!
- Compiler is NOT good enough to enforce this!
- Check statically the things we can check, and the rest we lecture about!

final - opposite of abstract
- abstract means the class does NOT really exist!
- Need to create a new subclass

- Old project - biologist was upset that the code was slow, and didn't trust parallel code
- He decided to put final and why would that speed up his code?
- Once you see something is final, you don't have to worry if it isn't subclassed
- If you see a call to a final method, you can look at the method and replace the call to the method with the contents of it.
- If it is a short method, it will actually be faster.
- **final allows inline and optimization!**
- With compile-time type checking, you get some speed back!

In Java, objects are always represented by pointers
- Other methods might give useful information about a class
- Instance of a more general theme in Programming Languages: **reflection**
- If you have a program, there is a phrase called **fully reflective**
- Use the mirror to find any aspect of the program!
- **Java is NOT fully reflective**
- You cannot find every detail and we don't have that much information
- Why would you want a getClass method?
- There are some limited things you can do with getClass and is it worth adding this primitive to the language?
- Argument here is that you want Java to be easy to **debug**!
- C++ is a pain in the ass to debug!
- C++ is NOT intended to be easy to debug!
- Intended to be a low-level language and if you cannot debug it, too bad!
- Is there a getClass method in C++?
- NO!

toString: just returns this!

- Simplest method imaginable

finalize(): Called whenever the Object is discovered to be garbage and the garbage collector is going to collect it.

- As a last act, it calls the object's finalize() method
- Default is to do nothing!
- You can override finalize() to do cleanup actions associated to the destruction of this object
- Are you going to tidy up the Object?
- Why are you going to do this?
- Reason: If you are dealing with a primitive from an ancient OS that you have to call open() and close()
- NOT automatically managed for you and you have to keep track of which file descriptor you are using.
- finalize() can be communicated to that old fashioned piece of software.
- Done with database connection
- Costs an arm and a leg to work with that database
- If my database connection is such that no one cares about it anymore, finalize can be crucial because it deallocates system resources
- **throws** is for exception handling
- Default is that it doesn't throw any exception
- Garbage collector has to be aware of exception handling to occur
- Every time you run new, an exception can happen, so does every "new" throw an exception?

clone(): will this always work?

- Shallow copy (just the top-level by default), not a deep copy!

Cannot override getClass()

- If you can override getClass, you could return a class that is not correct and this would break reflection!
- Reliability!
- Best thought of as generic

Suppose type is NOT simply x, but it can be a Collection

<u>erasure</u> of a generic: Throw this away and substitute the erasure for this type

Java interpreter needs to keep track of erasures of type, NOT the original!

- record only erasures of types, not the original!
- We want the interpreter to be fast and efficient
- When we do a cast from one type to another type, we want that runtime check to be fast.
- As long as we deal with erasures, we can do that
- **This is for efficiency!**

Our laptops can do only do about 8 GFLOPS!
Super computers are 10 million times faster!

GFLOPS / W is where a lot of action is in this type of computing!

We are kind of at a loss to how to write code for machines like this that are easy to write/develop, etc.

- Functional programming should be a way to go for machines like this, but people running machines like this do NOT trust functional programming
- They say it is too theoretical!
- Eggert hopes that some of us in the room are smart enough to write this program!
- This kind of machine won't seem outlandish 20-30 years from now.

In 1980-1990, the world's supercomputers would have 8 CPUs, so what kind of architecture did they have back then?

- SMP architecture (Symmetric multi processing): You just have each CPU which has its own instruction pointer
- Each CPU is the same and runs its own set of instructions
- MIMD (Multiple instruction multiple data)
- SIMD (Single instruction multiple data)

Thread model (Java, C++, C, …)

- You have a description of code you plan to run in the future but it isn't running yet.
- If you are in a busy system, other threads will be given less CPU
- How do threads stay out of trouble?
- synchronized: Method should have exclusive access to the object is is operating on
- Any other synchronized method will make on this
- If you have a mixture of elements accessed, you lose!

They put in synchronized so that you could make toasters reliable!

- This didn't quite work because it was slow as hell!
- Is it because of the lock and unlock calls?
- **Inserts bottlenecks into the code**
- The original Java if you look at the class library, it attempted to address this by declaring a bunch of standard classes to be synchronized
- One doesn't modify the hash table and everything was synchronized
- Although this works if your application doesn't do much, it doesn't help if you have high-performance applications on the server side.

If you are waiting, you need to others to access logic while you are asleep

- These other things are spin locks and if others are trying to get the lock, they will just spin!

Q. Why do we have both of these?
A. What are each going to do? Try to get a lock on the object! Why bother waking up three guys when one of them would have sufficed.

Java Memory Model

- Test will be based on things for hw for

W 5 Dis (Tomer) 2-10-17

- Use prolog and check the results on SEASNet
- Be sure to verify things on SEASNet because that is what you are going to test
- At least 3 or 4 people did that and they got bad grades

Running Prolog code

- http://www.gprolog.org/#download
- Prolog is a language implemented in certain ways with a lot of overlap
- Use the one specifically on SEASNet for this class

Prolog command line
consult('filename')

To Exit: CTRL + D

Prolog - not really a program but more like a database

- Populate a database with some kinds of facts
- These are all kinds of facts and these can create extensions using rules and create more complex ideas
- Query this database with yours and get all kinds of interesting things
- Facts vs Rules

What is Prolog?

- Declare a set of **facts**
- Declare a set of **rules**
- We can ask all kinds of queries on these things.

Prolog Code

- Here, the facts are that alice is a person and bob is also a person
- Every time you have a capital letter that starts your variable name, that means something different.
- Capital letter is a variable
- **Variable - start with Capital letters**
- Person is a predicate - an uninterpreted function that returns a boolean

Prolog Program

- **Left hand happens if right hand side is true!**

Rules
- Based on this hypothesis, this is my conclusion

Example
- commas mean AND in Prolog
- A dot at the end signifies end of the statement
- semicolons mean OR in Prolog

likes(john, susie) :- likes(john, mary). /* John likes Susie if John likes Mary *)

How to read this:
<First arg> <function> <Second arg>

X left of Y if Y is right of X.

X does not like Y if X hates Y
- You cannot have NOT on the LHS
- You can only have one statement and you can only define one without anything else

- Some people said some versions have issues with Prolog

If we had a boys category, you do , Boy, X
- Once you have a variable, if you use it once, it sticks for that meaning

likes(John, Y)
dislikes(John, Y)
- Different Y values per line

More Examples
- British family tree
- parent(child, parent).
- mother(X, Y): Y is the mother of X

male(george).
male(james2).
male(charles2).
male(charles1).
male(james1).

female(sophia).
female(catherine).
female(elizabeth).

parent(charles1, james1).
parent(elizabeth, james1).
parent(catherine, charles1).
parent(charles2, charles1).
parent(james2, charles1).
parent(sophia, elizabeth).
parent(george, sophia).

mother(X, Y) :- female(Y), parent(X, Y).

- If you want to see all possible answers, use ";"
- For debugging
- If you just want to see an answer, do "yes"
- Stops the program's check

- Check if same parents

sibling(X, Y) :- parent(X, Z), parent(Y, Z), X \= Y.
Q. How does parent know what Z is when it is passed in?
A. Z tries to see all the parent relationships where Z match to the same kind of constant and it tries to infer that.

Q. What does the question mark mean?
A. It will not go on forever and it will show you all the possible solutions?

- If you have implemented your solution wrong, you could have an infinite number of solutions

Overview of arithmetic
x < y is the same as X < Y.

Arithmetic - 'is'
?- X is 6*2 .

?- R is mod(7, 2).

Q. is can be used rather than = for integer assignment
A. You can use = too.
- Since X is a variable, you wipe out old values upon new assignments.
- Unification - mathematical word

X =\= Y.
- Arithmetic NOT equal sign

You can only use X if you have some kind of computation over here.

Arithmetic - 'is' usage

- If we are free to use variables on the RHS, the variable must have been already assigned or else you will get an error
- Expression should be on the right hand side of ‘is’, otherwise you get an error
- We have some things here and the other way is just reversed directional logic.

Equality Operators
- When we say likes(X, prolog) == likes(X, prolog) -> yes

Equality - unification
- Checks all possible cases and binds X to some kind of atom
- Tries to bind values
- = is similar to ==, but term1 and term2 are binding. Otherwise, it won’t be equal
- _ is a wildcard in Prolog just like in OCaml
- X and Y, we don’t care what they are, we can always find a situation when X and Y are the same
- In the end, you can just define different facts

Warm-up
% talk about lists in prolog
% [1, 2, 3] a list in prolog

% scalarMult(3, [2, 7, 4], Result) —> Result will be [6, 21, 12]
% Lists are written [Head | Tail] similar to ocaml
% Head is just one element
% Tail is the tail of the list (a list by itself)
% [X] —> only 1 element in list

```
scalarMult(Scalar, [H | T], [Result]) :- Result is X*Scalar.
scalarMult(Scalar, [H | T], [Hres | Tres]) :- Hres is Scalar * H
                                                    scalarMult(Scalar…;
```

- Tries statements iteratively (order matters so be careful)

% (dot [2, 5, 6], [3, 4, 1], Result)
% should yield
% Result = 2 * 3 + 5 * 4 + 1 * 6 = 32

```
dot([H1 | T1], [H2 | T2], Result) :-
        dot(T1, T2, TResult),
        Temp is H1*H2,
        Result is Temp+TResult.
```

List Manipulations
% append_([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3]).

```
% we will get the response yes.
% append_([a, b, c], [1, 2, 3], [1, 2, 3, a, b, c]).
% we will get the repsonse no.

% [X, Y] —> list has only 2 elements

append_([], L2, L2).
append_([H | T1], L2, [H | T3]) :-
        append(T1, L2, T3).
```

- Don't need the 2nd rule because awe don't have to check if the 2nd list is empty

```
append_([1, 2, 3], [4, 5, 6], [1, 2, 3, 4, 5, 6]). // Works

% prefix_(X, [a, b, c, d])
% x = [], [a], [a, b], …

% prefix_([1], [1. 2, 3])
% prefix_(X, [1, 2, 3])

prefix_(P, L) :- append_(P, _, L).

suffix_(S, L) :- append_(_, S, L).

% Take suffix of prefix, or prefix of suffix (doesn't matter, remember CS 181)
sublist_(SubL, L) :- prefix(SubL, L), suffix(SubL, L). % Problem, are L's the same in the
args?

sublist_(SubL, L) :- suffix(Temp, L), prefix(SubL, Temp).

% member(1, [2, 1, 3]) —> yes
% member(4, [2, 1, 3]) —> no

member(E, [E]).
member(E, [H | T]) :- member(E, T).
```

Compress

You are going to have repeated sequences of some patterns and you want to ignore them.

Q. Is == name equivalence while = is structural equivalence?
A.

W 5 Dis (Theresa) 2-10-17

- Average should be 50%
- Easier than last year's midterm

Logic-based programming

- Used in AI/machine learning
- Prolog has a very simple representation of straight up logic.
- Prolog facts are something we represent as true and it's anything that starts with a lower case and ends with a period
- We want to see relations!
- Arguments in Prolog are arguments in a relationship
- You can have likes(Alice, Bob): Alice likes Bob
- Ask Prolog whether a relation or not is true
- Return true if Alice does like Bob!
- **You cannot code in the Prolog shell!**
- **All facts must be put into a file to be loaded into Prolog shell**
- Rules are how you put all the facts together to come to some sort of conclusion.
- Free to use library functions in this assignment
- Anytime you call one of these, it binds one value of a list using the head and tail thing
- Choose all possible substrings to give you a full string and it keeps incrementing the head to include more of the list
- Keeps going until the user accepts one of your answers

HW 4

- Feed 1's and 0's as a list
- There is a comma between every single element (differs from OCaml)
- List processing in Prolog is not too different from OCaml
- Do NOT install SWI-Prolog?
- Use gnuprolog for Mac!

Think of it like a tree: goes from left to right and tries any subsequent functions

- If you go past the cut (!), don't backtrack anymore!

Signal Morse

- Changes to corresponding dots, dashes, #, etc.

- Using what we know about Prolog, write me a rule that binds M to a possible list of D's and M's

- a | b will concatenate lists

W 6 T Lec 2-14-17

- Synchronization in Java
- Methods you can implement yourself but more convenient
- Exchanger: Pass in value and get a different value
- Atomic in the sense that it is always reliable in terms of swapping
- Sets up rendezvous points!

- CountDownLatch: Parallel computation divides problem into subproblems.
- Pause and then let them exchange information
- Divide all of the area of the U.S. into a grid and basically split up the cores
- Set up a region of computation where the threads have stopped
- All these threads are doing computations
- Each thread is a horse that wanders around and then there is a gate.
- When they are in the gate, they cannot do anything, but then you press the button and then they are off!
- CyclicBarrierLatch: Countdown latch that can be repeated
- Horses are going into a circle and each time they pass the Grandstand, they go into a coordination area
- “embarrassing parallelism” is the best kind of parallelism
- Attempt to minimize coordination
- Try to have weather simulations where all we worry about is a state and we don’t care about other states
- All of these are built on top of the o.wait() and o.notify() method
- Overhead to these methods!
- NOT the pain of getting the spin lock and all that but when you are accessing objects, you are creating a bottleneck.
- If any one thread is in the middle of figuring out a wait() operation, the other threads are twiddling their thumbs and don’t know what to do.
- Speedup isn’t as much as you like

Classic Example

- We bought a machine with lots of cores but we don’t get that much out of it
- Do safe memory access without locking

Java Memory Model

- Provide rules for the road to provide unsafe code and it still works
- We want the performance (no bottlenecks) but we want our code to still actually work
- Problems this model has to face:
- Fast Java system, your compiler will try to generate the fastest possible code
- Optimizing the compiler
- Reorder instructions
- Cache variables into registers
- Compilers will make optimizations whenever they see redundancies or constant value calculations
- **Machine code can also get reordered as long as it doesn’t affect the end result**
- These optimizations are good because they go faster, but if you execute this set of code in parallel with other threads, you will get unexpected results!
- Data member z could get updated, but the object o would be not

- Inconsistent!
- What can be done?
- We can do these optimizations but we want it to be true as often as possible
- Think of each thread as an object moving through space
- Each of these can be considered memory locations

Minkowski diagram

- There is an explicit future and past, but we need to figure out **elsewhere**
- We could have two uses of the same variable but we don't know exactly what the issue is

Causal relationships?

- Affected by memory locations
- When you break the rules of causality, there is no physics stopping you, and your program will have undefined behavior
- Bug you might not know about could pop up much later on

- Can you interchange normal loads and normal stores in the machine code that you generate
- This assumes the code you are running is single-threaded

Single threaded - program will work either way in a single threaded implementation, so the compiler can reorder these loads and stores

- Some particular pieces of code cannot be reordered because the order will matter in some cases.

volatile int n;

- When the program accesses this variable, the compiler cannot optimize the access!
- **Must look at memory and NOT the cache!**
- volatile does NOT handle reordering instructions

How does Memory Model interact with the standard sort of synchronized methods?

- Category B is the trickiest one and is the most problematic when you try to do code reorganization
- This table is NOT obvious and it was NOT achieved by someone looking at the problem and saying we will use this table.
- Achieved more by trial and error
- People wrote compilers/code and they encountered buggy code with race conditions
- This model only sort of appeared after a lot of trial and error

- The value for x has NOT been set yet, and we want every piece of storage to be initialized to 0 or the null pointer

- In practice, compilers don't bother to clear storage if it will be initialized by the constructor
- For efficiency reasons
- Constructors cannot share their self pointers with other threads
- The JMM says if you try to write code like this, then you have undefined behavior

Q. Do you want to put a static variable and point it there?
A. You want new to be a fresh object that no one else is using. You naturally assume that, so why would you use a static variable?

- People wanted to add this restriction because it was a good design practice

Add features to Java before we knew anything about memory models!

- JMM (and even worse in C/C++) has the issue of retrofitting bad practices in older languages with a lot of legacy code

Global interpreter lock in Python

- Not intended to be high performance in threads
- Python is trying to branch away with this to optimize speed

JMM wants good performance + multithreading

Names & bindings

- binding - association between name and value
- When talking about bindings, you want to know which value are bound
- When is the binding established?

We bind types as well, and this is when we look at programs in the first place

- Allocate storage to hold the thing and you can then figure out how to organize the stored items

Suppose I go into SEASNet, become root, and I go and change this file.

- Cross out the long
- Why won't our programs work?
- They assume it is an 8-byte variable
- This sort of decision is a decision that was made before the compiler ran.
- Need to know what those times are in order to understand how your program works.

Convoluted program deciphering

- Each structure defines its own little namespace
- This f has to be preceded by an arrow and you can look at the type of this expression so there is no ambiguity
- From context, you can figure out which of the f's was intended

- Rules for labels in C say there is a single scope for the entire function

Primitive namespaces

- Hardwired into the compiler and you don't have to do anything special
- This is overkill

You don't have a single place to find all the code, so you have to navigate around, but a bit more flexible than Java

- As long as runtime things that get checked don't change, you don't have to rebuild your entire code.
- Suppose you have two different C modules
- These modules allow you to declare a global variable

Please read Prolog chapters for next time

W 6 R Lec 2-16-17
Logic Programming

- Basic idea is to compare these styles of programming:
- Imperative programming (C++, Java)
- Loops and assignment statements
- Functional programming (OCaml, …)
- Take away loops and assignments
- Recursion, functions, calls
- **Logic programming: Least used style!**
- Take away functions and calls
- **Keep recursion**
- predicates and assertions
- queries (recursion)
- Sort of like Boolean expressions but top-level

Algorithm = Logic + Control

- This equation is true for C++, Java, and functional languages
- The idea is when you write a for loop, there is sort of an algorithm that is there.
- Single piece of code that implements the algorithm
- Really two things going on here
- What functionality do you want?
- In conventional programming, you combine logic and control together into one program
- **With logic programming, one of the goals is to specify what control to use**
- Look in your program and control cannot affect what answers you get since you get the same T/F answers whether true or false.

Take a big problem and split it up into smaller subproblems

Sorting

- Quicksort, heap sort, bubble sort, insertion sort, etc.
- Don't want to think of any of those algorithms but you want to verify if the answer is right
- **No functions in Prolog, we have to write predicates**
- **Initial caps for variables, initial lowercase for predicates!**
- **The way you write "if" in Prolog is :-**
- **Feed its output as S and input as L**
- Every element in S must be in L, and vice versa
- Same number of elements and be the same elements
- **S has to be a permutation of L**
- All elements in S have to be in non-decreasing order
- **For all L and all S, if perm and sorted is true, then sort is true!**

Base cases of recursion come first

- Understand the easy stuff first

Names are important when writing programs

- Non-decreasing is better than ltfe

When stuck, it is recommended to draw pictures of what is going on

- If there is no H in the second list, then there is absolutely no way it is a permutation
- Permutation is defined recursively in terms of itself, and that is the key way this code is going to work
- Take this picture and convert it into code

Prolog

- Can you append P to [H | S] and see what happens?
- Neither O(n log n) nor O(n^2)
- Keep trying different permutations of L and the guy generating the permutation goes in random order
- No organized way of picking the sorting option
- The worst case: **O(N!)**
- In Prolog, it is possible to write logic that is as fast merge sort, but it is more complicated!

Atoms are indivisible - equal to only themselves

- Individual units of computation that you cannot divide up and look inside them
- Insist on being individualists

Numbers are like atoms - equal only to themselves

- You can compute with them

Variables (sometimes called logical variables)

- Have an initial capital letter or underscore
- Has as many letters and underscores as you want
- As you keep computing successfully, the variable never changes
- Bound on success and never changes
- Keep computing complicated expression and X inside your other expression cannot be changed
- With a logical variable, you can do one thing you cannot do in functional programming
- It can fail!
- You made the wrong decision and you have to backtrack in this case.
- When that happens, the variable becomes unbound again
- As you sort of use the variables and instantiate them to different values

In Prolog, you cannot do this:

- I++
- I = I + 1

Logically, Prolog is correct! It thinks in terms of the mathematical definition!

- To tell whether two atoms are equal, you have to see if they have the exact same spelling
- To tell if two numbers are equal, make sure the numerical values are the same
- If two variables are equal, look at the variables they are bound to
- If one variable is bound and the other isn't, you have to bind the unbound one
- ?- X = Y
- Bind X to Y even if we don't remember the values
- The two unbound variables have been coalesced into a single bound variable
- Compare function symbol (f/n) and the arity, otherwise, we fail
- Combine any set of Prolog terms

Prolog program structure

- Terms don't have to be this simple, it can be more complicated
- _ is a don't care variable analogous to OCaml

sorted([_]) : if you see one occurrence of a variable in a clause, probably should change it to an underscore

- I don't care what it is!
- This style rule is so common that GNU Prolog actually warns you

Rules

- Conditional expression of the form: consequent :- antecedent

- Consequent is a single term
- Antecedent is one or more terms separated by commas

Queries

- The way you see these at the top-level, you type in a question which will run and eventually give you an answer

Prolog can give up by saying “NO”, which means I couldn’t do it

The scope of a logical variable is equal to the clause that it appears in

- Clause is the basic unit of a Prolog program
- Govern the scope of the variables
- The scope of an identifier is just in the clause that it occurs in

Prolog is goal-oriented

- Computation is based on queries
- Query establishes a list of goals
- I have N things I want you prove and we always maintain a goal list of things trying to prove.
- Always looks at first goal in the list.
- Eagerly going depth-first, left-to-right
- If you are into AI, this is a really stupid way to do things
- Simple, easy to explain, and programmable, which is why we do it in Prolog

Built-in method of false:

```
?- fail
no
```

- Empty code will fail
- ?- fail.
- Program predicate with typos will fail because it is a missed predicate, and every Prolog interpreter has an error message in it.
- If you call a predicate with no facts associated with it, they will ask if you intended to do that.
- Even though you have no source code, there isn’t a typo!

From Prolog’s point of view, since we never get to the button, we can remove it!

- These are kind of silly little predicates, so let’s do something more interesting!

```
member(X, [X | _ ]).
```

; tells the interpreter to fail and try another solution

- append is O(n) and we call it n times, so we get O(n^2) algorithm

- Can we improve the performance of reverse?
- Have an auxiliary function!

W 6 Dis (Tomer) 2-17-17

```
index( [H | T], H, 0).
index( [H | T], X, Idx) :- index(T, X, idx-1)
```

shift_left and shift_right could appear in the final exam

HW 4: Morse code data recovery

- The tricky part is that audio recordings are of poor quality.
- If there are four 0s, it will translate to explicitly {000}

W 7 T Lec 2-21-17

- Prolog
- How do we add 2 + 2?
- is walks into machine code and evaluates for the variable
- In this case, it comes up with the value of 4 and it binds N with 4 and succeeds
- ?- N is 2 + 2.
- N = 4
- ?- 4 is 2 + 2
- yes
- You cannot run “is” backwards!
- ?- 4 is 2 + X.
- **What is X?**
- **BAD!!!!**
- Second argument of “is” must be **variable-free**
- This is a **ground term**
- Ground term is the kind of data structure you see in OCaml
- Variables are only in your program
- When you have restrictions like this, you run in functional language mode
- ?- 2 + X is 4
- Only ints are allowed on the left side
- no
- ?- N is 3, P is N + 1.
- ?- N is 3, N + N + 1
- 3 is 3 + 1
- On what planet will this happen in math, NEVER!

Debugging

- Helpful to know an introductory approach to debugging
- g(X) might have done a lot of work, but once you succeed, you will call h(X)
- You could backtrack into g(X) and fail

- g(X) may have alternatives and thus, FAIL!

?- trace.
yes

- Every time you cross a port boundary, it will generate voluminous output
- You can type commands to abort the computation or keep going
- The idea is that if you are kind of lost about how Prolog should work.
- When you are trying to learn Prolog, draw a small predicate and run it in trace mode
- Don't give it a huge question and try to single step through it with this 4 port model
- Similar to tracing Java code, except Java and C++ have a 2 port model
- When you call a function, it returns
- In Prolog, rules have 4 ports because there are two ways to get out, and two ways to get in

Memory management

- These are put on the heap, but there is a more efficient way of implementing these data structures in Prolog without using the heap
- Put all terms on the stack
- When you succeed, this means you are pushing onto the stack
- When you fail, this means you are popping from the stack
- Stack grows as you succeed, shrinks as you fail
- When you fail, logical variables become unbound
- You may have some terms that when they start off, as you succeeded, you bound these variables.
- When you fail you unbind them
- X = 37 and there is a compact representation as pointers to the value

Q. How does Prolog decide which variable to point to?
A. Left to right is probably what it does and as far as functionality goes, we could have done it another way

- When we backtrack, we have to undo this assignment.
- How do we know what things to undo?
- Not just terms, but also the stack contains a trail
- A list of variables that have been bound
- When you pop a trail entry off the stack, you better unbind it now!
- When you pop here,
- The trail is a stack!
- Grows as you succeed because you bind variables
- Shrinks as you unbind variables because you pop off this stack

Advantages

- Because these data structures are pushed onto the stack, accessing data is FAST
- Don't worry about heap management like in OCaml and C++
- Heap management is slower than stack management

Disadvantages?

- Eventually, the stack will overflow!
- If you have a very successful program, then it will keep succeeding and building terms
- This is a problem that occurs often enough in Prolog, and they wait until your stack gets too big and they say hold up.
- What they do is they take this enormous stack (GBs full of data) and they look in the stack for objects that cannot be referred to from here on out.
- These objects are NO longer accessible!
- **Prolog interpreters are like heaps because this is a garbage collecting stack**
- Normally, the program is like a stack but in practice, you still have the problems of a heap
- It probably won't blow the stack at all and it will run really fast.
- Algorithms that involve a lot of backtracking use better memory management principles than the same algorithms written in OCaml and C++
- **If you try to write some sort of random thing that succeeds all the time, then Prolog is probably NOT the best choice for efficiency**
- There are reasons to use Prolog for particular applications like if you do N-queens
- The program for N-queens is easier to write in Prolog than C or Java
- Backtracking is also much faster in Prolog, so the program will be faster

Variables and how they work (Unification)

- Process of taking two terms and unifying them so they become the same term
- Match a goal to the head of a clause to see if it is applicable
- Make these two terms equal down to the byte code!
- These two terms can contain logical variables, and we can bind these two logical variables to make the terms equal
- At each point, if you see a structure, make sure the function symbols are the same
- If the -arity is different, then you cannot do it and they cannot be possibly the same

Something in common with unification and pattern matching in OCaml

- There could be a goal at the top level
- More general in Prolog
- When you do a pattern match in OCaml, you **bind local variables to data**

- You can also do this in Prolog, but Prolog matches in **BOTH DIRECTIONS**
- You can bind variables in your data to local structures in your program
- There is a call where we bind data to code and code to data in a single call

```
p(X, X).
?- p(Z, f(Z)).
```

- These two data structures have to be the same

Prolog

- The problem isn't an implementation problem, but rather a philosophical one
- Part of the appeal of Prolog is that it is based on logic.
- If you go to the philosophy department, here is the answer that I get and the philosophers will look at you and say you are in trouble

Suppose we want Prolog to do arithmetic

- Assume there is no "is" predicate and we have to implement arithmetic on your own.
- Assume no int, long, char, float, double, etc.
- How the hell are you going to do this?
- This will be an issue!
- But it is possible in Prolog

We can write a predicate that will let us add two numbers using this representation

- This forms Peano arithmetic
- Working on a Hilbert proposition to formalize mathematics
- The goal was to take all of geometry and arithmetic and come up with a formal basis of it to prove anything that was true and disprove everything that was false
- This project failed because it crashed and burned
- Godel's Incompleteness Theorem
- Same result as the halting problem because you can reduce this to something undecidable.
- Ask the Prolog interpreter a question and it will get stuck!

Cannot prove infinity < infinity

- This Prolog interpreter is giving us the wrong answer because the infinite term is logically bogus
- You should NOT be allowed to do that.
- Logic does NOT allow you to assume infinity
- There is a built-in predicate in Prolog

unify_with_occurs_check/2

- If unification creates a cycle, then fail.
- Rarely used but used to pacify logic professors!

- Considered bad style to create infinite terms
- 1) most programs don't create infinite terms, they are bugs.

Analgous to "Don't create circular lists in Java!"

- This is bad practice

- 2) Slower because it checks for loops
- When it binds, it has to make sure it hasn't created a cycle
- When you bind a variable with a term, you have to wander through the term to figure out if there was a Z there.

Prolog has an order of evaluation problem

- OR glues together all the parts of a predicate
- AND glues together subgoals in one particular clause
- Think of all possible sets of executions for a Prolog program
- Big AND-OR tree
- Evaluates depth-first, left-to-right

If we get to here, these solutions cannot work

- Avoid useless predicates because we already evaluated true!
- There is an operator in Prolog called "cut" that lets you do that!

Cut
! is the symbol

- A little bit dangerous!
- Always succeeds when you call it
- When you backtrack into it, it fails and causes the calling predicate to immediately fail
- Do expensive test 3 times with the same value instead of cutting off with 1 single time
- Prunes away all alternatives in current predicate
- There are NO other alternatives in that predicate that will be examined
- This predicate can have 47 different other clauses, and all of them are equivalent
- Once we execute the cut, the rest of the predicate is gone and you finish.

If there are 5 different ways of proving something, just put in a cut and stop once you find at least one method that works!

With cut, you can approximate negation!

- If you want to prove not(G), where G is some goal, you try to prove a G and if you succeed in proving G, you do a cut then fail.

Ordinary fail will backtrack, but ! will cause the calling predicate to fail and prune away everything

- You have to know precisely how it works to make sense of it.

Once you have \+, it is no longer a commutative operation!

- Cut is a powerful operation that messes with the logic of your program
- Gives you the wrong answers sometimes because you can fail out even if there was an answer there.

Need a good heuristic to see when it is safe to use \+

- If you have a ground term, that will be pretty safe.
- **Be careful when calling \+ on bounded variables!**
- If the bounded variable does have a value at the time, then it might be okay.

This operation is as expensive as a call to G is and do whatever work G says to do

- Not G will be in general, faster than G because you prune away all the alternatives
- Not G is faster than G because it prunes away things

There will be other uses of Prolog where the Closed World Assumption is NOT correct.

- Logic says that this is NOT deducible from these statements.
- Nothing in here that says dance101 is NOT a prerequisite for this course.

Suppose you are working for NSA and X is a terrorist.

- Database that lists known terrorists and you can check if someone is a terrorist or not.
- You cannot make a Closed World Assumption in that case.
- There are people who are terrorists that are NOT on the list.
- In general, you have to be conscious of when an assumption is accurate or not.
- Make sure the assumption is actually true, and this isn't true just for Prolog!
- True for Java, Python, and C++.
- Some cases where you do NOT want to make a Closed World Assumption

W 7 R Lec 2-23-17

Definite Clause Grammars

- Another feature that is built-into Prolog that is syntactic sugar
- Instead of +, *, =, this lets you write parsers more efficiently or easily than you would in Prolog
- Put terminal symbols in square brackets []
- HW 2a - use Prolog instead of OCaml
- This would be a "easy hw"
- Just take your input grammar, translate to definite clause form
- You get a parser for free!
- Choice of notation matters!

- If you try to do HW 2 in C++, you would have gone nuts because it would be insanely hard.

You still have to know where the language is coming from so you know where the strengths and weaknesses are

Prolog theory (based on logic)

- Based on mathematical logic
- Some philosophy and mathematics involved
- Propositional logic:
- “It is raining.”
- “Traffic is busy on the 405.”
- We want to think about the logic behind it rather than specific propositions.
- Give them little names. i.e. p and q
- Connect together propositions with standard connectives
- Convince you that there is more to learn about these connectives
- Learn more about logic even when we think we know what is going on.
- Different camps of logicians
- A lot of them don’t like & for AND
- C programmers love ampersand
- Different people use different symbols but they all kind of mean the same thing

Causality

- Important topic but it is NOT what implies is about
- NOT about causality, only about implication
- In logic, if p is false and q is true, then p implies q is assumed to be true
- Implication
- If p is true, but q is false, that is the ONLY time p -> q is FALSE

AND is commutative and associative
OR is commutative and associative
IMPLIES is NOT commutative and associative
XOR is commutative and associative
IFF is commutative and associative

(p ^ q) <-> (q ^ p)
These two expressions are equal

- There is something interesting in this statement!
- This entire statement will be true no matter what
- **tautology**
- **Every possible rule is TRUE!**
- Don’t give you any new information in the real world
- Unable to derive any meaning from it

- **non-tautology**: p -> q
- Derive something meaningful out of this logic
- Non-tautologies are the interesting statements

In Java or C, this is what you have

- Bool is an implementation of propositional logic
- Probably shouldn't put tautologies in your program
- You have been doing this for months when you have written your own code!
- Propositional logic is NOT always enough in the real world

Socrates drunk the hemlock and came up with this idea

All men are mortal,	p
Socrates is a man	q
_____	
Socrates is mortal.	r

- Propositional logic is not powerful enough to solve real world problems

Predicate calculus

- logical variables
- quantifiers

Logical system that is powerful enough to handle a 2,400 year old syllogism!

- Aristotle could do this in his sleep!

We need a procedure of seeing whether it is true or not

In general, you can encode integer arithmetic into predicate calculus.

- You cannot tell if it is true or false!
- **Impossible problem!**

We don't need to solve this problem in general; we just need to solve this predicate calculus problem often enough to get our work done.

- Massage predicate calculus formulas so they are easier to think about

clausal form:

- Slow, complicated
- Hard to think about
- AI class will talk about the resolution principle!
- Look at a stripped down version of this problem that Prolog looks at
- Hard problem.
- Is there an easier way of looking at this?

Prolog uses proof by contradiction

- Goes on and uses all the facts and rules in its command to try to prove you are wrong
- If it cannot find a counterexample, then it says NO and I failed, so that is the best I could do.

Connection between Prolog and logic

- Connected to the way mathematicians think and try to be accurate about what they know and don't know

Storage Management

- RAM
- Fit the program into RAM as much as possible
- What do we have to manage?
- static variables - stay alive as long as the program is alive
- static code - We know the exact location of the code!
- Stack is the solution to the problem
- Put local variables onto the stack
- auto - like static except local variable to the function
- This would save things in a "temp"
- Local variable that you didn't declare but the compiler declared it for you
- f needs to know where to return to
- On x86-84, it is stored in memory
- Manage return addressees
- Program will have to manage the memory that contains the data and the code
- Have a manager of some sort to help us do this
- Standard library will be the C library and this is called the memory manager
- Memory manager
- Library code that manages all this
- If memory manager cannot get enough memory, the whole system falls apart
- **Gives itself priority**
- Multithreaded too!

Array Layout

- How do we represent arrays in memory?
- This array can be overkill for my application and I wasn't born until 1954.
- If we find the address of the head of the array, we know where everything else is!
- Ideally, accessing should be as fast as possible!

Multiplying two n-bit algorithms are O(n^2)

- You can make it faster by using shift operations!

Have the pointer point at the 0th entry of the array even if it doesn't exist

- This gets rid of subtract!

Insert a runtime check into our machine code

W 7 Dis 2-24-17

- Scheme should be easier than HW 4 and the HW is pretty easy
- 10 questions and maybe 1 question is hard
- There might be some confusing things
- HW 4 deadline is today and Homework 5 is published

Install Scheme on http://download.racket-lang.org

- Need to still test on SEASNet in the future
- It looks different on SEASNet, but use this first because it helps you visually

Arithmetic & Comparisons

- Binary Operators

> (+ 1 2) -> 3, (- 5 3) -> 2

- Equivalence in value (#t = true, #f is false)

If your code doesn't work, try to add parentheses

(list 1 2 3 4) -> '(1 2 3 4)
> (list '+ 1 2) -> '(+ 1 2)

Racket is an implementation of Scheme

- Scheme has some top-notch documentation
- People who learn Scheme say its easy (CS 31 stuff)
- Introduction to programming class

Functional programming - Scheme is basically LISP from CS 161

cons: important part of the HW

- Kind of like a pointer data structure
- Value produced by cons doesn't have to be a list
- You can just have this guy point somewhere else and do crazy stuff
- Generally used to build lists in Scheme

car = head, cdr = tail

- In addition to car and car,, we have cadr (car of cdr) i.e. 2nd element
- caadr -> car of car of car i.e. 3rd element

if (expr) (thenDo) (elseDo)
> (if (= 1 1) 1 2) -> 1

if & cond

- If you want to execute more than one expression in a branch, you have to wrap the expressions in a **begin**

or & and

- Scheme is short-circuit evaluation
- >(or #f 2 3)
- 2
- Why does it return false here?
- Strings are just always true!

Defining procedures

- define is the “let rec” equivalent in Scheme
- (define (square x) = (* x x)) ->
- (square 2) = 4

HW 5 Help

- Check if it is valid to use cdr here

The last two functions are the most difficult parts of the API

- This cons object represents a list of something and we now want to return a list that returns this listdiff
- Return this as a list and return something that looks like a list

W 8 T Lec 2-28-17
Scheme

- Teaching subset of Lisp and not exactly a subset (differs)
- Lisp has been a very popular AI language
- Traditional AI - NOT machine learning
- Based on logic and arithmetic
- A language that has a long tradition in computer science and ideas that when they were first introduced were considered way too bleeding-edge and inefficient
- Slow, academic side of things
- Too slow for academic use
- Ideas that we will see in Scheme will make their way into more practical systems and the rest of them will migrate more in the future.

What makes a language Scheme rather than NOT Scheme?

- Scheme values are all Objects in the OO programming sense
- Have pointers and addresses
- Scheme is OO in this sense
- Dynamically allocated so program can allocate an Object whenever it feels like and calls the equivalent of “new” whenever it wants
- They are never freed, so the program never has to free an object
- Up to the system of when to handle this condition

- Concept of Garbage Collector (G.C.)
- Like M.L., OCaml, Java, Python
- Unlike C++, C
- **Everything is an object in Scheme including integers!**
- Objects at runtime have types
- They belong to object values
- Manifest types are obvious if you read the program, you can just see it!
- Do dynamic type checking here, not static type checking.
- Scheme is like Python which does dynamic type checking but unlike OCaml, Java, C, C++ in terms of interpreted side rather than compiled side
- Scope checking - we want to do all that checking at compile-time
- Use caller's names to resolve things you don't know
- Very simple scoping strategy that means you don't know what X means until runtime

Q. What if you have two variables with the same name?
A. That would be an error in scope.

- You look at the closest call and this is like static scoping in C

Bash uses dynamic scoping for environment variables

- In CS 35L, we constantly tell you to set the PATH environment variable so you can run a shell script to run the value of that variable
- What is the current value? It is set when you log in.
- **Performance issue for scoping (Scheme doesn't use this!)**
- **Advantage of static scoping is speed!**
- **Misspellings in the program are also checked before it runs in static scoping**

Q. Why would you want dynamic scoping?
A. Lessens the number of arguments at runtime. Like setting the PATH environment variable modifies all of your shell scripts. Set your PATH variable to use your version of grep.

- Gives you extra flexibility and another way to modify the behavior of programs besides the standard ones (passing in extra arguments)

- call by value
- Evaluate the argument, get a value, and pass a copy of that value (a copy of the pointer to an object) to the called function

- Objects of procedure types are first-class objects
- Same namespace and values as everything else
- All of this is the same to Scheme and we don't have a special sort of case for procedures

- There is a special subset of procedures which will be called **continuations**
- Don't work the way you are used to
- Not functions you find with arguments
- Procedures you create and they immediately represent the entire future of your program which you later replay.
- Like snapshots but not really snapshots.

- Simple syntax with straightforward representation of programs as data
- Like Prolog
- More complicated than Scheme in syntax
- Even more radical than Prolog
- Unlike C, C++, Java, ML, Python
- Solve HW 2 just to get started
- In Scheme, it is trivial to write a Scheme program that reads in another Scheme program and checks its syntax
- A shit ton of parentheses

Lots of
Irritating &
Spurious
Parentheses

- LISP

Metaprograms has made it survive for many years!

- Tail recursion optimization is required
- Call zillions of f until you call h
- Your stack will be proportional to recursion depth and you would have problem if you try to concatenate a list of one million
- Depth of the stack becomes bounded and as a programmer, because Scheme says every program uses tail recursion, you can take advantage of it.
- You won't blow up the stack because of this!

High level arithmetic

- Take an integer and keep squaring it, multiply it then by itself. What will happen?
- You will run out of memory
- You won't have an integer overflow, but you can generate data so big that it doesn't fit into RAM
- The only thing that can go wrong where the code doesn't match math is that you can exhaust memory
- Works like it did in 3rd grade!

Scheme also has floating point numbers if you want them like C

- Has complex numbers but only EE majors care about that

Combination makes the language Scheme and any other language with these properties is probably a derivative of Scheme

Q. How is Scheme different from Common Lisp?
A. Heavy emphasis on static scoping for Scheme. Common Lisp has a bigger library and intended for production code.

- Scheme was intended to be more stripped down
- Eggert used Scheme in production for small stuff
- A big AI system would use Common Lisp probably

Syntax identifier

- "*" is used all the time in Scheme
- NOT a built in operator, but rather a function in standard library!
- Don't allow +, -, ., or digit to start an identifier (except +, -, …)
- This sort of freedom of identifiers will let you do all sorts of stuff.

By convention, identifiers ending in ? are predicates

- eq?
- Functions that return Boolean

Does capitalization matter?

- cons, CONS, Cons, cOns

Lists

- (l i s t . s)
- () : empty list
- (a b . c)
- Improper list
- Data structure built out of pairs except the last item isn't nil
- . is just syntax like a parentheses to represent this notion
- Proper list
- Either ends in nil or a pair

Booleans

- #t, #f
- **#f is the only false value**

Vectors: Contiguous vector used for convenience in Scheme and can create a large object and we don't want any overhead of other points.

Strings

- Exist in Scheme and syntax is like C

Quote character - symbol a is unevaluated and this expression returns true if this local variables value is its own name

- Look at little tick marks because they matter in Scheme

Quasi quote - a template of a subexpression where you fill in values preceded by commas

Scheme semantics

- (a b . c): Function takes two or more arguments, and passed into trailing arguments (c in this case).

List semantics

- This function list is a standard function in the Scheme library
- If it didn't exist, we could define it ourselves

Calling a list function

- I should mention that this idea of define and lambda is so common that there is a common shorthand for it.

if (A B C)

- if A yields any other value than #f, then it looks at B's value and sees it if it can return. Otherwise, return c.

(and E_1 E_2 … E_n)

- Short-circuiting and
- If everything works, it returns the last value!

(or E_1 E_2 … E_n)

- Short-circuiting and
- Returns first successful. otherwise, it goes through and returns #f

Once we have a lambda and a call, we have our core semantics

- What is the point of let?
- Both of them bind local variables
- Look at order of evaluation and it really does matter because arguments to a function have to be evaluated in Scheme before the function body is evaluated
- When you write the code using a let, it runs in the order that you call them.

W 8 R Lec 3-2-17

Metaprogramming

- Topic that is very tempting for people who like programming languages
- If you see code that uses meta programming, you are also likely to say this is "programming and programming"
- Easy to go off on the deep end (too far)

Macros and metaprogramming is controversial!

- Mentally, you have to know what is going on
- Suppose you make a mistake (typo in your program)
- What is the error message going to look like?
- Something weird!
- Debugging tools go haywire because they are NOT on the same level
- Milstein loves meta programming and goes way off the deep end
- Eggert is far more skeptical because we might be getting into trouble

Create new keywords in Scheme, and these keywords can stand for whatever you want

- Define AND using one of these syntactic binding constructs
- Operating using pattern matching at compile-time RATHER than run-time
- Defining a macro instead of a function
- Rules checked left-to-right, top-to-bottom
- Like a Prolog program is running here
- Don't do backtracking but rather just one rule and go with it
- Suppose you do "and" of one argument
- It should just be that argument

"and" is NOT built-in; it is just a macro supplied by the standard library

or is buggy!

- If there are side-effects, this will end up ugly!

If we were insisting on truly functional code, one could argue this isn't incorrect.

- This macro could possibly be slow, but it could be an expensive function (3 seconds to evaluate)
- In that case, our macro will make it look like it spends 6 seconds rather than 3 seconds
- This macro implementation isn't good enough

Solution: Create local temporary variable and use that temporary variable

What spelling should we use for this variable "v"?

- Suppose we have a macro implication in our code
- We are dealing with a variable called "v"
- Macro "v" can collide with variable "v"
- This is the **problem of capture**

#define CPUSH(x)

- **Evaluates x twice, so expensive operation**
- Replace this definition!

Macros are troublesome, so this is part of meta programming that kind of doesn't work very well

- How do we solve this problem in C?

Tell the preprocessor to give me a symbol that the program isn't using.

- Attack this problem of capture with this hack but it is awkward.

The v is only visible in the macro itself

- Scope of v extends into the curly braces
- The problem is if we call CPUSH of v, which is textually substituted into the body, which appears in the curly braces
- v should look outside the curly braces, but this is migrated into the curly braces and this doesn't work in C the way we want.

syntax-rules

- Evaluate expressions left-to-right
- When the compiler sees this pattern, it can mentally substitute for the following expression
- See the "let" and this is equivalent to a lambda
- The way you specify a call in Scheme is to use an open parentheses, a function and its corresponding arguments
- If we see the "let tag" pattern, what code do we turn it into?
- Appeal to a lower level primitive in Scheme called "letter"
- Think of it as ordinary "let" but scope extends to initial value.
- They can look at their own names!

- The scope of this variable tag doesn't extend into the call
- Don't expose its name into this function

car cdr (anecdote)

- Implemented on a 2 register machine
- Address register: pointers
- Decrement register: index to decrement when you use it
- We used these registers for array subscripting
- With LISP, they didn't care about arrays. They only cared about pairs
- This was put into the address register and this is where these names come from
- Address register and decrement register

Equality operators

- The distinction between the four kinds of equality in Scheme comes up in other languages too!

(eq? a b) - check if they are the same object. If yes, return #t. Else, return #f. Pointer comparison rather than value comparison

Read entire program, return floating point constants, and check duplicates!

- This call will look like 2 pointers to the same thing
- Totally a runtime thing

Scheme allows infinite precision integers

- Could return #t or #f and it is up to the implementation

Local variables

- If you have local variables being compared, those have to be equal because you are comparing object identity rather than object value

eqv? looks at an object's immediate content

- Looks inside the strings and compares their content

equal is slower because it is a recursive comparison

- Does NOT just follow a list to its end, but it also recurses down through two sublists
- Tree comparison rather than just being list comparison
- Can throw arbitrary objects and recursively descends through all objects that Scheme defines
- Has a deeper problem!

= is numeric comparisons

- These are 4 functions in the Scheme standard library
- Semantics and NOT syntax
- All the arguments have to be numbers, and if you think about the numbers mathematically, all these numbers have to be the same

Continuations

- Controversial topic
- Large number of Scheme programmers and theoreticians who think it is too powerful
- Like ! in Prolog and goto in C
- Possibly removed and toned down in the next version of Scheme!
- Low-level and very powerful
- Maybe too powerful for their own good
- We are NOT simply evaluating or executing instructions
- We are executing instructions inside a context
- This context is a data structure that specifies things like my return address
- I am going to call this point the "ep" - environment pointer

W 8 Dis 3-3-17

- Moving on to the Project.
- Work with a library called Twisted

- Other servers will learn the location without having to talk to the database
- Any other server that can reach that server should have access to that information
- Look at Twisted Event driven network

Research project
- Working with Twisted who provides you with most of the library function calls you will need
- IAMAT tells you what kind of message it is and the client ID

API
- Google Places has a Nearby Search request and we can search this HTTP URL with all these return values
- You can always default to Python's own web stuff.
- JSON metadata returns all this info to create nice little pictures showing you where you are

Flooding algorithm
- Servers should continue to operate even if neighboring servers go down

Log input and output into a file
- Don't publish your key!
- The big part of your project is this mini 5-server herd
- The other part is a written report that summarizes your research and make a recommendation and describe problems that you ran into

Pay attention to the report guidelines so you don't lose bullshit points!

telnet - You can ping certain addresses via telnet
- The easiest way to test your server is to use telnet
- Look at the man page for it
- Give it the IP address or localhost as well as the port number

Maximize the time where you are actually doing stuff

Event Driven Example:
- Instead of waiting for something to finish first, you can differ a response until you get that information
- The nice little diagram is this
- GET twisted matrix
- GET google
- Wait for response and don't be idle while waiting
- You need a way to parse this message

Please log your information because it makes it easier!

- Do NOT use UCLA_WEB for the ports
- **eduroam**

W 9 T Lec 3-7-17
Midterm Statistics
- median: 57
- mean: 55.9
- standard deviation: 13

Continuations in Scheme
- NOT like POSIX threads but use these in other kinds of languages
- We will model a multi-threaded application as a bunch of continuations
- Append continuation to the current list of processes

Minor bug in this code because we are calling thunk with no arguments
- If we use this code as is, we will have a runtime error and this is a continuation k we have
- Pass in no arguments and we want

Suppose I have a thread and I want two copies of that thread
- If one copy yields, how would I write fork?
- Don't invoke start because that means I exited.
- Instead, I keep going!

Simplify implementation of yield as follows
- yield is a fork followed by a thread-exit

Cooperative threads are nicer to implement and we don't have to worry about race conditions as much
- What are some of the disadvantages of this approach versus POSIX threads?
- We don't get the processor improvement because there is only one thread active in any given time
- Only one instruction pointer that is actually running!
- If you want to build a true multi-threaded system, this won't suffice
- Useful for the IoT since you have one CPU anyway
- Saves energy
- This kind of threading is available in other languages but perhaps NOT quite as elegantly as this

C version
- These primitives don't know about the stack
- It is up to you to make sure the stack is right
- If you are careful in this, you can make these primitives as a substitute to threads
- **Reasonable compromise for a low-energy consuming device**

Scheme is about stripping down a language to the smallest primitives you can so you can understand the key concepts of the language rather than gingerbread.

- Although continuations look like they are built-into the system, in some sense, we can do without them and write continuations ourself without call/cc

We call the continuation function and pass in the value

- Take the program we wrote last time which walked through the list and computed the product of all the items in the list
- When it finds the 0, run the continuation so it keeps going
- Call break of 0 once you find a 0
- Have an internal recursive function where we use named “let”
- Take two arguments because we take the list of numbers we want to multiply as well as a continuation
- We could have a number that is 0 at the start of the list

Continuation multiplies the value that our callee returns and passes it to our caller

- There is nothing new here; this is the same concept as HW 2

Q. Can we do this in C, C++?
A. NO! Why not? **We cannot use lambdas in C/C++**.

- This is a procedure that refers to a local variable and mentions a local variable.
- This procedure is something we will hand off for someone else to run.
- You cannot have a procedure that is active after we have returned.

This makes continuations work well in Scheme

In Python, how do you simulate this kind of snapshot?

- Take a snapshot of where your function is and resume that snapshot later.
- This is a central problem in **Twisted** framework
- Deferred object in **Twisted**

Storage management

- What happens to allocate the array?
- Array gets put on the stack
- This array can be allocated as follows

Each function has a fixed-size frame

Controversial

- Calculate the size of the array and you won’t know the size at compile time
- You won’t know the address of the array relative to the start of the stack frame

- If you have varying sized frames, this implies that you can no longer survive by simply having a stack pointer
- Have a stack pointer that tells you what the top of the stack is and have a frame pointer

Q. How is stack overflow detected on most machines?
A. Stack pointer looks like this and each time you do a call, you push something else on the stack

- Say you have a forbidden zone, does this approach know?
- Whatever number we pick, take that number and divide it by 8 and add 1
- As long as you don't touch that array zone, it should be okay.

One solution is to write the program so that n is never so large that it will flip over the negative forbidden zone

- **Don't put subscript errors in your code**
- This means your program is buggy but people make mistakes!

Q. Is there a way to gain access to the forbidden zone?
A. There are some ways to use your memory map.

- Buffer overrun attacks will create a request to artificially create a need

The forbidden zone will catch recursive programs that nest too deeply

- Catches infinite loops

Subscript check on this n and the runtime can look at the end and check that n is actually in the proper range for the current stack.

- You need to have a stack limit pointer as an extra pointer and an extra runtime check if you have an array of unknown size.
- Microsoft and **Apple** do NOT want this check
- They prefer fast code that is dangerous

Nested functions

- Need a runtime representation to see what the x's are
- Each of these frames have some way of telling you where the frame is

Static scoping uses the same technique that gets continuations to work

- Function is NOT represented by a single pointer but rather two pointers
- Each function is for the instruction pointer and the environment pointer
- Points to the instruction's frame
- If f is recursive, you can point to an instance that is valid for that particular occurrence of g

allocation via

- fixed-size frames
- variable-sized frames

- static vs dynamic chains
- heap vs stack
- registers
- Registers don't have addresses

Q. What about embedded systems?
A. You add on TLS (thread local storage)

- Don't worry about other threads competing for it.

volatile

- Attribute of a variable that is to some extent orthogonal to where you put the variable
- Even registers can be volatile if you are using a register that others use

W 9 R Lec 3-8-17

- Max of three numbers that are close to each other

Scheme vs Emacs Lisp

- If either number is floating point, it should convert the number to floating point and return the result as a floating point
- This rounds to that and so, you get 1e16
- We haven't considered what the right answer is!
- The Emacs guys haven't decided this yet
- Are we okay with this check?

- This error does NOT come up in C, Java, JavaScript, etc. because we have pair-wise comparison
- CanNOT compare more than 2 operands!

Heap Management Issues

- We are trying to figure out what sort of programming language features or aspects are going to be reflected by the fact we have a heap manager and we have to keep track of the objects our program is dealing with.
- Do we have a garbage collector?
- If not, we place the responsibility on freeing objects for the programmer
- We get fine-grained control over your storage in this case
- Also more efficient!
- **Downsides**
- Leaks: people forget to free the storage and it clogs up the RAM and people complain
- If you refer to an object after you free it, this will cause issues!
- **Dangling pointer**
- Like using an uninitialized behavior but worse
- Strongest argument against C++ because of this issue!
- This is one issue you have to decide fairly early on in language design and we should be agnostic here!

Suppose we assume GC

- If we assume we have a garbage collector, how do we implement this?
- **Registers are used to keep things fast since we don't have to copy onto the stack**
- Need to know where these things are, and this is the problem of finding roots!
- When you are the garbage collector, you get called when storage is low
- Allocate buffers and allocate memory
- Call the garbage collector and figure out which part of the heap is used
- **Which part of the heap are objects that no-one cares about anymore?**
- Reclaim that portion

Let's say you have to build a manager, what is a good way to do that?

Root-finding

- Compiler records where the roots are
- Compiler and the linker together give you a table of all the roots and this table will find all the stuff in static memory
- A single static table cannot tell us what the stack layout is currently because that is a dynamic function

If you look in a stack frame, there are return addresses, integers that are NOT pointers, etc.

- How do you figure out what is junk and what is valid?
- Once you find the pointers, you can talk about its reference count!
- Create pointers back to the roots and what we can do for each activation record, we can have a little pointer to that activation record.
- This points into our table and specifies the layout of the current activation record
- Tells the heap manager which slots in the stack are pointers that we have to worry about
- What the garbage collector does is walk through the stack and each descriptor gives it information.
- Will this approach work?
- Probably NOT!
- You can have roots in registers
- **Here is a pointer and there might be NO stack area that points to that same object**
- This is starting to get a little bit tricky here
- You can have a static table and that table will tell you, for every machine instruction, you can see which table contains roots when the instruction is executed.
- Machine instruction addresses to patterns of which contain roots and which don't

- Static table is NOW looking very complicated
- Complex table that maps instruction pointer (IP) values to sets of registers
- When a garbage collector is invoked, it looks up what your instruction pointer is and looks at those roots
- After you do a move into a register, you should update your static table
- For each object, you can have the descriptor that points to a static table and this tells you the object layout and where the pointers are
- We map heap addresses to object layout
- Another approach that avoids this need for book-keeping
- **We do something that is almost astonishingly lazy!**
- Method B will NOT generate any tables so we do NOT rely on bookkeeping
- Instead, the heap manager will look at the registers and the stack and take the following algorithm
- See a register and if the value equals the value on the heap, we assume it is a pointer
- We can do that for every word in the stack
- Look through all the words in the stack
- **If the word smells like a pointer, it is a pointer!**
- **Don't need a descriptor value and just look at the words and since we are the heap manager, we know where all the objects are.**
- If it contains a bit pattern that looks like a pointer, it is a pointer.
- Otherwise, it is NOT!
- How do we then allocate and free storage?
- Which objects can be reclaimed and which do we have to keep?

B) No tables, but guessing conservatively!

- AKA conservative garbage collection
- This object is in use and we won't collect it even if it is garbage.
- If you thought wrong, it is NOT actually a pointer and you guessed wrong.
- Most integers are relatively small and if you arrange for heap addresses to be large values, then most of the time, your guesses will be right.
- **Doesn't leave much memory left over!**
- **+ side is you avoid bureaucracy in creating table roots**
- Usable in C/C++!

If it runs low in storage, look on the heap and free all the stuff that isn't being accessed

- The advantage of this approach is that you NO longer have to trust the program and you can ignore it.
- If a programmer forgets to free an object that is garbage, he will catch it when the next error occurs.
- Reasonably commonly used in large C/C++ programs
- Used in browsers, gcc, emacs
- Dell is becoming obsolete!

A very common approach here is to use an algorithm called **mark + sweep**

0. Clear a mark bit associated with an object
- Clear all the mark bits via the storage manager

0. Apply this algorithm to find all the roots (either via table or conservative scanning algorithm)
- Keep scanning as long as you have more pointers
- Mark each time you have an object
- If you find an already marked object, **skip it**
- Recursive algorithm that is Depth-First, left to right traversal
- When you find a cycle, break it and skip to the next one

0. Find all objects whose mark bits are still 0 and free them

- Suppose I am a robot and Java uses a mark + sweep algorithm
- Will take a lot of memory to represent my internal state
- A downside of this approach is that this can take a long time!
- While you are doing this, can you let computation proceed?
- **NO!**
- If you are doing this delegate algorithm, your mark bits might become wrong and you might mark an object that is still reachable.

This is why garbage collectors are FORBIDDEN in nuclear power plants!
- Same applies to brakes in your car
- Cannot have this fail while garbage collecting; lives are at stake!

Q. What if we have a garbage collection thread in addition to a main thread?
A. The problem is if two threads access the same object. Race conditions!
- Doable, but you have to have good checks in order to maintain this consistently!
- Very complicated locking systems and slow down access and other stuff

Moving to another thread won't help with the problem of responding in real-time.
- Garbage collect before you need to and keep some storage free even if you don't need it yet.
- If you are worried you need it in the future, you can do **incremental garbage collection**
- Do a little bit of marking and if you are in the middle of a sweep space, I will do more sweeping now
- Allocation operation can do a little bit of garbage collection so no piece of garbage collection will take a long time.

Real time GC
- Hard upper bound for storage allocation
- Example: every "new" allocation will take 20 seconds or less

Let's assume that we are using a traditional mark + sweep algorithm

- Heap is an array of storage
- Heap arena
- We can see several different objects that don't have to be the same size
- We can have some free space
- Goal in managing this arena is keeping track of everything
- When someone calls new or malloc, it is your responsibility to figure out which free space to use.
- How do we do this?
- We can have a "free" list to keep track of this!
- It can be a simple linked list and each item in the list can say where the free block is and we can say how big the free block is here.
- Need an amount of storage that is NOT known ahead of time to manage our free list

- What if we have metadata to represent the free list?

A. Create a static area and no matter how big you make this array, it will probably be too small.

- It cannot hold enough info if it is too small, but if it is too big, we are wasting space.

You can use one word to represent this free list no matter how big.

- Take advantage of the fact no one is using the storage and allocate stuff into the empty pieces of storage

If we take this model where there is a free list, which is managed inside the free storage, what are some things we have to do?

- When an object is free, we are going to have to put it into the free list.

Add O_3 to the free list

- Naively, you can have a pointer to the start of the free list, and instead of pointing to the old value, we have it point to O_3
- Change O_3's first couple of words and then we can put it back into the old free list
- Make O_3 become garbage and we are done!
- Cheap approach, really cheap, but WRONG!
- **Doesn't coalesce adjacent free blocks!**
- You won't notice things so you will have a lot of fragmentation!
- **A problem with this naive approach is that it doesn't coalesce adjacent free blocks!**

If you have a general purpose manager, then this becomes a real issue!

- How do we coalesce adjacent free blocks cheaply?
- Record the length of the free memory and if there are two adjacent chunks, add the length.
- Every object has a length field.

- We want whoever pointing to this to point back to us.
- How will we arrange this?
- Use a doubly-linked list!

Suppose we free O_3 and in front of O_3, we have free space.

- What do we do?
- The problem is we don't know the size of this object.
- Use another piece of information which is the size of the next object before this one.
- Look for the object below the address of this object.
- In general, when freeing objects, you got coalesce twice (in both left and right directions!)

When you need storage, what will you do?

- **Allocation is fast if the first block is big enough**
- When you first stop, the block is big enough so this will be very fast!
- After allocating a lot, things start to get smaller and harder to work with
- Start carving off pieces of the second block
- A problem with this approach is that you end up with a lot of small blocks at the start of your free list
- NOT assuming we are using virtual memory
- Could be physical or virtual memory
- Scala or Java almost always use virtual memory
- Runts: small free areas (analogous to runts of a litter of pigs)

In a lot of these systems, most of our RAM is devoted to this and pushing everything to the side will cause your algorithm to pause

- Have a solution that avoids the runts without the necessity of moving everything around
- Could you have objects that don't span non-consecutive blocks of memory?
- You could but this will make other parts of your system less efficient.
- They have to check if it is 1 block of storage or something else.
- **Inefficient!**
- Access an instance variable by doing a load.

Algorithm I am using is first-fit

- Walk through left to right and the first object big enough, I will allocate off of

Another alternative is worst-fit

- Take object that is the biggest!
- Sort a free-list in descending size of order so most allocations happen very quickly.
- NOT necessarily a good idea!

- Once things stabilize, you are NOW going to have a number of free areas that are roughly the same size and when you allocate one of them, you have to re-sort the thing

roving-fit

- Free-list is NOT a list, but rather a cycle (circular list)
- Walk through the list and leave the free list pointer there so when you do an allocation, you work off of that
- The other blocks which are somewhere else in the free list ought to be growing gradually and walk through circular list again
- This is actually a fairly good approach for some applications
- A lot of different possibilities

Basic assumption about heap management

- **malloc can be slow**
- Many applications try to avoid malloc when they can because of speed concerns

Assuming x86-64, this will be 16 bytes and I will be calling this over and over again

- You are telling me that malloc can be slow!
- I better fix this then!
- Maintain a private free list FL
- Point to a bunch of cons cells, but I have NOT actually freed them

This concept of a private free list can greatly improve performance for typical malloc implementations!

Alternatives to the mark + sweep implementation!

0. Python - does not use mark + sweep
- Uses reference counting instead!

Reference counts

- Let's say you have a count with the number of pointers elsewhere that point to this object.
- If we have three pointers to this object from somewhere else in this system, we can count the number of pointers!
- With reference counting, the downside is that pointer assignment is more expensive

No longer copying one register from another

- Instead of one instruction, it could take 7 instructions

A downside and the Achilles' Heel of reference counting is cyclic data structures

- In Python, you could create two dictionaries and refer back to each other.

- Two of these objects d and e are pointing to each other even if NO other part of the program points to d and e.
- **Dropping reference count is kind of annoying**
- Attitude in Python is do NOT do this
- **Do NOT create cycles because it is trouble**
- **Reference counting algorithm won't catch cycles**

Python currently uses reference counts + mark + sweep as a fallback

Python atop JVM uses Java GC
- This is more portable than C-Python!
- In a Python reference count based system, this finalization method is prompt
- Without cycles in data structure, the finalization method is run when the data structure is NOT in use
- If you are using Python atop JVM, then finalization is lazy!

Generation-based copying
- Used in high-performance Java

We want "new" or "malloc" to be fast!
- Check if we run out of space, then we have to inevitably use slow code

"new" is very fast in OO code
- We don't have to coalesce free blocks so we avoid the problems mentioned earlier in lecture!
- What if you run off the limit and how do you reclaim storage here?
- This is how "new" usually works, and we contort the rest of our design so that this is our common path
- We only care about how the fast code works

Generations
- **If you look at objects at most OO programs, and look at how old those objects are, you will discover that you can partition all the objects into some generations**
- **Less common for old objects to point to new objects**
- Possible but less common

Let's say you fill up the nursery generation
- Where is your garbage most likely to be?
- If an object has been around for 10 seconds, then most likely, it will stay around.
- Partition your object into generations and focus garbage collection attempts on generations that need it
  - Collect the nursery

- Take all the pages of older generations and make them read only!
- chmod 1

If you want to find roots into nursery, look at pages that have been marked writable

- This becomes feasible to just collect part of your system and NOT your whole system
- Just collect area where actual garbage will be
- Occasionally, space will free up and we might have to actually garbage collect the whole world, but that is rare.

copying - just collecting the nursery and let's suppose that we are going through the nursery and finding all objects in use.

Since objects were allocated at the same time, there is a good chance they point to each other.

- Addresses in copy will point to a new copy and point to the same cash line.
- A big advantage of this copying collector is that it makes better use of hardware caches

If you have a root pointing to an old object, you have to point it to a new one

- Change it so it points to the new O_2
- Downside is that you have to relocate all the pointers to these moved objects

- Positives:
- When you are done, you have a fast malloc and realloc
- Better use of hardware caches

- This algorithm does NOT have a data structure inside the free list
- Mark objects in use, move them, and if there are dead objects in here, you never have to look at them.
- If there is a dead object in RAM, you don't have to pull it into RAM and look at it at all.
- Cost of this from cache's POV is proportional to amount of storage in nursery that is used.
- Free storage isn't touched and costs you nothing!
- **This can be an important point**
- If you created a lot of throwaway objects, you don't have to throw them away at all

Look at all of the stack and you don't know where the pointers are!

- If there is a bit pattern that looks like a pointer, then it is just a small memory leak and I am willing to pay that price.
- Assumptions for conservative collectors will cause this approach to do the completely wrong thing.

Warren Buffet now has less money he started off with and customers will be unhappy.

- Cannot combine this approach with the conservative approach
- Completely incompatible
- Know exactly where your roots are!

Multithreaded application

- Can have race condition in this case!
- Performance will be bad if we have a lock.
- Can we do better than this?
- If you lock the free list, then the two news will conflict and you will have a huge bottleneck
- **Each thread can have its own nursery!**
- Nursery based approach wins here too
- As long as two nurseries are independent of each other, you can garbage collect in parallel
- Commonly taken in multithreaded approach

Q. What happens if objects are in shared between two threads?
A. Each generation will have an elder, and a nursery generation will be owned by its thread. Put one thread in charge of any generation at any given time.

- For the purpose of garbage collection, you need to put a lock and it will be a bottleneck if someone tries to do garbage collection at the same time.

Q. How does this combine with the ideas of continuations in Scheme?
A. Continuations are objects that point to pieces of storage. These contain local variables and these pieces of storage manage and live on the heap.

- Do it in a Scheme implementation and have a copying collector.

W 9 Dis 3-10-17

- Project due today!
- March-09-2017, 11:55 PM
- Homework 6
- Evaluation of instruction reminder

Homework 6

- Write an academic report and it will look like an academic paper
- Doesn't have to be very fancy but don't put an ungodly amount of effort into it

Compare docker to these other languages {Scala, (Mozilla) Rust, (Google) Go, Elixir}

- Have a virtual machine and rapidly deploy some software to it
- Don't mess with OS but mess with things through there
- Deploy some software, make it run, and explain it!

Basically, we have some experience with Twisted and we want to deploy it somewhere

- Virtual machine - emulates an OS with some limited form of OS

- Having an entire OS will be heavy so keep it cheap
- This is why we use Linux Containers such as Docker
- Contains the application and adds small dependencies

Google's goal: Cloud and distributed computing - use this language (Go) instead of another one!

- One application example is TensorFlow - deep learning network

All of these languages are more and more exotic so we have to investigate and read about the pros and cons of these languages

- Are these good choices for distributed computing applications

Scala

- Statically Typed
- Checks for types before you run your program
- Java, C++, C
- NOT statically typed: Python
- Object Oriented
- Java, C++
- Scala is implemented on top of Java
- Runs on JVM
- Functional
- Get all these nice, short syntax APIs so you don't have to write a shit load of for loops
- Everything is an object
- Numbers

Functions

- Easy to learn from Java
- It is like writing Python code from a C perspective, but this is a nice thing to check
- **Loved by people in startups because of machine learning and distributing contexts**
- Compiled to Java Byte Code
- Need Java installed on your computer to run

Scala in action

- Big data frameworks: Apache Spark
- In Industry:
- Twitter, LinkedIn, Tumblr, Netflix, Amazon
- Introduced by a professor from EPFL (Switzerland)
- Entirely academic at the beginning and now it got hella popular
- Strong Static type system
- Prevents Bugs!

Rust

- Statically typed, make C++ 2.0

- Object Oriented
- Functional (Big difference from C)
- Replacement to C++
- Syntax is very similar and you can move on to Rust
- Memory allocation/pointers are similar to C/C++

Rust in action

- Not that popular of a language
- Lots of room for growth
- You can use it as a conversation starter because it is no new and possibly revolutionary

Go

- Leader right now
- Docker is written in Go
- Statically Typed
- Imperative style
- Syntax is similar to dynamic languages like Python
- Make it is simple as it can be
- The syntax is super simple and even easier than Python
- You can start to use it for a lot of your applications
- Built in concurrency primitives and everything is built-in

Go in action

- Docker - entire application was written in Go
- Used in:
- Google
- Dropbox
- MongoDB

Elixir

- Dynamically Typed
- Functional
- Syntax is similar to Ruby
- Marketed as lightweight - easy and simple to implement
- Claim you can run hundreds of processes at once
- It is your job to investigate this
- Replace Docker and Go framework with Elixir
- Built-in failsafe mechanism
- If my program crashes, do certain things and this would be really nice
- Built on this language called Erlang, which only really old people like
- Built in the 1960s and 1970s
- We saw with Scala that this was run on the JVM and Elixir was run on Erlang VM

Elixir in action

- Used at Pinterest

Final Pre-Prep questions

- OCaml
- Will be on the final

If it is tail-recursive, you have to call the function and this is the last thing we do, so we cannot remember what the value is

- In that function, we have to remember what the value is because we need to append it to those function calls
- When we call our traverse function recursively, this has to be the last thing we do in that function
- You need an accumulator so you can pass things on to the recursive calls
- **Add a parameter to the function call to handle accumulation**

What is tail-recursive again?

- You get it over with and never have to come back to that level again
- Save the results of the recursive call and you can just append the result of that

Try at home!

(c)

- I chose None of the above

(a) Describe one advantage of static typechecking over dynamic type checking, and provide an OCaml expression that illustrates the point

- You know if you mess something up before you run your program
- let bob = 3 and then you try to do bob @ <something>, this won't work
- Dynamic type checking allows for greater flexibility

W 10 T Lec 3-14-17

- Some of the book chapters will be stuff we already know, others we will spend more time

Object Orientation

- Object oriented languages like C++ and Java
- Object oriented subsystems such as OCaml

Some (newish) points at the top level

- Distinguish between OO language vs OO programming
- You can write OO code in C or Fortran or Scheme (non-traditional OO languages)
- In C, you can implement a class through a struct that contains function pointers

OO revolution occurred about 20 years ago, but NOT everyone agreed on it

- Wide variety of opinions or options regarding this topic
- Because of this wide variety, if you want a language to support your style of OOP, this will lead to a wide variety of language implementations and features

static vs dynamic type checking

- Static type checking would need something like **reflection** to get the same effects you would see in Python

Q. Should you have classes at all?
A. Why? There is another approach that has been around just as long as classes - **prototypes**

- With classes, the way you create an object is to invoke new
- For prototypes, you take an object that already exists but you clone it
- If you have a class-based language, you are probably going to need a cloning operation anyway
- There is one in Java, one in Python
- Changing the prototype is like changing the Factory for your class and this approach (**prototypes**) is more flexible but also more dangerous
- This all happens at runtime and NOT compile time
- With more flexibility, you have more responsibility!
- If you like flexibility in dynamic type checking, why don't we like it in prototypes
- Python has a botched design because it is classed based design but it also has dynamic type checking and prototyping
- It doesn't make sense for Python to be class based

Classic prototype based language is called **self**

- JavaScript is a prototype based language!
- If your languages is dynamic anyway, why do you need classes!
- What they taught in CS 31 and 32 is unnecessary!

inheritance?

- Classes are cool because you can have inheritance
- When writing code for a subclass, your job is easier because you can inherit code from your parents
- There is a school of thought that says inheritance is a mistake.
- Why NOT?
- **Inheritance breaks encapsulation**
- If one module is broken, it breaks everything else.
- Violates a standard software engineering rule
- delegation
- There are lots of different ways of doing OOP and we shouldn't assume the Java and C++ way are the only way.

Q. What does delegation look like?
A. Please implement this method by calling this method from another class

- Delegate the job of implementing an operation to some other bit of code somewhere else
- In Java, instead of inheriting something, we could have a member inherit something.

Although that is sort of a common thing you want to do, you could delegate this method from another class and it could be a straight jacket like Java.

Parameter Passing

- One of the most common things you do in real world programming is you call methods.
- When you call another function, method, or procedure, it is important to get that call right.

Calls must be fast and clear

- Call-by-value: widely used in Java, C++, etc.
- Most languages use this instead
- **Simple**, but there is a downside!
- **Doesn't work well with large objects**
- ***Unnecessary crashes if expressions isn't needed***
- Another downside but even worse than an object that is too large
- Call by value can make your object crash, while other things would be okay
- The way to attack the problem of dealing with large objects is known as **call-by-reference**
- You write one thing in your source code, and what happens is an address gets passed rather than a copy of the data
- Take the address of the object
- The callee will put a star (*) in front of the argument!
- **The main advantage is that large objects can now be passed efficiently!**
- Let's say we want a simple function that returns a.i - this will be fast with call-by-reference
- **Disadvantages**
- Accidental modification
- In C++, use **const**
- **Aliasing**
- Assume m, n are variables of type "int"
- It is completely possible if m and n are the same object!
- We want calls to be fast and clear, but in some cases like **aliasing, it hurts performance**
- **Call by result**

- Execute the function and when it returns, you copy the argument's value back to the call
- When you execute code in the function, you don't worry about aliasing between parameter and global variables
- Call by result decouple the local variables from the globals (i.e. heap objects)
- You still have the problem of passing two arguments are the same i.e. f(buf, buf) but why would you do this?
- **Call by value-result**
- Combination of call by value + call by result
- When it is done, it copies its result back to the caller
- The difference here vs call by reference is that **the callee is operating with a local copy of the parameter and doesn't have to worry about aliasing**
- Get the safety without the sort of problems you have with call by reference and it looks like it was originally a call by reference style
- Gets rid of aliasing by having all parameters be copies of the original, but if you modify the parameters, since they are independent copies, they cannot be aliases of each other
- When you return, you copy these back to the original and run the original code.
- Compiler will use call-by-refrence but they try to make it efficient in the normal case that you have no aliasing
- **Ada** - language used in aeronautics that makes heavy use of call by result and call by value-result
- Trying to do a good job, but could lead to issues
- Fortran - care about performance more than anything else
- The standard says you can implement a Fortran call via call-by-value, call-by-reference or any of those
- You have to be careful NOT to rely on any aliasing assumptions in Fortran because they might behave differently

- **Call by unification** (Prolog)

- **call by name:** call by reference ::
functions : pointers (*n) callee
- If you have a function n, you don't pass in n directly
- Rather, you use Lisp notation and pass in a func and when you call it, you will return n)
- This is what you pass at the low-level but you haven't evaluated n at all
- Packaged up evaluation of n in a procedure that will be called in a callee
- Let's say we have an integration function
- We can specify boundaries and a function, but **what is actually generated in the object code is a lambda expression!**
- Smart pointers - pieces of code that get evaluated and will tell you what you want

- Advantages
- Skips useless or dangerous computation
- Disadvantages
- Slower if all you're passing are values you will need anyway

lazy evaluation - put off executing code as long as possible

- When you run your program, the implementation doesn't actually do anything
- It just remembers what you want it to do
- Springs to action when your code decides to print something out
- If you want to print out this number, it lazily goes back and only calls the functions it needs to call and skips out on the functions it doesn't need

call by need = call by name
+ callee caches result of call

Haskell
let s = list of prime #s

- as long as you don't try to print it out, this code will work
- **Computes lazily!**
- This sort of thinking in Haskell is typical!
- You are always sort of describing the intent of your program using infinite data structures and your code still runs in a finite amount of time
- Actually, your code will run quickly and you won't get these features in OCaml and C++
- The people who like Haskell and lazy evolution have a pejorative name for call by value
- **eager evaluation**: Whenever you do a function call, you eagerly go and evaluate arguments even if you don't need them
- We want our language to be more like mathematics, so we like this notation!

Errors

- We don't want to admit that our programs have mistakes
- We don't want to waste a lot of time thinking about these
- A sizable fraction of your professional career will be dealing with errors
- Too many people who spend 80-90% of the time writing error handling code
- Chew up a lot of software development code and you can limit this by choosing your programming language right.

Clarity + efficiency

- Which of these two goals is more important?
- Give up one goal or the other
- What is the most efficient way to allow the application to run as fast as possible?

- Crash on errors
- Put copy of program's memory into a file for later inspection
- Simple and easy to understand
- Users don't like seeing it but it can be quite efficient.
- Is there something even faster?
- Don't handle the error and let it corrupt the data - undefined behavior (C standard)
- C focuses on performance over safety.
- Undefined behavior is truly crazy in terms of the possibilities
- Fastest performance and taken by traditional languages like C, C++, Fortran
- Exception handling
- What if the catch statement has errors?
- Nest it in a bigger try-catch block
- Changes the way you think about writing your code
- The way that you think about your code is important
- Two different ways of handling the same bit of code
- **Exception handling makes it harder to read code!**
- You need to understand the transfer of control for this
- You cannot really think of this as being straight line code!
- Branches built into it that you cannot see
- If you get into a large project, you might see a contrast in styles and people have to somehow agree
- **In most cases, we have RARELY used exception handling code because it is so much of a pain to write exception handlers for every arithmetic computation!**
- Compile-type checking
- You can design a language that checks pre-conditions and you know the error cannot possibly happen!
- Eggert was told two days ago of a Stanford professor who sold his company for $300 million
- Checks for pre-conditions for C, C++ code
- **Apple** and Microsoft use this to check their software!
- Trace the program
- When you see the trace, you can see whether the trace is valid and check the traces
- Debuggers
- Probably NOT the best way to test efficiently!
- You would spend hours and hours chasing that bug.
- There are better uses of your time

Cost Models
Mental model of how your languages implementation behaves

- The reason you want this mental model is because you want to understand how the system will operate so you can write efficient code compared to your competition.

- (append ‘(a) b) is cheap and will run really fast all the time
- (append a ‘(b)) is expensive and depends on the size of a

What costs?

- CPU time
- RAM - how much memory does your application take?
- Cache - how much cache does your computation take?
- Can you fit a big computation in a small cache? This will heavily **optimize** your code
- network access
- Turning into a huge deal nowadays
- Often #1 these days
- Power consumption
- Embedded applications prefer solutions that are lower powered over things like greater CPU times
- Modern CPUs have different cores running different power levels
- Whichever guy has the most variables will be the smaller of the two

Here’s how we can speed it up

- Change number to be 91 because it is a prime number
- Do NOT use a power of 2 OR a non-prime number
- From memory’s point of view, they are staggered and your program will run ten times faster than before because you have a cost model of array access that works
- Suppose you do this in Java, will the same lessons still apply?
- Java doesn’t have 2D arrays
- It just has arrays of pointers to arrays
- This kind of caching thing is NOT something you have control of in Java
- You need a cost model in order to understand how to make programs go faster
- Whatever model you have in your head is probably 25% wrong.
- Don’t just trust the stuff I have said here and it could be wrong for your implementation/

Please finish reading the book!

- Next lecture will cover semantics and the history of programming languages

W 10 R Lec 3-16-17

- There is one aspect of C and C++ where register does have a meaning.
- **Declare the variable like this and if you say &i, then the program is invalid!**

- You cannot take the address of a variable that's been declared as register

&x leads to aliasing issues.
- Can make code slower
- Can make code harder to read

Inline keyword
- Any restriction between C and C++ for inline functions?
- **Cannot be recursive!**
- You can actually write recursive inline functions in C
- Inline is advice to compiler that it can ignore!
- Is there a semantics of inline that affects the meaning of the program?
- **Purely performance advice!**
- You can work around it!

Meaning: The heart of the program language!

PL = syntax + semantics
- Syntax is what programs look like

Q. What if you try to say register if you have no registers left?
A. C standard has 8 registers and we could see if it was pretty much obsolete.

inline is a sign of weakness for current compilation techniques in C/C++, Java
- In 20 years, hopefully we can figure out and make **inline** keyword obsolete

Syntax is a solved problem
- Like to argue about it, but we have BNF, CFGs, RE, etc.
- CS 181 material solved this problem
- Takes some engineering work to get it right and you need to think about it for performance
- **However, this is NOT the hard part of Computer Science**

Semantics - hard part of the Computer Science
- We can divide this into two major subcomponents: (1) static semantics and (2) dynamic semantics
- **Static semantics**: rules of the language you can find out before it runs
- Example of things you specify in static semantics are type checking in Java
- **Dynamic semantics** (array accesses): these things are obvious to programmers but you have to nail down all the details
- Lots of different techniques that have been developed over the years for dynamic semantics

- attribute grammars: write down a grammar and next to act grammar rule, you place extra constraints on type checking and scope checking
- These rules operate whenever that grammar rule fires
- Type checking on a very simple language with arithmetic expressions and just two types (int and float)
- We have a grammar that looks as follows

Add attribute rules to our attribute equations

- Put little subscripts on our nonterminals to tell them apart from each other
- These let us add little things about attributes, and attributes will say what is the type of this expression
- The type of E_1 will be computed from the types of E_2 and E_3

Let's say we have a type attribute and we have a right attribute that we use to do the static semantics for scope checking

- Dictionary - specifies all the identifiers visible at this node and for each identifier, what type it is
- Add extra attribute rules and NOT only is the type of E_1 but there is another rule that the dictionary E_2 = dictionary E_1 and dictionary E_3 = dictionary E_1
- **These rules start to add up!**

Inherited attribute

- Attributes can flow bi-directionally!
- NOT limited from parent -> child

block node that inherits a dictionary from its parents

- declarations + statements
- In order to interpret these declarations, you know the dictionary from the parent and you need to know what those are from the environment
- Declarations node will return a revised dictionary which says what new identifiers were added.
- Sometimes it is inherited, and it can be a compromise and then in that case, we just call it inherited.
- Information flow about all the compile-time information about your program and how to interpret it

Suppose you specify x depends on y and y depends on x

- Even in different grammar rules, you won't notice this cycle!
- **Can you detect cycles in these attribute dependencies?**
- Can you just look at a grammar and say it isn't right because it can allow cycles
- **It turns out you CAN detect cycles**
- You don't have to run the compiler to find bugs in your grammar

- Some people right programs in this way and people write attribute grammar processors that generate C code and the compiler will then check your programs out that way.

Static semantics are becoming more and more complicated because Java and OCaml have generics

- This is sort of a solved problem because once you have written the static semantics for a language, you have sort of put it to rest.

This turns into a hairy problem pretty quickly!

- When people have thought about dynamic semantics, they have come up with three major approaches

dynamic semantics

0. operational semantics: If you want to say what a program means when you run it, you explain it by writing an interpreter for it
- How a program behaves by illustrating it with another lower-level, simpler program to see how to run a bigger program!
- Explain program P in a language L via an interpreter
- Stealing ideas from imperative languages and this is what operational semantics are!
- To see what a program means, you run the interpreter

0. axiomatic semantics
- Explain p in a language L by providing general axioms and rules of inference about L and then derive a conclusion from it
- **Supply axioms and rules of inference!**
- If someone wants to know what P means, then ask to be more specific about these rules.
- If I give it inputs 5 and 7, will it output 12?
- That is a specific logical question you can ask about the program.
- Apply these axioms for individual statements and applying these rules of inference
- Don't have to rely on understanding an interpreter!
- Just think about the program
- **More abstract approach and to some extent, more satisfying approach**
- If you like logic based languages, then you will probably like axiomatic semantics

0. denotational semantics
- Supply a function from program to meanings
- Given a program, call a function that will output the meaning of a program
- **Typically, these meanings are functions!**
- If you have a program P, and you want to know what it means, call "meaning P"
- Take that function and try it out on your particular test case!
- Looks like functional languages

- Sometimes, the distinction between these three major camps may NOT be as strict as we like
- There can be a combination of approaches that are NOT mutually exclusive!
- No need to insist on one technique over the others exclusively!

operational semantics for Lisp

- The basic idea here is that we will write a Lisp interpreter
- You can predict how any Lisp interpreter works through Fortran
- There is an assumption here that you need to have a base language, but once you know one programming language under this approach, you can learn how any other programming language works!
- They didn't actually write the Lisp interpreter in Fortran
- They actually wrote it in Lisp

Suppose some very smart person at MIT has written a Lisp implementation

- You now want to understand how LISP works and you read the source code of it and run it on MIT implementation which was written in machine code.
- Once you build the first Lisp system, you will understand how it works
- Look at first element of list, if it is the keyword 'let', do what you normally do with 'let'
- Code that implements Lisp but the interpreter is written in Lisp!
- Interpreting lists in Lisp is easier because programs are always valid data and this Lisp interpreter was quite short.

English language is defined in the Oxford English dictionary of course!

- English is defined in a circular way and we haven't fallen apart yet.
- This approach can be a very good way of defining a language!

Give a feeling of how axiomatic semantics of ML might work

- Adding two numbers
- The way we are going to say that is that I am going to show the result of adding two numbers, which yields a value
- In order to figure out the values of these variables, I need to know context in which these variables occur
- Map variables to values
- The way you write an inference rule in logic is that if you accept these parentheses, then you can conclude this result
- Analogous to Prolog
- If you read this part of the book and the semantics of MML, this + is in a slightly different font
- We can write another rule for * and - as well!

How do we evaluate a constant in any context?

- Just get the constant!

Let expressions

- Prove that E_1 yields a value in this context!
- E_2 has a change in that X_1 is now bound to V_1
- **If you can prove all the base rules, you can draw a final conclusion!**

Functions

- Kind of an easy question because we don't have to evaluate E
- Say that function expression in this context yields an expression
- Instead of having an inference rule, we will just have a fact
- This lambda here is just a constraint, which has X and E as its components
- What Eggert made was an ML interpreter written in Prolog
- In order to evaluate this and get that as a result, the result of evaluating a call from E_1, E_2 in the context C gives the result V
- Translate the bottom part of this rule into the head of the clause of my program
- Call with function, argument and comma in between
- A little more verbose but that is okay

- This has to yield V in the end
- Translate the mathematical notation into Prolog code and this is the kind of code we can actually run!

eval(V, C, R) :- member(V = R, C).

- Look into name-value pairs for the name that you want and grab the resulting values

Prolog is a natural match for doing axiomatic semantics

- Now that we have seen the semantics, we can answer questions about our language
- Does this semantic specify dynamic scoping or static scoping?
- Static: When you look at a function definition, you know what defines every variable.
- Dynamic: Look at caller, caller's caller…
- Look at context here and what context are you using here?
- Suppose X isn't the local variable of this function and you are pulling it out of the context
- You get it from the call!
- If I haven't defined the variable, look at the caller's context and figure it out!
- These semantics are for dynamic-scoping, NOT static scoping!
- If we want to use static scoping, we will have to alter the semantics
- We forgot what the context was because we forgot the context of where the function was defined

Notice that this order of evaluation question comes up in ML when there is only one argument for every function

- When you execute this expression, if you E_1 and E_2 both have side effects, which expression evaluates first?

Axiomatic semantics to imperative code

- Imperative code is where most of the world works with
- JavaScript, Java, etc.
- Axiomatic semantics can help you write more reliable and trustworthy imperative code

Give a general rule for assignment statements

- One in which we can have any expression here and this could be any variable i.e. x = E;

Prove a variant of Q which holds before the assignment statement

For thinking about the code you write, keep the pre-conditions and post-conditions in mind and this will help it be more reliable!

History of Programming Languages

- Fortran
- Language in Hidden Figures - one of the heroes in that story learns Fortran
- Major contribution to programming languages was to prove that it was possible
- Before Fortran, having a high-level language to do arithmetic operations was considered AI!
- Asking the computer to write your program for you
- Fortran proved that languages are doable and easy!
- IBM developers wasn't told that what they were to do was **impossible**!
- Contributed loops, arrays, and subroutines!
- Still stuff we use today
- Took over so much that IBM's competitors were alarmed!
- Manual written in English but if you wanted to build another Fortran system, you had to make sure it ran on IBM's systems!

People wanted independent language from Fortran

- Algol 60
- Basically fixed the problem of grammars by introducing the notion of BNF
- First language that had a formal syntax
- Enormously influential!
- Had a few other features as well
- Call-by-name
- Recursion
- Innovative in how it formalized things

People writing business applications did NOT want to write Algol, so they defined their own language

- **COBOL** ~ 1961
- Designed by a committee led by **Grace Hopper**
- Records/struct
- Heterogeneous data structures of different things
- IBM saw this diversity of languages as being a problem

IBM wanted to recapture the market

- PL/I 1964 - 1968 (when it finished)
- Original prototype of the kitchen sink
- Wide spectrum language that could do anything

- BCPL 1968
- Simple low-level

Combined syntax of PL/I and BCPL to create C

- C 1971
- + pointers and types

Simula67 - 1967

- Objects

Smalltalk 1975

C++ 1980

Java takes the ideas of Smalltalk and merges in the semantics of C++ (1994)

C# (2001)

Lisp

- S-expr
- Recursion

Scheme (1972)

Basic (1961)

- Fortran for dummies!
- Proved enormously influential!
- Visual Basic is widely used
- Python is thought of scripting for dummies!

Final Exam is next week on Tuesday - open book, open notes

- Over the whole course but emphasis is on the 2nd half of the course (60/40 or 66/33 split)

W 10 Dis 3-17-17
Final

- Tuesday March 21st from 11:30 AM - 2:30 PM
- Open books, open notes
- More emphasis on 2nd half of class
- After midterm, not OCaml
- But also first half
- More like 1/3 to 2/3 ratio

There will questions about hw

- Print that stuff out!
- Discuss whatever problems you had doing the hw

OCaml Warmup

- Have a type *tree* with a list of trees
- Don't stick with a tuple

List.map

- Takes a function and activates it on every element in a list

Which of these are advantages of throwing an exception over simply returning a distinguished error value (e.g. null)?

- What could be a good answer for this?
- Exceptions can contain additional information about the error that occurred.

Q. Inheritance without sub typing is best described as:
A. ii. A form of code reuse.

- Interfaces are a form of sub typing!

Q. Subtyping without inheritance is:
A. ii. Expressible in Java by a class implementing an interface.

Scheme

- cond -> expands to if then else in obvious way
- (1) -> not an expression! not an valid expression in Scheme

Q. What if each object in an array must have the same size?
A. Save space because you don't have to align them.

- To do random access, you add a constant number to a pointer address!
- If they are different sizes, you do NOT have this luxury
- Can help quite a bit and this speeds things up

- The arrays must have just one dimension
  - Kind of an advantage because we don't have to support 2D subscripting

- Arrays must always be directly accessed; there are no pointers to arrays, and arrays cannot be passed by reference to procedures
  - Advantage is there is NO aliasing, but slow as hell!

W 10 Dis (Theresa) 3-17-17

- Python project will be started to graded now
- Professor Eggert writes a new final every iteration
- The final is cumulative (if you forgot everything you did for the midterm, it is a good time to refresh)
- Stuff after the midterm will e emphasized
- The setup is the same as the midterm
- Points are given in terms of the number of minutes per question
- 180 minutes

Practice Final

0. Why is heap fragmentation not a problem in a traditional Lisp implementation where "cons" is the only heap storage allocation primitive?
   - Allocate large blocks that don't align.

Q. What is a stack?
A. When functions get called, they are pushed onto a stack. First in, last out.

- When you have a function, you push its current scope onto the stack and then you pop it off later.
- Returns and you move on to the next thing in the stack.

Q. What is a heap?
A. No order to it, but just a giant chunk of memory that is allocated at runtime.

- Stack has a set size, but it is faster
- The only limit to your heap is your virtual memory size.
- Think of it as a heap with a bunch of crap on top of it

Q. When would you want to use a heap?
A. Scope needs to outlast the function it was declared it.

- Even after the function has returned, there is something you want to keep in memory

Heap fragmentation: If there is space between, it is free but you cannot really use it if you need a block bigger than that free size.

- Free blocks peppered in won't work in those modules
- Disk fragmentation: Defrag your disk once in a while because it is a pain in the ass!
- Same idea for the heap
- You think you might need this much and you end up using much less than that.

- NOT used for anything so nothing else can claim it!

You can cons anything as long as Lisp accepts it

- It looks like you can cons something that is huge with something that is little

Q. Why are cons cells the same size?
A. There is NO internal and external fragmentation

- In order to put something into a little piece of memory, it has to be an associated value with it (?)

We can guess this object will never be too large!

0. OCaml question

- This part doesn't require you to know how the function works to figure out the types of these things
- Do some type checking practice in the interpreter
- fpr is an 'a list

2.b. Find prime numbers

- 1 is a prime - keep it
- 2 is a prime - keep it
- Cross out every second number
- 3 is a prime number, so then cross every multiple of 3
- Keep going until you reach the next number (5)
- Sieve of Erasthothenes (Greek guy we learned about in like 3rd grade)

0. Part of Twisted assignment was dealing with waiting. If Google was hanging for some reason, then this was an issue I would have run across.

- callback: a function that calls when something completes
- More of what a callback does
- What happens if I physically put the word callback into the function?
- If something else returns, then that function is completed.
- How does the function know is returning and what should be completed?
- What is a callback and what does it do to that function?
- Sort of like a break point.

Q. How does Twisted handle callbacks?
A. Creates a new Differed object that returns.

- When you return a struct in C or your own personal class, it returns with all your personal information.
- Because it has been returned, you don't get hangs.
- Callbacks must be finished ASAP - a function returns.

What does a function local variable mean?

- **A local variable inside the function!**

- How do callbacks work outside of Twisted?
- When you hit a callback, you have to save the current state.
- You save everything and because the function never returns anything, the function is always on pause while the stuff the callback waiting for finishes.
- Because callbacks don't want any hanging in Twisted whatsoever, we compress it and return it as an object.
- We are no longer in function local variables and this makes sense.
- You cannot keep persistent state in multithreaded applications because you cannot block threads.

- **Related to continuations**!
- Not necessarily a function,
- Each callback must finish as soon as possible:

0. Why do we no longer use the first major object-oriented programming language (Simula 67)
- New languages are released and the older languages lose support.
- Why does that NOT happen with functional, logic, and imperative languages
- The new things we can do in these languages are more powerful than the old ones!
- Updated the language, why did these get updated without dying?
- Simula 67 was overtaken by newer language

Be able to read code and figure out how it works
- Scheme - actually only has a tiny set of syntax and everything is built out of this by creating macros