Computer Programming I

Avery Karlin

Contents

Primary Textbook:
Teacher:

1 Introduction to OCaml

1.1 Program Design

1. Programming can be viewed as the division into four steps, understanding the program, formalizing the interface, writing test cases, and implementing behavior
   (a) The first involves finding any relevent concepts and how they relate to each other within the program, while the second determines input and output, and their respective formats
   (b) The third determines the correct answer to standard cases, edge cases, and erroneous cases (in which the user is at fault) for how the program should behave, writing test cases in the program
   (c) The fourth step is the programming, often taking recursive decompositions of the problem, dividing it by concept, the relationships between, and the interface to allow easier debugging

1.2 Basic OCaml Programming

1. OCaml supports integer (int), boolean (bool), and string (string) primitive/atomic types, able to be defined into expressions combining the types and operations on them, working by order of operations
   (a) Basic operations on integers include +, -, *, /, mod, string_of_int (converting integer to string), on strings (surrounded by double-quotes) include $\hat{}$(concatenation), on booleans include not, &&, ——, and general comparisons include =, ¡¿),(inequality ¡, ¡=, ¿, ¿=
2. Models of computation are ways of thinking about how a program executes, allowing prediction of its behavior, such as object or value oriented programming
   (a) Value-oriented is the idea that running expressions reduces it to a value, rather than an action, such as input/output, as the basis of OCaml
   (b) This is designated by the notation $< exp >=>< val >$, called the evalutation of the expression by calculation as an abstract model of the computer calculation
      i. The internal steps are written out by $\|- >< intermediaryexp >$ by order of operations
3. If-then-else constructs, written "if ¡exp¿ then ¡exp¿ else ¡exp¿", evaluating one of the last two based on the initial expression
   (a) Since the constructs are expressions by themselves, the else is required since the construct expression needs a value if the expression is false
4. OCaml programmings are made of declarations and commands, the former defining constants, functions, and expressions purely value oriented, the latter testing and generating output, non-value oriented
   (a) The let declaration is written as "let ¡id¿ = ¡exp¿", binding the identifier to the value, unable to be modified otherwise in the program
      i. It is often annotated by "let ¡id¿ : ¡type¿ = ¡exp¿" for ease of reading, though this is not required
      ii. Let declarations then automatically substitute each subsequent ¡id¿ with the value of the expression
      iii. Redeclaring a variable is called shadowing, such that the each identifier is substituted by the nearest enclosing declaration, though this is not recommended, producing

ambiguity
        A. Shadowing rather than redefining is apparent within the limited scope of some shadows, such that it reverts back to the original
    (b) Let-in constructs are local declarations, written "let ¡id¿ : ¡type¿ = ¡exp¿ in ¡exp¿" only using the decaration within the in expression
        i. This is able to be nested, such that each applies in the next which eventually all apply within a single expression
    (c) Parameterized expressions/functions, declared by "let ¡id¿ (¡id1¿:¡type1¿) (¡id2¿:¡type2¿) : ¡type¿ = ¡exp¿", called by "¡id¿ ¡val1¿ ¡val2¿"
        i. This runs by substituting the function expression for the function call and the variable id into the function
5. OCaml is strongly typed, such that each type is different with strict rules about combination or conversion, called well-typed if it has at least one type, able to be determined by the operations acting on it or the data
    (a) The compiler automatically typechecks for ill-typed expressions which cannot be evaluated due to type contradictions
        i. Comparison operators return an error with two values of different types
        ii. The compiler will also alert a missing argument or an extra argument to some function
    (b) Functions use explicit type definition to improve clarity, abbreviating in the description the input types and the output type by "¡id¿ : ¡in1Type¿ -¿ ¡in2Type¿ -¿ ... -¿ ¡outType¿"
6. OCaml has the "failwith ¡errorString¿", ending the program and printing the error message if the expression is evaluated
7. Commands in OCaml are used for I/O, not calculating a value, but rather having some external effect, only able to be used in the top-level of a program, precluded by ";;" to distinguish commands
    (a) "print_string ¡String¿" is used to print a string in output, ending the string by '
    n' for a newline, able to use an expression to evaluate to a string
        i. Similarly, "print_endline" prints the string with an automatic newline, "print_int" for an integer
    (b) The "run_test" command is stored in the Assert library, called by the "open Assert" command at the start, used by "run_test ¡descriptionString¿ ¡boolVal¿"

2 Lists and Recursion

1. Lists are sequences of at least zero values, either [], the empty list/nil, or "item1::item2::[]" as the list of [item1, item2], able to compound to longer lists, where :: is the constructor operator for the list
    (a) Lists of lists are also able to be created, and expressions can be used as items, evaluating to the value added to the list
    (b) Lists are also able to be made by shorthand, "[item1;item2]"
    (c) List types are denoted by "¡primitiveType¿ list", required to be a homogeneous/single-type list, such that double lists are denoted "¡primitiveType¿ list list" and so forth
    (d) OCaml is able to deconstruct expressions by pattern matching, written "begin match ¡exp¿ with — ¡case¿ -¿ ¡return¿ ... end"

        i. Thus, for a list, the two cases to test for are "[]" and "<var1>::<list1>", checking for an empty list, such that if the latter case is found, the ¡var1¿ and ¡list1¿ identifiers are bound to the cooresponding list parts
    (e) List concatnation is performed by the "@" operator
2. Lists in OCaml are defined self-referentially, as an inductive data type, stating how to build an atomic instance, and construct a larger value out of it
    (a) As a result, both calculations and creation of lists are done by structural recursion, using an empty list base case, calculating for the head assuming the tail has been calculated already
        i. This can be determined by writing out each new recursion to analyze how each level of recursion can be used to get the desired result
    (b) Recursive functions in OCaml must be defined by "let rec" at the start instead of just "let"
    (c) OCaml does not ensure that all recursion is structural (taking a smaller portion of the structure towards the base case), providing the possibility for loops or an infinite loop

3 Tuples and Nested Patterns

1. Tuples store a set number of values (at least 2) of different types, able to use expressions or complex types, denoted generally as "(¡val1¿, ¡val¿2, ...¿) : ¡type1¿ * ¡type2¿ ..."
    (a) Pattern matching can be used similarly, though generally only needing a single case, due to only one form of construction
        i. Let constructions can also be used, written "let $(< var1 >, < var2 >) =< var >$ $in < var1/var2 >$" to set $< var >$ to either $< var1 >$ or $< var2 >$ for a two-item tuple
        ii. Nested patterns of lists and tuples can also be combined for more complex pattern matching, testing the patterns in the order listed in the match statement
        iii. If there is no match, it will return a Match_Failure when run, such that it will give a non-exaustive pattern-matching warning during compilation
        iv. In a pattern match, if a variable is not used, it can be replaced by _ to act as a placeholder/wildcard for an unused value
    (b) Empty tuples are represented by () with type of unit, used as the function paramter for functions without proper inputs, though able to be shorthanded as () in that case instead of (¡var¿: unit)
        i. It is also considered the return type for a command function, such that no output is produced

4 User-defined Datatypes

1. User-defined datatypes are needed to represent application specific types, used to avoid issues with primitives such as invalid values, confusion of types, and invalid default operations
    (a) They are defined by "type $< id >= | < val1 > | < val2 >$ ...", where $< id >$ must be all lowercase, while each $< val >$ must start with an uppercase letter, each representing an atomic, discrete value
        i. These are constructed purely by the ¡val¿

(b) Types are also able to be represented as tuples of multiple datatypes, defined by "type $< id >= | < val1 > of < type1 > | < val2 > of < type2 > ...$"
   i. These are constructed by "$< val > (< type\_values >)$", with more complicated match expressions
   ii. Similarly, type abbreviations can rename existing types, such that "type $< id >=< type >$", acting as a substitute for a general constructor name
(c) Types can be defined recursively as well, with some base case, with the other discrete values using the type itself

5 Binary Trees

1. Trees are used in CS either naturally or due to allowing ordering large amounts of data for efficient searching, defined recursively as either empty or a node
   (a) Root nodes are those at the top of the tree, leaf nodes at the bottom, internal nodes as all others, with each node having 0-2 child notes in a binary tree
   (b) Each node stores some value, and a link to each of its child nodes, but not its parent node, generally "$< leftChild >, < val >, < rightChild >$"
   (c) Trees can be traversed by pre-order (left, right, node), in-order (left, node, right), post-order (left, right, node), able to be used to list items or search
2. Trees can allow for easier searching if there are linearly ordered labels, such that binary search tree invarient are such that the right node being greater than, the left less than, Empty as the default, such that search is logarithmic
   (a) Binary search trees are constructed such that the nodes are inserted or deleted in accordance with the ordering
   (b) Inserting is similarly done to searching, while deleting requires a simple case for 0 children and 1 child, but for 2 children, the left maximum leaf replaces the deleted node and is removed from the bottom

6 Generic Functions and Datatypes

1. Generic functions are those where the type doesn't change the function, given type '¡var¿, where '¡var¿ is the type variable, instantiated to any type by the function, but unable to be reassigned to a different type
   (a) This allows the reusing of functions and algorithms, saving time in both coding and debugging, rather than having large numbers of close to identical functions
   (b) Types can also be made generically, written "type $('< var1 >,'< var2 >) < id >$", such that they can be generalized to being made up of any type components, able to be computed by generic pattern matching

7 First-class Functions

1. Since functions are values in value-oriented programming such as OCaml, functions are able to used as parameters for other functions and returned by functions, called the first-order function characteristic

(a) As a result, functions can also be partially applied, giving only a portion of the inputs, such that it returns a function with that variable filled

(b) Anonymous functions are those without a name, written by the keyword fun, with syntax "fun $(< var1 >:< type1 >)(< var2 >:< type2 >)- >< exp >$", acting as values

i. These are able to be bound by a let expression, "let $< funName >:< type1 > - >< type2 > - >< retType >=< anonFunction >$", equivelent to a standard function form

2. As a result, first order functions are able to be combined with generics to produce higher order functions, which are more generalized functions, taking a function and applying it to some generic object

(a) Common examples of this are used to transform a list by some function or to fold a list by some combine function and a base case into a single object

3. Partial functions with option types are those which do not always find a value, used for those that require specific inputs to return a proper output, using an option type to return a null option if no value is possible, and a value storage option otherwise

8 Modularity and Abstraction

1. Abstract types are those which bundle the operations and type name with an interface for the functions by which it can be used, but do not allow direct access to the implementation/proper function codes

(a) Modules are independently compilable collections of code, containing abstract data types and their functions, which when providing common data structures or algorithms, are called libraries

i. These are accessed either by ";; open $< Module >$" or by using dot notation for any identifiers given by the module $(< Module > . < Id >)$

(b) Interfaces are created either in an mli file type or a module type signature, the former written by the type name, followed by each function written by "val $< funcName >:< input1 > - >< input2 > - >< output >$", where val is used to denote an expected value

i. Module type signatures are written inside an ml file by "module type $< typeName >= sig < typeInterface >$ end", with the interface written as in an mli file

ii. The implementation can be written as a module of the interface as a nonabstract type, by "module $< typeName >:< AbstractTypeName >= struct < typeImplementation >$ end", called an explicitly named module

A. In this case, the ml file in which they are defined must be opened, after which dot notation must be used for each type, rather than either

(c) For each implementation, OCaml checks if it fully complies with the interface, implementing all expected functions, though it is able to have additional ones

2. Abstract types are useful for not showing the specific usage of implementation, allowing any particular means of implementing, using some implementation type combined with an invarient/rule to preserve the type properties

(a) Each choice of representation and invariant has respective advantages and disadvantages in terms of time optimization

9 Unit and Sequencing Commands

1. Functions are able to take or return unit variables in cases in which the parameter/return type are irrelevent, used by commands to represent the external effect produced
   (a) Thus, "print_string" and other similar commands return a unit type
2. Commands can be embeded in an expression using a single operator, ";", between the command and the next portion of the expression, returning a unit value which is ignored, but which may have side effects within the program
   (a) As a result, the semicolon is always used as a separator, rather than as a terminator like in many other languages, seperating commands, fields, elements, or running top-level commands

10 Mutability and Abstract Stack Machine

1. Records assign names to specific values, such that they is defined by "type $< name >= \{< name1 >:< type1 >; ...$¡nameN¿ = ¡valN¿;$\}$", with records written by "let $< var >:< name >= \{< name1 >=< val1 >; ... < nameN >=< valN >\}$"
   (a) Values are accessed by "$< var > . < nameNumber >$", able to be changed slightly in a new variable by "let $< newVar >= \{< var > with < nameNumber1 >=< valNumber1 >; ...\}$"
   (b) Existing records can be copied and modified by "$\{< oldRecord > with < nameNumberN >=< valNumberN >\}$"
2. Mutable data are those which are able to be properly changed, rather than shadowed, destroying the original data, called imperative programming rather than pure programming, allowing aspects of the program to share data elements
   (a) Mutable records are defined by "type $< name >= \{mutable < name1 >:< type1 >\}$", able to be defined outside of a function as a global variable
   (b) These are changed by "$< var >< - < newVal >$;", acting as a command returning a unit
   (c) Due to mutable values being shared by functions, multiple variables are able to point to the same data structures, called aliasing, destroying the substitution programming simplification method
   (d) Programs using mutable data are refeerred to as imperative programming, while those that don't are pure programming
3. The Abstract Stack Machine is an alternative to the substitution model, accounting for the location of data in the computer memory, assuming the language is type-safe and garbage-collected, such as OCaml or Java
   (a) It is made up of the workspace to keep track of the expression currently being evalutated, the stack for bindings, primitive values, and partially simplified expressions, and the heap is for data structures
      i. Bindings are made up of an identifier and a reference to an address/location of a piece of data, denoted by an arrow
         A. References are abstract locations, not having a specific value within the computer, while pointers are the specific reference location value
      ii. The heap contains cells (labelled data types with possible parameters/references), records (containing a table with the variables on the left, the values double boxed

on the right), and anonymous functions

(b) The ASM begins with an empty stack and heap, finding the left-most ready expression within the workspace to simplify, adding to the stack and heap as it moves along, removing those expressions from the workspace

i. Each value is added below previous values in the stack, using the most recent value within the stack at a given time to display shadowing

ii. The stack functions by popping out all elements from the bottom as it leaves the scope, such that it follows the LIFO procedure

iii. The evaluation of the expression is complete when the result is the remaining item in the workspace

iv. Function definitions are converted to an anonymous function with a binding, the binding put in the stack, and the anonymous function placed in the heap

(c) Functions are simplified by adding the outside layer of the function into the stack with a space inside for the result of the inner layers, changing the inner layer into the new workspace, evaluating, then bringing it back to the workspace and popping off the stack for the inner layer

i. The space inside the outer function layer is denoted by "(___)"

(d) Aliases are denoted by references in the stack to the same object in the heap, tested by the reference equality operator "==", rather than the structural equality "="

i. For primitive values, since they are only in the stack, the reference equality operator returns true if the structural equality operator does

4. Options are denoted about the value with an arrow at the top left corner, while a slash through the reference box implies it links to a None option

11 Queues/Linked Structures

1. Linked structures are made out of a data block attached to other data blocks in some sequence with some set of operations able to act on it

(a) Queues are structures with the function enqueue to add to the tail, and dequeue to remove from the head, with last in, last out (LILO) usage, with each node made up of a value and a mutable option link to the next value

(b) The queue also contains a link to the first and last non-zero elements within the queue

2. The Queue Invariant states that either the head and tail must both be None, or for the head as Some n1 and the tail as Some n2, n2 is reachable by following next pointers from n1 and n2.next is None

(a) As a result, the same node is not able to be in the queue more than once as an alias, due to violating the invariant by creating a cycle

3. Mutable structures can be evaluated by a loop helper function defined within the overall function, either by recursion or by tail call optimization

(a) Tail call optimization saves space by modifying recursion into iteration, emptying the workspace of the previous function call fully before calling it again, such that the bindings can be removed from the stack

i. As a result, the final call within the recursive function is the calling of it again, having fully evaluated the previous call, using an accumulation argument to preserve values from previous calls

(b) This provides more risk of undetected infinite loops, due to not causing a stack overflow, either by not iterating through the structure or iterating through a cycle

12 Local State/Objects

1. Local states are states packaged with several functions that operate on it, shared mutably between the functions, unable to be modified outside of those functions, used to avoid the need for many global states
   (a) Rather than a global record to keep track of a state, there is a function producing a new record each time it is run, returning a function to increment that counter
   (b) This is accomplished by the ASM storing the function, along with any local stack bindings needed to evaluate it, to the heap, such that the counter isn't on the stack, but is readded by the function immediately when called, even before arguments are added to the stack
   (c) The combination of a function and locally stored bindings is called a closure, with the localization of the state being called encapsulation, restricting access to the data to the specifically designated functions
      i. Multiple objects are also able to be tied to the same data, to allow separation between the modification functions and the usage functions
2. Objects are created by making a record of several functions, and a creation function that produces a state record modified by the function records
   (a) This is made easier by the 'a ref (reference) type, used as shorthand for a record with a single mutable field called contents, with the ! operator to provide the contents, and the := operator to redefine the contents

13 OCaml GUI Library

1. GUIs use an event-driven model of reactive programming, reacting to user-caused events, such as mouse or keyboard input
   (a) OCaml uses the event type, as a record to indicate the status of keys, the mouse button, and the mouse location
2. The OCaml graphic library is used by adding the "graphics.cma" compiler flag, providing functions such as open_graph (to open a new window), clear_graph (to erase the window), resize_window (to set the window size), size_x (to return the width), and size_y (to return the height)
   (a) It also provides the type color, as a record with items r, g, and b, with various predefined colors
   (b) It also contains a pen with functions set_color, move_to, and (plot x y) to color the pixel with the current pen color, as well as (line_to x y) (from the current location) and functions for other shapes drawn
   (c) The graphics library uses double buffering, modifying a hidden second copy of the window, pushing that into the main window and erasing, instead of modifying the main window directly to prevent flickering
3. GUI libraries are generally made with built-in buttons and textboxes, as well as systems for positioning relative components of the GUI, to avoid precarious global window coordinate

systems
   (a) Widgets are made such that they generate themselves using a widget-local coordinate system, with the left upper corner as the origin, offset by the main window
      i. As a result, there is the Gctx module used to draw the widget, with the gctx type including the color and location, used to get the offset, all from the upper left hand corner, as well as including methods to convert from coordinate systems
         A. Gctx also has functions used to draw different objects
      ii. The OCaml graphics library has the origin at the lower left corner, rather than the upper, which is non-ideal for a GUI, due to adding space to the bottom, rather than the top
   (b) The widget module is a closure containing a repaint function (using a gctx coordinate) and a size function, including a space widget, a label widget, a canvas (parameterized drawing) widget, and a border widget (taking another as a parameter)
      i. The hpair function is used to pair two widgets into a single widget, aligned at the top, equally next to each other, and a border widget, padding the outside of a widget into a larger widget, connected to the original widgets
      ii. This allows forming a widget tree to create a complete widget appearance, generating a complete widget, with the base as the highest level/toplevel
4. GUIs are generated by creating an infinite loop run function, repainting the entire GUI with any changes constantly, waiting for an event, then called the handle function of the root widget
   (a) Event types are mouse drags, moves, up, and down (returning the type and position), as well as key presses (returning the character pressed)
   (b) Event handlers of a widget take in the graphics state (gctx) and the event, routing through the widget tree to the final widget
      i. Notifier widgets are used to check for events, keeping a list of event listeners, sending the event to each event listener, which in turn performs an action function if the event is detected
   (c) Widgets which have states also have controller objects to control the state, routing to the same local state
      i. The controller object might also have a list of change listeners, modifying those values based on the new object value

14 Introduction to Java

1. Programming is divided into functional (with immutable data structures and recursion), imperative (with mutable and iterative), and object-oriented/reactive (with abstraction and encapsulation)
   (a) Java is superior at the last, equal with OCaml at the second, and the worst at the first, while OCaml is the reverse
   (b) Java objects combine the local state, method definitions, and instantiation into a single construct, called a class, made up of fields (instance variables), constructors, and methods (member functions)
      i. The keyword public makes the class globally available, while the keyword private means it can only be accessed by code within the class itself, such that fields are generally private, while classes are public, methods either

   ii. Methods within Java are declared by "$< permission >< returnType >< name > (< type1 >< param1 >, ...)< Code >$"
   iii. New variables of a class are created by "$< className >< var >= new < initializer >$;"
(c) The main function in Java is stored within a class, with the definition "public static void main(String[] args)"
   i. Void is used similarly to unit, to denote the lack of a return value

2. Java is a statement language, made up of a series of commands executing, each ended by ;, with conditional statements reducing to a statement
(a) Java contains the null value to imply the lack of a reference, automatically assigned to a variable, such that attempts to evaluate return a NullPointerException, instead of the option type
(b) Unlike OCaml, all variables are mutable by default, able to be redefined with equals
(c) Interfaces in Java, similar to OCaml, are defined by the header "public interface $< intName >$ {}", containing the headers for each function, without any constructors or fields
   i. It is used by adding "implements $< intName >$" after the name of the class
   ii. Since interfaces can render the specified implementation unknown until the program is running, Java uses dydnamic dispatch to provide the function mode based on the dynamic class stored in the interface object
      A. On the other hand, static methods can be used for those that are associated with the class, rather than the object, able to be evaluated at compilation, thus only able to reference other static methods and fields of the class
      B. Static methods are called by "$< className > . < methodName > ()$", not requiring producing an object instance to use
      C. Static is also able to be used to provide fixed values to a field within a class, given a value outside of the constructor automatically

3. Java contains primitive types of int, byte, short, long (integer types), char, float, double (floating point types), and boolean, using overloaded operators such that the same operator can act on multiple types
(a) Referential equality is denoted by ==, while structural equality is determined by the .equals() method
(b) Logical not in Java is denoted by "!", logical and as "&&", logical or as "——", modulus as "%", and inequality as "!="

4. Java identifiers are stored in different namespaces for classes, fields, methods, and local variables, such that the same identifier can be used for different types of items, though this can produce confusion and should be avoided

5. Java has arrays, or sequentially ordered collections of elements indexed with integer positions, accessed in constant time, denoted by "$< type >[]$", with each element found by "$< name > [i]$", with the field name.length to provide the size of the array
(a) Attempting to access a value out of bounds gives the ArrayIndexOutOfBoundsException, starting from 0 going to name.length - 1
(b) Array elements are automatically mutable in Java, such that they can be redefined, initialized for any object or type by "$< type >[] < name > =$ new $< type >[< length >];$"
   i. They are also able to be declared statically instead of initializing by a list of comma-seperated objects within curly brackets, done at the time of declration

ii. Arrays require predetermination of the size, such that the name is a reference to the location on the heap

(c) Multidimensional arrays are noted to possibly not be in order, with only the references to each inner array in an order in the outer array, and the inner arrays in order by themselves, denoted as the inner having the row number of items, the other of the column number of arrays

i. It is noted that the inner arrays can be declared independently (or statically), such that they may not be the same length

6. The Java Abstract Stack Machine is different from OCaml in that almost all variables are mutable, the heap only contains arrays and objects, rather than other data types, it has the null reference, and methods and constructors are stored outside the three components, called a class table

(a) The length of an array is never mutable, while other fields may or may not be (marked with a thick black box), with the array elements just stored in order

15 Subtyping, Extension, and Inheritance

1. Types are used to restrain interaction between parts of code, such that Java is strongly typed, each expression given a type, using objects of a type, built from a class

(a) Interfaces are considered to allow subtyping, with each class as a subtype of the interfacts it implements, able to be declared for the supertype, only using functions from the supertype

(b) Classes are able to implement multiple interfaces, seperated by a comma, satisfying the method requirements of both

i. In addition, interfaces themselves are allowed to extend or inherit from another interface, and a class can extend or inherit from another, gaining the methods of its supertype

A. Interfaces are able to inherit from another interface by the "extends" keyword, gaining any methods from the higher interface, allowing it to be fit in a variable of either the interface, or the extension

B. Inheritance allows classes to gain the fields and methods of its superclass by the "extends" keyword, though each may only have a single superclass, though able to have many subclasses, able to be placed in a variable of itself or any level of superclass

C. Private fields in a superclass are unable to be accessed within methods of the subclass, except if the "protected" keyword is used instead

ii. Since constructors require the class name, they are unable to be inherited, but is allowed to access the superclass's private fields by the function, super($< params >$), acting to call the superclass constructor within the subclass constructor

A. This is noted to be required to be the first line within the constructor

(c) The compiler is only able to see the type declared for a variable, such that it isn't allowed to be used as the actual object type, but only as the variable type

i. As a result, the variable type is static, while the class itself is dynamic, such that the type restricts how the class is used and provides the expression with a type

ii. Similarly, the dynamic class determines which particular methods are used during runtime, with a single type for each memory block, unlike the static type which may

have multiple possibilities for each object

2. The highest Java class is called the Object call, automatically extended by every class without a superclass, acting as the root of the class tree
   (a) The Object class provides the toString method and the equals method automatically to every class

16 Dynamic Methods in the ASM

1. Dynamic dispatch is used to show the normal method calls and constructors, controlled by the dynamic class, rather than the static type
   (a) The class table is the fourth component, acting as a tree, with extends fields linking to the superclass, and containing the methods and constructors of the class
      i. Methods are able to reference the class that called them by the "this" keyword, implied in every field access, such that the workplace modifies the code to make it explicit
      ii. In addition, "super()" is made explicit at the start of the constructor by the workspace as well, such that there must be a default constructor unless a parameterized super constructor is used
      iii. Static values are stored within the class table as well, called purely by the class name rather than a specific object
   (b) When the constructor is called it allocates the object onto the heap, creating space for all fields and setting the default values (0 for numbers, null for references), having saved the remainder of the workspace on the stack, taking the function from the class table
      i. A pointer is then created for the class, called the dynamic type of the object, creating a "this" field on the stack, attached to it
         A. As a result, the main difference between a static and dynamic method is the lack of a "this" pointer to allow access to nonstatic methods and fields
      ii. It is noted that on the heap, the object is given the dynamic class as a point of information
   (c) Similarly, dynamic method calls create a "this" reference, while using dynamic dispatch to find the correct method, searching the class table upward with the class table pointers
2. It is noted that while methods of a subclass cannot modify the private fields of a superclass, the inherited methods from the superclass are able to

17 Generics, Collections, and Iteration

1. Libraries are useful due to the premade abstraction, such that it is a skill to be able to read the documentation and use the abstraction efficiently, as well as to be able to design functional libraries
2. Polymorphism allows functions to take different argument types, contained in Java by subtype polymorphism (using a supertype variable) and generics/parametric polymorphism, the latter ideal for container data types
   (a) Generics are used adding "$ObjectName< E >$", after which E is used as the type, declaring the object variable itself by specifying the E type by similar notation

        i. This allows specific objects to be returned, rather than generalized static types, allowing easier usage, though it limits the type allowed for each object instance (able to be regeneralized by setting E as Object)
    (b) Generic types must have the type parameter as invarient, such that if type A is a subtype of type B, if A is the parameter for one object and B for another, the former is not a subtype of the latter
3. The Java Collections Framework contains List, Deques, and Sets interfaces, each with a generic, under the Collection$< E >$ interface, as well a finite map interface with $< K, V >$ (key, value) generics
    (a) It is noted that the contains and remove functions both use Object, rather than E, due to the equals function of a specific class not necessarily returning false for different classes of objects
    (b) ArrayList¡E¿ and ArrayDeque¡E¿ stem from List¡E¿ and Deque¡E¿ respectively, while LinkedList¡E¿ stems from both, as the three main sequence implementations
        i. ArrayList and LinkedList allow accessing by position, the former in an array, such that it is equally fast to access any item, while in LinkedList, it is faster to add items into the middle
    (c) Set has HashSet stemming from it, as well as the SortedSet interface, from which TreeSet stems, while Map has HashMap and SortedMap, based off of hashing and binary search trees respectively
    (d) $Collection < E >$ extends $Iterable < E >$, requiring an iterator method giving an $Iterator < E >$ object, the object checking if there is another item in the list, providing the next item, and eleting an item
        i. The iterator allows abstraction of traversing a collection, regardless of the collection implementation
        ii. This can be made easier for any Iterable by the for-each loop, written "for(Type TypeVariable : IterableVariable) ...", automatically iterating for each item in the iterable
            A. It is noted as well that arrays are iterable in Java, such that the for-each and iterator methods are usable

18 Overriding and Equality

1. Method overriding is the idea of a subclass redefining a superclass method, utilized by dynamic dispatch methods
    (a) It is noted that dynamic dispatch controls every single method invocation, such that even if it is called within a superclass method, dynamic dispatch still begins again from the dynamic class
    (b) As a result, each superclass must account for the implementations of the subclass, such that the behavior of any overriden methods still aligns with the superclass functions
        i. Thus, it is generally used only if both classes are from the same author, or the parent class is made for overriding
        ii. Composition and delegation can often be used as a substitute for method overriding
    (c) The "final" keyword can prevent overriding of a method by a subclass
2. The equals method generally has to be overriden by subclasses, or it can return that objects are structurally equal at incorrect times

(a) It has to be overriden when the class represents immutable values, such that there is a distinct set of states, such as coordinates or strings

(b) The method creates an equivalence relation on non-null objects, such that it is reflexive (returns true when on itself), symmetric, transitive, consistant, and "x.equals(null)" must return false

i. The symmetric property can cause issues when an object of the superclass and subclass are compared, fixed by ensuring they are of the same dynamic class

(c) It is noted that by default, it is overloaded, rather than overridden, such that the superclass method is still present

i. It can be overrided by the "@Override" tag before the method

ii. The "$< var >$ instanceof $< Class >$" command returns a boolean determining if dynamic typecasting to a more specific variable is allowed

iii. The dynamic class of an object can also be accessed by the getClass() function, which returns an Class$< dynamicClass >$, such that it is referentially equal for objects of the same classs

(d) Equals methods also often check for referential equality to provide a shortcut on the method if they are referencial

(e) The "super.$< fun >$ ()" keyword can also be used to call the superclass function, even if it has been overridden

19 Exceptions

1. Exceptions are necessary for incorrect inputs, optional interface methods when they are not found in a particular implementation, incomplete softwarre, or overflows

(a) Failures can be dealt with by returning an error value, an option call, or an exception (ending the program and decribing the error)

(b) Exceptions in Java are objects in a subclass of the Exception class, thrown by the "throw" keyword when there is an error, ending the program

i. Exceptions can also be caught by a "try ... catch($< ExceptionType >$ e) ...", such that instead of ending the program, the program can respond to the exception

A. Multiple catch blocks are allowed to check for each type of exception, skipping the remainder

B. It is generally bad to have too generalized catches, because that can hide unexpected exceptions

C. Try-catch blocks can also have a "finally ..." block at the end, always running after the try and catch blocks, regardless of if anything was caught or not

ii. Constructors are unable to return null, such that the only way to show an error is to produce an exception

iii. Exceptions are dealt with in the stack machine by putting the catch handler on the stack when the try portion is called, such that when the exception is found, the stack is popped until catch is found, ending the program if emptied

2. Exceptions are a subclass of the Throwable object, while Error objects are subclasses of the Throwable as well, the latter unable to be caught, purely for fatal errors

(a) Exception objects are divided into subclasses of IOException and RuntimeException

i. IOExceptions must be declared in the header of both the method itself and the

interface method by "throws < *ExceptionType* >", while RuntimeExceptions do not need to be
   ii. Methods that call methods that may throw an IOException also must declare the exception throwing in the header
   iii. If the method catches the exception, it does not need to declare the exception
   (b) RuntimeException classes include NullPointerException, IndexOutOfBoundsException, and IllegalArgumentException, not necessary to declare due to the commonality
   (c) User-created exception classes are necessary to declare if they are not a subset of the RuntimeException class
3. Declared exceptions are more difficult to adapt to new code, but are more noticable to the coder due to being explicit, generally only used for very general libraries
   (a) Catch blocks are also supposed to be present as close to the throwing of the error as possible
   (b) It is also often good to catch more generalized exceptions and rethrow more specific, user-defined exceptions as a replacement

20 IO in Java

1. Java abstracts IO by steams, acting as a sequence of values in order, possibly without a specific end, either as an input or output stream, feeding values individually
   (a) The InputStream abstract class has a read() method to return the next byte of data, throwing an IOException if it cannot be read, returning a -1 at the end of the stream
   (b) The OutputStream abstract class has a write(int i) method to write a byte of data, throwing an IOException if it cannot be written
   (c) Both abstract methods are limited to values from 0 - 255, or bytes, and are abstract such that subclasses must implement the methods
2. FileInputStream and FileOutputStream extend the abstract classes to allow reading and writing binary data, using the function close() to end the file stream, constructed with the filename as the parameter
   (a) Similarly, BufferedInputStream and BufferedOutputStream is a wrapper, constructed with the file stream as a parameter, allowing input and output of groups of characters at once to save time
3. The PrintStream class extends OutputStream, used to print different functions, such that the static field "System.out" contains an instance of the class, using the overloaded println method to print various values
   (a) It also contains the "print(String)" function to print a string into the stream without flushing or ending the line, and the flush() function to flush the buffer
   (b) The buffer is automatically implemented by the PrintStream for the output, and flushed at newlines or when full
4. The Reader and Writer abstract classes are used for streams of characters (16 bits), similar to the byte Input/OutputStream classes, returning -1 when the stream is finished, constructed similarly
   (a) There are also cooresponding FileReader/FileWriter and BufferedReader/BufferedWriter wrappers for these classes as well
   (b) Since Strings are immutable, each character would have to be appended to the String,

unless the length was previously known, such that it would be slow

i. The StringWriter class produces a resizable array, doubling the array when filled, able to be written quicker, and then converted by toString()

21 Homework Notes

1. No multi command conditionals?
2. In needed for complex functions?
3. Can't create an empty user-defined datatype, accessed by begin match similar to regular types (Node (a, b) -¿, etc)
4. Need either variable or wildcard
5. When can you use multiple lines in a single begin match command?
6. Type is not properly implemented in interface, though a function for an empty can be, since functions are values in OCaml
7. Equals checks if the structure is identical, in addition to the data for any object
8. Dot library notation can be used rather than including the library directly
9. End of a unit function has no ;, and returns nothing
10. Commands can be used to throw away a function result
11. Cannot call other Java constructor for same class within constructor
12. Arrays are stored within the heap in OCaml
    (a) The Some option is denoted with a bubble at the end of the arrow
    (b) Some/None must be typechecked for, and are noted to not be referentially the same if declared in seperate places, even with the same internal