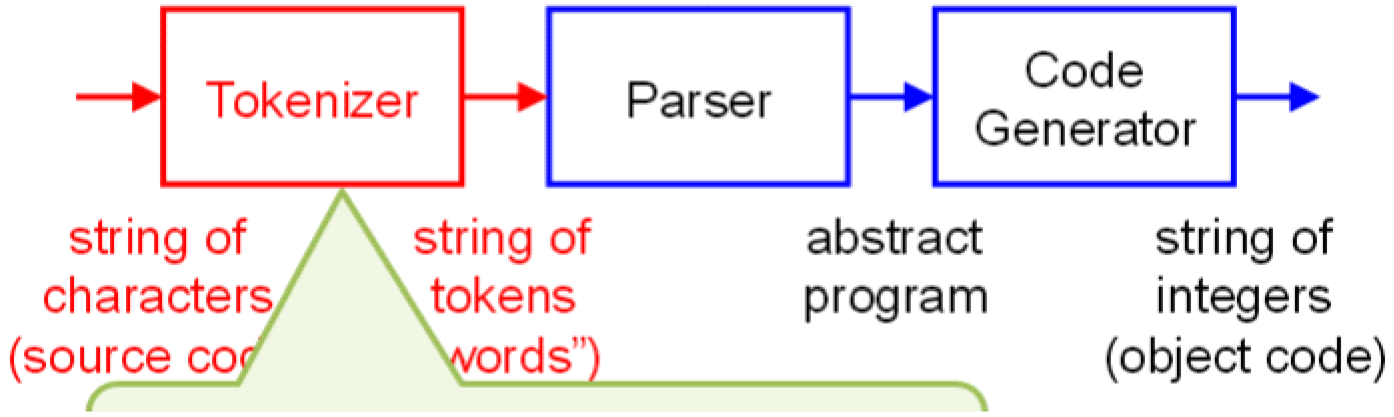
Final Review 2 AM Fuck Me

Wednesday, December 16, 2015 2:07 AM

BL Compiler Structure:


Tokenizer
Parser
Code Generator
string of characters (source code)
string of tokens ("words")
abstract program
string of integers (object code)

12/16/2015 2:57 AM - Screen Clipping

The Tokenizer's job is to take the source code, which is a string of characters, and turn it into a string of tokens that can be individually processed by the Parser. The BL tokenizer just grabs every word typed into the editor of BL. It then passes this ordered string of tokens into the parser.

The Parser reads the string of tokens and turns it into an abstract program. It does this by using the rules of the Context Free Grammar of the language, and recursively sifts through the code putting it into an appropriate format for the Code Generator to read. The way our BL parser works is it turns the tokens into a Program component, composed of smaller statement components for the user defined instructions contained in the context, and the body portion of the code.

Recursive Descent Parsing:
The idea behind RDP is to continually decide how to parse your code. It really depends on your CFG. When you see a {, it means a while loop is going to come into effect, as when you recurse into your statement, you're removing everything contained within a while loop, if statement, or just a singular call. All three of these are nonterminal symbols in the BL CFG. The top of any block of code is actually a block nonterminal symbol. When a | is featured, it means OR or IF ELSE, because the nonterminal symbol can mean anything in that nonterminal symbol definition.

Here's a look at how a RDP would be written:

- Every nonterminal symbol gets its own method. That way, you can recurse down the actual tree that represents the CFG. You have to start at a really unclear symbol, as it sends you down many paths. This is a block in the BL CFG, which can send you to an IF, WHILE, or call statement. You would use this selection to predict the next word that would come out of your string of tokens and handle it appropriately.
- When you hit a terminal symbol, like a call identifier, which would happen when looking at a call, you would remove that symbol permanently from the string of tokens. This is typically the end of your recursion.
- The CFG can actually bring you back up to the top, so you have to evaluate an entirely new structure contained within the body of an IF or WHILE. This makes sense, because there can be

nested loops.

- With IF statements, we also have to be wary of ELSE appearing. You would check for this AFTER you recurse through the block of code contained within. If ELSE is your token in front, you know what kind of structure to add to your statement object, which is an if else, which means a statement comprised of the condition of the if, the body of the if, and the body of the else, all obtained from a premade method that handles the language.
- Parsing means different things. In BL, it means putting the code into a format (a Program object) so that our Code Generator can turn it into Object Code. For an expression evaluator, it means finding the value of the expression. Different purposes require different code.

Code Generation:

The purpose of the code generator is to read a abstract program, parsed in from the parser and to turn it into object code. BL does this by reading in from a Program using Statement objects, and turning it into BL object code, which is just a file with a number on each line. You don't need to know these numbers, just how to output them to a file based on the values found in Statement objects.

There are two types of numbers, or instruction set:

- Primitive instructions, which are premade and selected based on certain conditions and calls in your Statements.
- Jump calls, which are GOTOs to different lines of code. These fall after certain places in your code:
    - After an IF, if there is no else, jumps to the end of the block inside of the IF. If there is an ELSE, jumps to the number that evaluates the condition of the ELSE. After the ELSE, there is a number that points to the end of the ELSE block.
    - After a WHILE, it points to the end of the block. Right before the end of the block, another jump statement is created, that points back to the condition so it can be reevaluated again.
    - After a CALL, two things can happen. If it is a premade call that comes with the VM, we just use the preset number. If it is a user defined method, found as a statement in the context, we print out that entire thing using the method we are writing into the object code.
    - For a BLOCK, we just scroll through all the children of the block in an ordered fashion, and put them into the object code in order of first appearance, then put them back.
    - For all of these cases, its hard to know when the end of the object code will be. We edit the jump GOTO numbers last by using a dummy variable. It's easy to do this with the OSU sequence component.

Interfaces:

- These are used to specify what needs to be done in a class. Clients can read the interface to see what the class does, as well as what the individual methods do.
- Classes that *implement* an interface must provide method bodies for every method declared in that interface.
- The class level implementations describe how to do what the interface says it does. This is through individual contracts for the methods and the program code that makes them operational.
- An interface can extend other interfaces, which makes classes that implement the interface responsible for implementing multiple interfaces.
- Interfaces can only define constants (static final variables) and instance methods of an object or class.
- For our class, the kernel interface defines the math model for the type, the contracts for the constructors, the contracts for the kernel methods, and the contracts that are inherited from Java library interfaces that OSU interfaces extend.
- Kernel methods should be a minimum set of methods that do everything that the object needs to do, involving giving a variable of the type any allowable value, and finding the value of a variable

of the type. It also needs to allow a client to implement equals and toString, and to check every kernel method's precondition.
- Interfaces are compile time constructs, and are used by Java for type checking declared objects to ensure there are no compilation errors.
- A package is a grouping of interfaces and classes that belong together for logical reasons. Every single OSU component is located in the OSU package. Also, units from 2 different packages with the same name can be used simultaneously. So, a Java Queue and an OSU Queue can be used at the same time.

Equals, toString, hashCode:
- Each class in Java extends Object, which has the three methods Equals, toString, and hashCode. Every class you write should override these, unless absolutely unnecessary.
- We would implement these in a secondary abstract class, so as to ensure that different implementations of the same object type would be able to be compared and hashed the same. Don't use anything from the different kernel implementation when making these, use the actual kernel methods themselves.
- To check for equals, first, check to see if they are aliases, and if they are, they are equal. Next, if one of the objects is null, then theres no way they can be equal, check for that. Finally, make sure that both objects come from the same interface. If they do not, they are not equal obviously. To do this, use the object name *instanceof* interfaceName.
- If you try to compare two objects that take generic type, Java presents a problem: It only compares the raw type: ie the interface they both implement, not the generic type, which may be different for two objects with the same type. This is called type erasure.
- To remedy this, you need to iterate through the object (if this is possible) and ensure that the data they hold is identical. If they're both empty you are fucked. It's a Java problem.
- toString is simple man, its based on your object.
- hashCode should run in constant time, or the lowest asymptotic time since it's only use is for speed.

Java Collections:
- Basically operate like OSU's components with some slight differences.
- Everything implements Collection<E>, which is a finite multiset of E.
- The Set is like OSU, but there's no removeAny, so you have to use an iterator and a forEach loop. (Which you cannot modify in the loop.)
- List<E> is like Sequence, but most sane people use ArrayList<E>.
- Queue<E> in which add enqueues and remove dequeues.
- A view is a collection within a collection. The only views are in Map, which are a set of Keys from keySet(), a set of Pairs called the entrySet, and Pairs are called Entries.
- When something is backed by the underlying collection, this means if you edit the underlying collection, you edit the collection it came from. EX: EntrySet().
- You can only iterate over the entrySet, keySet, and Collection of values from a map.
- HashSet uses hashing to find stuff in a set. Do not use clone.
- TreeSet uses a binary search tree.
- Every collection has an abstract class that it extends, called AbstractCollectionName
- ArrayList uses arrays to implement a list.
- LinkedList uses doubly linked lists.
- HashMap uses hashing.
- TreeMAp binary blah
- Many JCF objects have useful algorithms that sort, reverse the order of, and do a bunch of other stuff with the collections.
- The JCF does not use formal contracts to describe its objects and methods.
- They also do not have the parameter modes clears restores replaces and updates.
- They allow aliasing. They also allow null to be stored.

- JCF interfaces have optional methods, which mean that classes that implement them do not have to implement the optional methods.
- They have 2 constructors, no arguments and conversion constructors, which copies the references to elements of the argument, meaning it copies the elements from another collection to itself.
- Exceptions can be thrown.
- There are no kernel methods, instead with all methods described in one interface.

Java I/O

- An I/O stream is a series of data items. An input stream is a flow of data items from a source to the program you are writing (read from source). An output stream is a flow of a data items from the program you are writing to a destination (writing to a file or console).
- You have to use try and catch to handle errors that might arise from Java IO.
- To check for end of file/stream, you have to ensure that the strings you get from the objects aren't null, if they are, then you're at end of file.
- The reader and writer classes are independent of sources and destinations, and can change them on the fly.
- Standard stream: System.in, System.out, and System.err. These correspond to console input, console output, and console error output.
- Byte streams: Streams of 8bit bytes. Who cares.
- Character Streams are streams of Unicode characters, adapted from 8bit bytes. This is why we wrap a standard stream inside of a byte stream inside of a character stream.
- A buffered stream is a stream that increases performance, and allows for string creation. We wrap character streams in buffered streams.

Java Loose Ends:

- Classes have different types of members, which can be static or instance members. A static member does not change between instance variables of a class. It remains the same for every object created from that class's constructor. Only one copy will be created at runtime. Instance members are edited and changed with each new implementations, and multiple may exist at runtime.
- An inner class like Pair is declared within a class, so that the outer class can use it. A static inner class cannot use shit from the outer class.
- Access Modifiers: Public, Private, Protected, and Package Private.
- Public members can be used anywhere at anytime as long as the class they are contained in is in a package that is imported.
- Package private is the default access when you do not specify anything, and can be accessed by anything in the same package as it.
- Protected class members are accessible from subclasses and anywhere in the same package.
- Private class members can only be used within the same class. All static and instance variables should be private, and you should make methods to change and access them.
- A class declared final cannot be extended.
- A method declared final cannot be overridden.
- A variable declared final cannot be given a new value, once it has one.

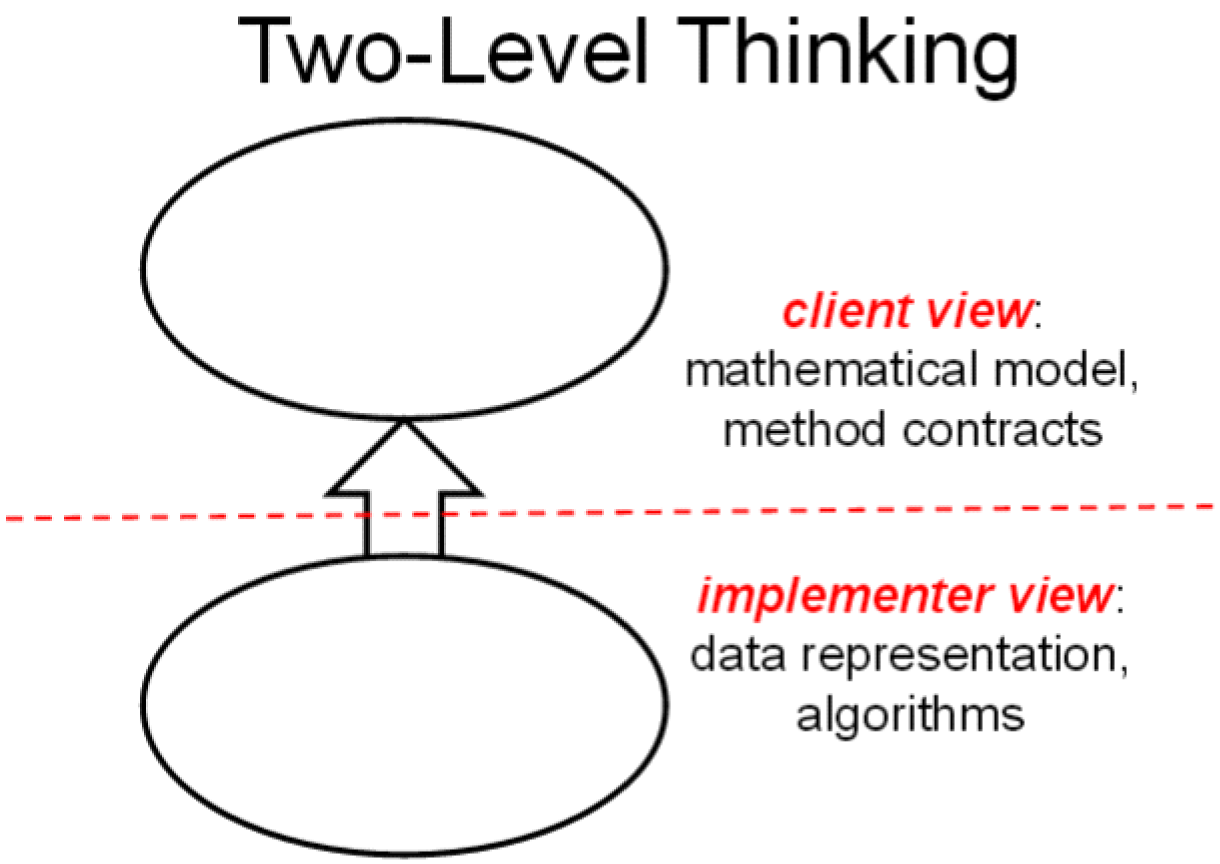
Abstract Classes:

- An abstract class allows you to write a partial class that contains bodies for some but typically not all of the methods it claims to implement.
- This allows for methods to be written that work for any implementation of any interface, so as to factor out common code that would appear in every implementation. This is a best practice.

Kernel Implementations

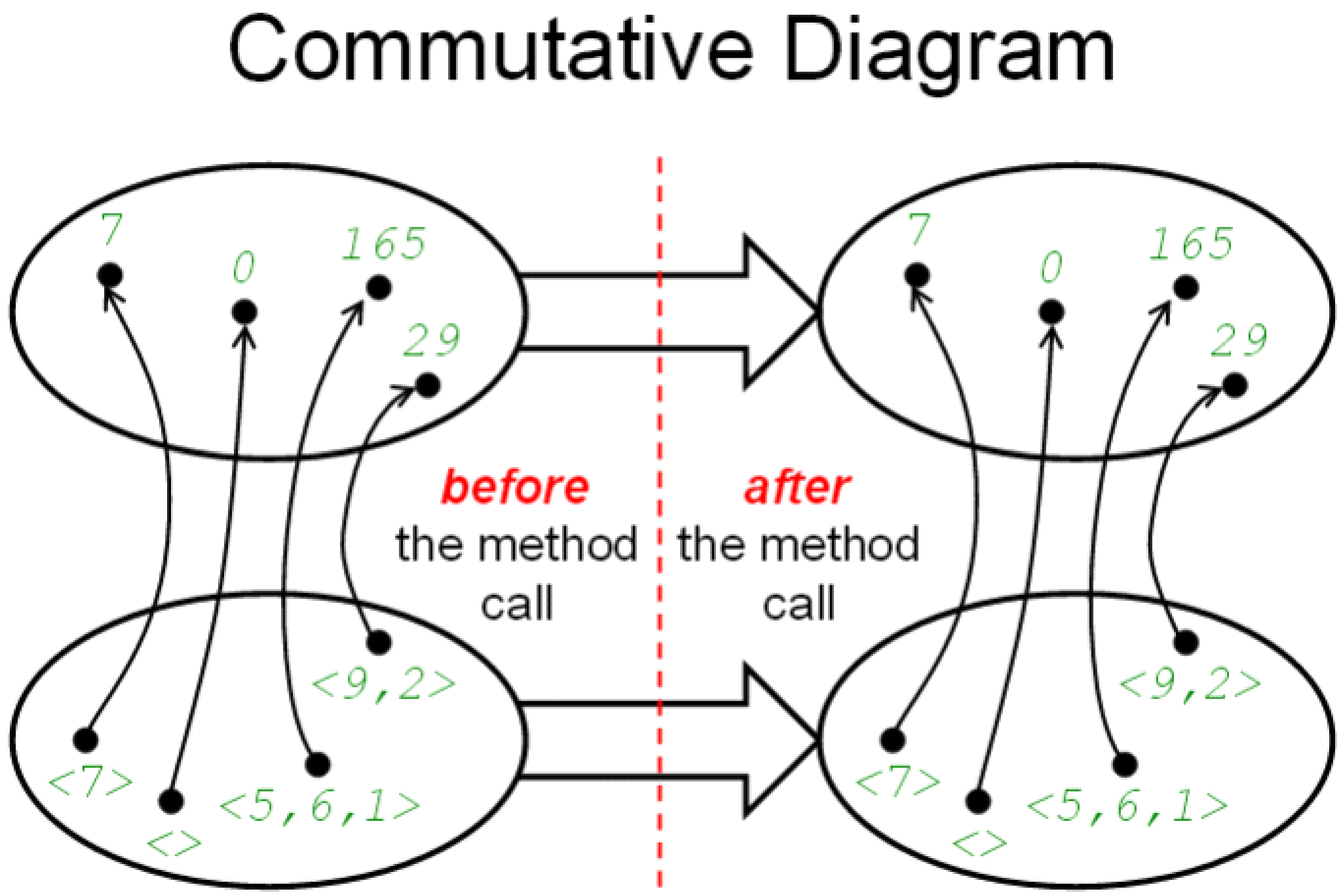
- We can look at configurations of low level ideas to build a higher level idea, or to interpret them as one.

- When implementing OSU kernel methods, what low level data structure can you use to create a representation of another data structure? How can you interpret that data structure as something else? By coding it that way.
- An instance variable is a variable that can be different for different constructors of an object, EX one List1 object's name instance variable is Steve, anothers is Fuck.
- Kernel implementations should be easy to understand, efficient, and work right.
- A client only sees the object you have build, an implementer sees how you built it.



12/16/2015 6:08 AM - Screen Clipping

- The abstract state space is the top circle, it shows the set of all possible math model values as seen by a client. "What can be done with your object".
- The concrete state space is the bottom circle, it is the set of all possible math model values of the data representation. "What I use to do things with my object".
- The line in the middle is the interpretation of each concrete value as an abstract value, or how we see the data structure corresponds to what the client sees when they use your object.
- The commutative diagram is a way to thing about how this diagram is affected before and after a method call.



12/16/2015 6:12 AM - Screen Clipping

- The top arrow is the abstract transition, where each state of the object before the call might end up because of a call.
- The bottom arrow is the concrete transition, where each state of the data structure you are using might end up based on the body of code you have written based on your implementation.
- A class is correct if for every legal input state for a method, the method body always results in a state that satisfies the method contract. This means that the client gets what they expect(up and then over), and your code does what you expect (over then up).
- The representation invariant tells us which configurations of values of the instance variables can ever arise. This describes what values can even exist in concrete space. This is called the convention. Constructors of the class must make the convention true. It can be assumed to be true at the start of each method body, and each body must make it true again at the end of themselves.
- The abstraction function describes how we interpret the values in the concrete state space to get the values in the abstract state space. This is called the correspondence.
- Kernel Purity Rule: No method body in the kernel class should call any public method from the same component family. This means no List1 calling List2 methods.

Sorting, Sorting Machine:

- A SortingMachine is a component to allow for easier abstraction of the sorting of items.
- The best way to think about it is as a machine with a large funnel on top of it. You pass into this funnel a finite multiset of parameters *T*, and a comparator that can compare those values of *T*.
- SortingMachine has two modes, insertion and extraction. You cannot add new items when the

machine is in extraction mode. In insertion mode, you add a set of items in. When you are done, you change it to extraction mode, and remove every item one by one, one at a time. The set is sorted when you change the machine to extraction mode.
- The standard Java .sort() method sucks the penis. Do not use it. (It actually rules but this is what CSE 2231 wants you to think).

Sorting Machine Methods:
- The constructor has one parameter, which is the Comparator object *order.* This is what the sorting algorithm housed inside the machine will use to compare two values.
- The math model of SortingMachine can be explained by three attributes, and these are *insertion_mode*, which is a boolean, *ordering*, which is a binary relation (a relationship between two objects) for two objects in the machine, and *contents*, which is a finite multiset of variables. The contents are what we are going to sort using the comparator order.
- Void add(T x) - Adds a variable x of type T to the machine. Can only be done when the machine is in insertion mode. Updates the *contents* attribute.
- Void changeToExtractionMode - Updates the *insertion_mode* attribute to false. This prohibits a sorting machine from adding more variables to *contents*, and usually sorts the variables in *contents* according to *ordering*.
- T removeFirst() - Removes the first sorted element from *contents*. This can only be done when *insertion_mode* is false.
- Boolean isInInsertionMode()- Returns value of *insertion_mode.*
- Int size() - Returns size of *content*
- Comparator<T> order()- Returns the Comparator object that is described by *ordering*.

Methods of Sorting:

Quicksort
- ○ This algorithm is a way to sort a data set recursively, and was penned by some guy who was sexually attracted to cars or something at a talk in like 15 minutes. It's the best!

  It starts like this:
1. Choose a pivot from the data set. Wikipedia says that the last number is a poor choice in the case of an already sorted data set, so let us go with the middle.
2. In a process called *partitioning*, Put every element of the set smaller than the pivot on the left side of it, and every element of the set larger than it on the right side.
3. Then, recursively call the above operations on the left partition (not including the partition element) and the right partition (not including the partition element).
4. The base case for this recursion is when the size of the array we are sorting is when the set size is zero or one.

Heapsort
- This algorithm will always run O(nlog(n)). It's used when you need to securely sort big sets of data, as Quicksort can have a worse running time.
- We can use a "Heap" to return elements from a set in a sorted order. First, let's try to understand what a heap is:
  - ▪ It is a Binary Tree that is complete, as all levels of the tree are "filled", or have the maximum amount of nodes. This is not the case for the bottom level of the tree. The rightmost node MUST have every single node that can exist before it to exist. Otherwise, we say the heap structure is violated.
  - ▪ It has the **local ordering property**, which states that every child is larger than its parent.
- The method of "heapsorting" can be drawn in parallel to SortingMachine. In changeToExtractionMode, we build the heap, making sure that the two attributes are

present above. In removeFirst, we remove the root of the heap binary tree, and fix the now godless "non-heap" creation so that it can once again be a heap.

- We worked with a method called Heapify, which recursively made the entire binary tree a heap. The actual function of the program is this: If the root is the only node left, this is the base case and we don't have to worry anymore. If it's not, then we heapify both the child subtrees if they exist. We would then use the next method to ensure the right element is in the root. This is what is done in changeToExtractionMode.
- We also worked with a method called SiftDown. This finds the ideal candidate (according to the specified order of the heap) and fixes the root so the elements in the set will be returned in the correct order. It then recursively fixes the heap so it still has heap ordering.
- **Anna said we probably wouldn't have to implement Heapsort in code on the exam, but we would have to be able to work with heaps and understand the process.**

Linked/Doubly-Linked Lists:

We're first going to talk about why we use lists, as opposed to Arrays.

- Arrays are of a fixed size, as they cannot add on more elements after they are initialized. However, they can be directly accessed, which means that providing an int location allows for instant obtainment of an element in the collection.
- However, we might run out of room in the array, as it is not **dynamically sized**, which means here that it can have no limit on its size and can be incrementally adjusted by adding/removing things in the middle. To add/remove things in the middle of the array, we have to move all elements of the array which is inefficient. Initializing an array is also expensive.
- Lists are objects that allow for a dynamic collection to be accessed quickly through sequential access, which is going next, next, next through the collection.

- From Anna's mouth to our ears about eight times over
  - ***A list is a pair of Strings.***
  - ***It's concrete representation is (<...>, <...>)***
- The big idea with our implementations of linked lists relies heavily on *pointers*. A *node* in a linked list is a collection of pointers--one of which points to the node's data value and another that points to the next node in succession.
  - In a doubly-linked list, there also exists another pointer that points to the previous node to allow for retreat() to be possible. This pointer is called
- In lists, we use what are known as *Dummy* or *Sentinel Nodes*, which exist to make our lives easier and (in some cases) our algorithms faster.
  - Singly-Linked Lists only contain one Sentinel node, located at the front of the list. It will never hold a data value, but it exists as the non-null basis for our list (so that we can build off of it). It has a next pointer that points to the first node that actually contains data.
  - Doubly-Linked Lists contain two Sentinel Nodes--the preStart and postFinish nodes. They bookend whatever list you create and allow for easier node management. They will never have to be adjusted once they're established within the constructor.
- To refer to a specific object OBJ in a Linked-List, create a pointer (or two) to some node with OBJ dat in it. If you want the object directly after some other object, reassign the pointer of the original object's .next pointer to the node containing the data OBJ. If it's a doubly linked list, you would have to take what the original object's .next pointer pointed at, and set its previous to the node containing OBJ. The node containin OBJ's pointers would be .next = old .next, and .prev = the original node we were working with.

A List3 has a couple of interesting pointer objects. These are:

- This.preStart: Points to the first, leftmost sentinel node.
- This.lastLeft: Points to the very last node contained in what we are abstractedly referring to as the "left string". This is useful, as we only work with the beginning of the right string in doubly linked lists, for efficiency.
- This.postFinish: Refers to the last, rightmost node, which is another sentinel node that points to the last element in the list.
- This.leftLength: Reports length of "left string".
- This.rightLength: Reports length of "right string".

Linked List Methods:

- addRightFront(T x):
    - Adds x to the beginning of the right String
- T removeRightFront():
    - Removes the first element from the right String
- advance():
    - Moves the marker forward one position
- moveToStart():
    - Move the marker to the beginning of the List
- int leftLength():
    - Returns the length of the left String
- int rightLength():
    - Returns the length of the right String
- T rightFront()
    - Reports the front of the right portion of the String without removing it
- retreat()
    - Moves the marker back one space, allowing for better maneuverability within the list (better implemented within Doubly-Linked Lists)
- moveToFinish()
    - Moves the marker to the end of the list

Implementing Iterators:

- A good way to think about an iterator object is that it serves one purpose. If we have a string collection of objects, it divides it into two strings, this.left and this.right. Left contains everything we have seen and gotten, right contains everything we haven't.
- When you are working with an iterator (using a Java for each loop), remember that you cannot call methods on the collection you are iterating over and you cannot change the values of the elements you retrieve from the collection.
- In implementing an iterator, we should know what a for each loop actually does, because this is what we are trying to build. It is basically a while loop, that examines if there are any more elements in this.right, and if there are, allows access to the next element in the this.right string.
- To do this, we need two methods, boolean hasNext() and T next(). The first returns true if this.right >0. The second takes the next unseen element in this.right, and puts it in this.left, making it a seen element. It requires that this.right >0.

Mathmatical Tree Theory:

- The arity of a tree is the maximum degree of a tree (the degree of all nodes added together.
- A tree is a structure of zero or mode nodes, that all have a label of some math type T called the **label type**.
- A tree is called **empty** if it has zero nodes.
- A tree's structure is recursive, it has a root node and one or more subtrees contained within a string.
- The math function *compose* is basically like taking a value of label type T, and a string of tree of

type T, and making that value the root node and making the string of subtrees the children trees of the root node we made.
- The size of a tree is the total number of nodes. Denoted |tree name|.
- The height of a tree is the longest simple path that exists if the tree is looked at like a graph. Denoted ht(tree name)
- The labels of a tree is the set whose elements are the labels of the tree. Denoted by labels(tree name).
- An empty tree is also referred to as an *order zero*.

<u>Tree:</u>
The Tree component is well, a tree. The methods are as follows:
- Constructor: Needs a parameterized type, initializes an empty tree.
- Sequence<Tree<T>> newSequenceOfTree() - creates and returns an empty sequence of trees using the type passed to it.
- Void assemble(T root, Sequence<Tree<T>> children)- *Composes* the generic variable root with the children subtrees. All types must be the same. This clears children.
- T disassemble(Sequence<Tree<T>> children)- Disassembles this into the root label, which is returned, and replaces children with all of the subtrees in this. Clears this.
- T root()- Reports root.
- Int size()- Reports size.
- T replaceRoot(T x)- Replaces root of this with x, returns old root.
- Int height()- reports height of this.
- Void addSubtree(int pos, Tree<T> st)- Adds st as position pos in the subtrees of this. Will move the original tree (if one exists at position pos) over to the right pos+1 spaces. Clears st.
- Tree<T> removeSubtree(int pos)- Removes and returns the subtree at positon pos of this.
- Int numberOfSubtrees()- Reports the number of subtrees at the root of this.

<u>Bugs Language (BL):</u>
- BL is a language to run bugs on a grid, that can move, turn, and change other bugs to their distinct type in a process called infecting. We will call these things the Bugs Word Game.
- The game is played on a system consisting of a server, a client, and multiple displays.
- The server keeps track of clients, resolves conflicts, and tracks the state of the world as Bugs run rampant in it.
- The clients are the bugs themselves, which in reality are programs that simulate bug behavior for all creatures of one species. This behavior is what is written in Bugs Language.
- The displays show the current state of the world, and statistics about each client.
- Each BL program has a name, a set of user defined of instructions with magnitude >=0, and a main block of code called the body.
- The BL compiler's main purpose is to take the source code of BL, and transform it into object code, which is a string of integers.
- The compiler is composed of three parts: The Tokenizer, the Parser, and the Code Generator. The Tokenizer takes the strings of characters in the source code, and turns them into a string of tokens, or the words in this case. The Parser will then take these tokens, and create an abstract program that the Code Generator can use. This will be done using Program and Statement. The Code Generator will then take the abstract program, and turn it into the object code, which is a string of integers.

<u>Abstract Syntax Trees:</u>
- An AST is a tree model of an entire program or "program structure". It's nodes are made up of a bunch of statements.
- We call it abstract because some of the characters in the program don't show up in the AST.
- The statements can be instruction calls, primitive calls, if, if then, and while checks. Underneath every boolean check, there is a block. A block is any sequence of zero or more statements nested

in an IF or WHILE construct.
- The AST is how our parser makes our code look to our code generator.
- Every conditional check can be see a root node of a subtree, with its label indicating the kind of check and what boolean it checks.
- Underneath every conditional check is a block node. This is useful, because everything in that block of code is organized in a smart way. The first statement past the conditional check is the first child of the block node. This way, we can access every line in the while check.
- A CALL node indicates the bottom of one path of the tree. Nothing can be done past a call.
- Mathematically defined, each node has a 3 tuple of info, containing the KIND of statement (EG BLOCK, WHILE), the TEST condition (EG true, NEXT_IS_EMPTY), and the call of an instruction (EG infect, move, some user defined instruction). This 3 tuple is called STATEMENT_LABEL.

<u>Enumerations:</u>
- Java enums are constructs that allow the use of meaningful symbolic names that you cannot do arithmetic with.
- We use them in Statement to identify the attributes of the math type representing it: STATEMENT_LABEL

<u>Statement Component:</u>
- The point of the Statement component is to be used by the Program component, which we will educate you on later.
- Working with Statement and Program, we're getting a string of Tokens from our Tokenizer (also explained later) to be able to be read from our Code Generator (which is not on this midterm lol).
- Mainly, this component allows you to manipulate values that are ASTs for BL statements. It has the math model of a tree of STATEMENT_LABEL, which is what we defined our AST as.
- The constructor creates a single node with a BLOCK type, so that things can be added onto it. Up next are the methods.
- Statement.Kind kind() - Reports the enum KIND value that this has.
- Void addToBlock(int pos, Statement s)- Adds statement s at position pos in this when this is a BLOCK statement. Clears s.
- Statement removeFromBlock(int pos)- Removes and returns the statement as pos in BLOCK.
- Int lengthOfBlock()- Reports the number of statements in the BLOCK statement. (Reports the number of children one level below the block node).
- Void assembleIf(Statement.Condition c, Statement s) - Replaces this with a statement with root label (IF, c, ?) and only one subtree the BLOCK s. Clears s. Basically, makes a root node and then puts the s as a block as its child.
- Statement.Condition disassembleIf(Statement s)- Deconstructs the IF statement this into its test Condition and returns it, and passes its only subtree into s. Clears this.
- The rest are the same, but with different ways to handle the TEST condition attribute (if it exists).
- To process a statement, go by its KIND attribute, using a switch statement.

<u>Program Component:</u>
- This component allows you to manipulate models of complete BL programs.
- The math model for Program is tricky, because it includes the entirety of the STATEMENT_LABEL math model as an attribute. The program's NAME is an attribute, the CONTEXT (a set of instructions with names and bodies made up of BLOCKS), and the BODY, which is just a STATEMENT_MODEL of kind BLOCK.
- A no-argument constructor's model is *this = ("Unnamed", { }, **compose**((BLOCK, ?, ?), <>)).* The first string literal is the title of the program. The empty set is the empty context with no user instructions. The compose is the tree of STATEMENT_MODEL. To create a new Program, use Program p = new Program1(); Up next are methods.
- String replaceName(String n)- Replaces the name of this with n, returns old name.
- Map<String,Statement> newContext() - Creates and returns an empty Map of Strings and

Statements of the dynamic type needed in replaceContext. The type of context is a map because the context is a set of STATEMENT_MODELS with individual and distinct names, which are the keys of the map. This makes sense.

- Map<String,Statement> replaceContext(Map<String,Statement> c) - Replaces the context of this with c, returns the old context. Clears c.
- Statement newBody()- Creates, returns a Statement with default initial value, with dynamic type of this.
- Statement replaceBody(Statement b) - Replaces body of this with b and returns old body. Clears b.

Context Free Grammars:

- A grammar is a set of formation rules for strings in a language.
- A language is a set of strings resulting from combinations of individual characters from alphabet X. If L is a language, then it is a set of string of X.
- An alphabet is a set of characters, alphanumeric or otherwise (thanks Brainfuck).
- In BL, our alphabet is one of tokens of strings, collected from tokenizer (why the fuck did we do tokenizer so late)
- Honestly, just read the slides on this one, I give up. It's late, and the slides are going to teach you way better for this topic, as it doesn't translate well to bullet points and sass delivered via PDF.

Tokenizer (FUCKIN' FINALLY):

- The Tokenizer takes a string of fuckin' characters (the fuckin' source code) and puts them into strings of fuckin' tokens (or fuckin' words).
- CFG's deal with fuckin' languages over the fuckin' alphabet of individual fuckin' characters. We want our CFG to deal with fuckin' strings or fuckin' tokens.
- I am tired.
- The job of the fuckin' tokenizer is to transform a fuckin' string of fuckin' characters into a string of fuckin' tokens.
- EX: I fuckin' Add fuckin' my fuckin' ass to fuckin' bed becomes <"I", "fuckin' ", "Add" "this string is too long you fuckin' get it">
- THE BL TOKENIZER (FUCKING STONER PIECE OF SHIT) USES WHITESPACE AS SEPARATORS AND TOKENIZES EVERYTHING ELSE AND RETURNS IT AS A STRING (BUT NOT A STRING LITERAL YOU KNOW WHAT I MEAN)

DO HASHING AND BINARY TREE SHIT TOMORROW

Hashing:

- Instead of searching through all items linearly, we store objects in many smaller buckets and search through the one that contains the object we are looking for, and this bucket can be quickly identified.
- The bucket your object will be contained in (value of hash function) mod (number of buckets). The collection of buckets is called a hashtable. The hash function is different for different objects.
- The hashCode should run in constant time, and give different output values for different input values. Worst case, everything falls into one bucket. Best case everything falls into its own bucket.

Binary Search Tree:

- The size of the tree is the total nodes in the tree, the height is the number of nodes in the longest path, and the labels of the tree is the set of the labels of each node on the tree.
- The binary tree can be the empty tree, where it has no nodes at all, or it has a root node with two subtrees under the root, on the right and left.
- The compose functions adds two trees under a root so it makes one binary tree.
- Traversal orders:
  - Pre-order means the root is visited before left and right. Root, left, right.
  - In-order means the left is visited first, then root, then right.

    - Post-order means left, right, root.
- Binary search trees can be used to search for items of any type T for that has defined a total preorder of those items. This means subtrees on the left have T's less than the root, and the right have T's greater than the root.
- Using this idea, searching can be done by comparing the root with the value you are searching for. If its greater, recurse into the right subtree and keep checking the root until the tree is empty, left otherwise doing the same thing.
- Insertion is the same, except you would only insert the value when you have an empty subtree, which will be where you make the value the root, and reassemble the previous tree.
- Removal is easy for the left and right trees, but difficult for the root. When you need to remove the root and have no right tree, make the left subtree the new tree. If you have the right tree, use the smallest label from the right subtree.