Ryan Brooks
Prof. Flatt
CS 3520-001
11/30/2021

IMPLEMENTATION - README

Completed Sections:

1. cast - One Star
2. if0 - Two Stars
6. arrays - Three Stars
8. null - Two Stars

Total Completed Star Value:

Eight Stars

Code Change Locations:

0. General Changes:

- All code changes are marked in the code such that If I say a change is located in a place such as "the interp test module in class.rkt" the start of my changes will be clearly commented. The comments are written in the following format where name and # refer to the name and number of the section and function optionally refers to the goal or purpose of the code. Written: ;;#: name - function. An example of this format would be: ;;2: If0 - test - interp and should be easy to locate given the general locations listed below.
- **Moved typings** to their own file along with the parse-type function which is visible in the new types.rkt file. This file types.rkt is now required by inherit, inherit-parse, typed-class, and typed-parse.

1. Cast Changes:

1a. Cast - Inherit.rkt

- Added castI variant in ExpI type definition and corresponding case to type-case in the exp-i->c function definition
- Added tests for castI in the exp-i->c test module
- Updated flatten-class, flatten-super, etc. to use new version of classC which includes super-name

1b. Cast - Class.rkt

- Added castE variant in Exp type definition and corresponding case to type-case in the interp function definition. This required the help of a helper function instance-of-sub?
- Added the function instance-of-sub? (essentially a copy of the is-subclass? function) to find whether class A was an instance of another class B or any of B's subclasses. Located below interp.
- Added super-name to classC

- Added tests for interping casts in the interp test module.

1c. Cast - Inherit-Parse.rkt

- Added s-expression matching for the castI form into parse along with a test in the parse test module
- Added a test for a program with castI in interp-prog

1d. Cast - Typed-Class.rkt

- Added typechecking for castI variants via a new case in the typecheck-expr function.
- Added tests for typechecking expressions with castI in the typecheck testing module.

1e. Cast - Typed-Parse.rkt

- Added a test for a program with castI in interp-t-prog

2. If0 Changes:

2a. If0 - Inherit.rkt

- Added if0I variant in ExpI type definition and corresponding case to type-case in the exp-i->c function definition
- Added tests for if0I in the exp-i->c test module

2b. If0 - Class.rkt

- Added if0E variant in the definition of Exp and the if0E case to the interp function.
- Added tests for if0E in the test module for interp.

2c. If0 - Inherit-Parse.rkt

- Added implemented the case for parsing an if0I from an s-expression in inherit-parse.rkt along with a test in the parse test module
- Added a small test for a program with if0 in interp-prog

2d. If0 - Typed-Class.rkt

- Typechecked if0I expressions in typecheck-expr by adding a new case in the typecheck-expr function. This required finding the least upper bound of two types via the new functions find-lub and find-lub-recur.
- Added find-lub and find-lub-recur to find the least upper bound of two types. These are located below typecheck-expr.
- Added tests for typecheck with expressions that use if0I inside the typecheck testing module.

2e. If0 - Typed-Parse.rkt

- Added a test for a program including the if0 expression in interp-t-prog

6. Arrays Changes:

6a. Arrays - Inherit.rkt

- Added newarrayI, arrayrefI, arraysetI, and beginI variants in ExpI type definition and corresponding cases to type-case in the exp-i->c function definition
- Added tests for newarrayI, arrayrefI, arraysetI, and beginI in the exp-i->c test module

6b. Arrays - Class.rkt

- Added newarrayE, arrayrefE, arraysetE, and beginE variants in Exp type definition and corresponding case to type-case in the interp function definition. This required the assistance of two helper functions construct-array and arr-list-set.
- Added the functions construct-array and arr-list-set which are located below interp.
- Defined the Arrays-set class for use with testing the imperative nature of arrayset.
- Added tests for interping newarrayE, arrayrefE, arraysetE, and beginE in the interp test module.

6c. Arrays - Inherit-Parse.rkt

- Added s-expression matching for the newarrayI, arrayrefI, arraysetI, and beginI forms into parse along with tests for these in the parse test module
- Modified interp-prog to include a case for arrays that simply returns `array
- Added a basic test for a program with an array in interp-prog

6d. Arrays - Typed-Class.rkt

- Modified is-subtype? to work with arrays and added corresponding tests to the test module for is-subtype?
- Implemented typechecking for newarrayI, arrayrefI, arraysetI, and beginI variants' by adding several new cases in the typecheck-expr function.
- Added tests to verify the functionality of typechecking for newarrayI, arrayrefI, arraysetI, and beginI. These are located in the typecheck testing module.

6e. Arrays - Typed-Parse.rkt

- Added a simple test for a program with an array in interp-t-prog
- Modified interp-t-prog to include a case for arrays that simply returns `array

6f. Arrays - Types.rkt

- Added an array type variant of Type
- Added s-expression matching for arrays in parse-type using the form: < Type >
- Added a test for type parsing arrays from `{< Type >} s-expressions

8. Null Changes:

8a. Null - Inherit.rkt

- Added nullI variant in ExpI type definition and corresponding case to type-case in the exp-i->c function definition
- Added tests for nullI in the exp-i->c test module

8b. Null - Class.rkt

- Added the nullE variant to Exp and nullV to Value
- Added the nullE case to interp
- Added tests for null as well as a class to help in testing

8c. Null - Inherit-Parse.rkt

- Disallowed null as a class name in parse-class
- Added expression matching for null when parsing
- Added tests for parsing null in the parse tests
- Added a case for null values in interp-prog as well as a test for this in the interp-prog tests

8d. Null - Typed-Class.rkt

- Defined null to be the subtype and supertype of everything in is-subtype?
- Added tests for is-subtype? in the test module
- Added a nullI case in typecheck-expr and corresponding tests in the test module below
- Added a class definition to assist with testing
- Modified the getI case to return a type error if the type is null
- Modified the sendI case to return a type error if the type is null

8d. Null - Typed-Parse.rkt

- Disallowed null as a class name in parse-t-class
- Added a test to check parsing a "null" class below parse-t-class
- Added a nullV case to interp-t-prog
- Added a test for interp-t-prog

Implementation Strategy:

1. Cast:

The implementation of cast began in inherit.rkt where I defined ClassI to have a Symbol referring to the type and ExpI referring to the body expression that was being cast. At this time I defined the conversion to Exp in the exp-i->c function and the parse from s-expression into ExpI in the inherit-parse.rkt file. Both are very straightforward, so after this I moved into typechecking. The goal of the typechecker for the castI case was to reject any expressions where the type of the body expression was not a subtype or supertype of the type described by symbol. To do this, however, I would need to parse the symbol into type. To allow this, I moved the definitions of types for expressions and the parse-type function into types.rkt which allowed them to be seen by typed-class.rkt without a cyclical build reference.

After doing this, I was able to parse the type from the symbol by converting it to a s-exp and running it through parse-type. I did specifically By calling recur on the body expression, I now had both types. Thus, to check if the body expression's type was a supertype or subtype of the type indicated by the symbol. I checked is-subtype? in both directions. If either function call returned true then the typecheck succeeded and I returned the type indicated by symbol as in the description we were able to assume that the body expression had this type. On the other hand, if one or both function calls failed then I returned a type error as this meant that the cast would always fail.

Similarly to inherit.rkt, in classes.rkt I defined a form castE which took a Symbol and an Exp. For interping castEs there were two major outcomes. If the expression doesn't produce an instance of symbol or one of its subclasses this would cause an error as we can not implicitly convert to types without other information. Otherwise, we return the value of the body expression.

To do this, I first needed a way to check whether classes were subclasses of each other from within class.rkt. Consequently, I updated the classC variant of Class to include a super-name field which is denoted with a Symbol as it is in classI. Next I fixed all references to classC to include this super-name field and consequently allowed super class names to be imported into class.rkt. Under the castE case in interp, I started by checking the type of the value found by interpreting the body expression which we will call v for the sake of simplicity.

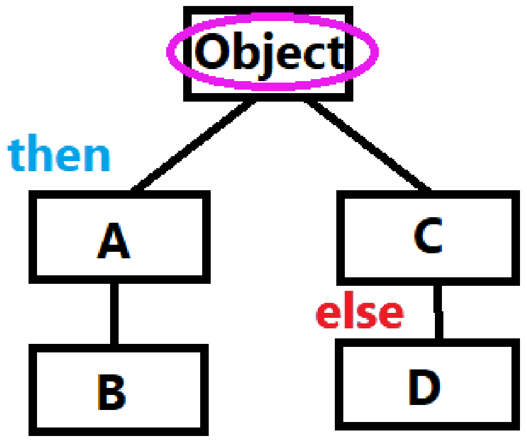
If v was a numV I checked to ensure that the type symbol for the cast was ‘num, returning v if it was true and raising a “cannot cast” error otherwise. If v was nullV if the s was not an Object I raised an error, otherwise I returned nullV. If v was an object, I checked the result of a new function instance-of-sub? which Is a copy of the is-subclass? function that is visible to class.rkt. If the body result was an instance of a class that was a subclass of the class indicated by symbol, I returned the ObjV v. Otherwise I returned a cannot cast error as we can not implicitly cast upward. Finally, if v was an arrayV. I was stuck on what to do for this one. Firstly, Symbols could not have spaces so I was unsure how to define “< Type >” in the input to cast such that it wouldn’t interfere with strange object names and be uniform. As I was unsure if we were supposed to resolve the issues between the different parts of the project that didn’t state otherwise I chose to raise a casting error and just say that you cannot cast to an arrayV (especially as there is no symbol that would directly become “< Type >”).

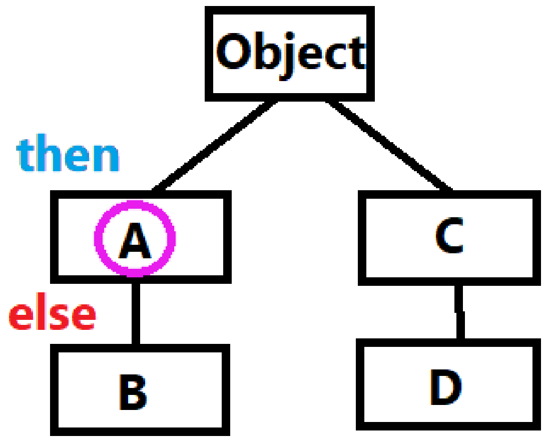
2. If0:

A lot of the challenges that lied in implementing if0 were present not interpreting this expression but rather in typechecking. As usual I began by implementing if0I into inherit.rkt. if0I has three fields: the condition (c), the then (t), and the else (e) each of these an ExpI themselves. Next I began adding the case for exp-I->c, and writing the case for the parser in inherit-parse.rkt. After this was done I actually first moved onto implementing Interp before typechecking.

I first implemented the if0E variant of Exp in class.rkt which was identical to if0I but with Exps in place of ExpIs. Implementing the functionality of this in interp was very simple. I began by checking if the value of the condition was a numV by doing a type-case on the Interp of (c), returning a “not a number” error if it wasn't. Then if the value of the numV was equal to 0 I returned the value of interping (t) otherwise I returned the value from interping (e). This was trivial to implement and easy to see why it works.

Finally I began to consider typechecking. Other than checking if the conditional expression was a numT, which was trivial, If0 has two other types that need to be checked. These are the type of the then expression and the type of the else expression. I knew logically and from the description that the most accurate type to return would be the “lowest” type that encompassed both then and else’s types as it would be the closest.



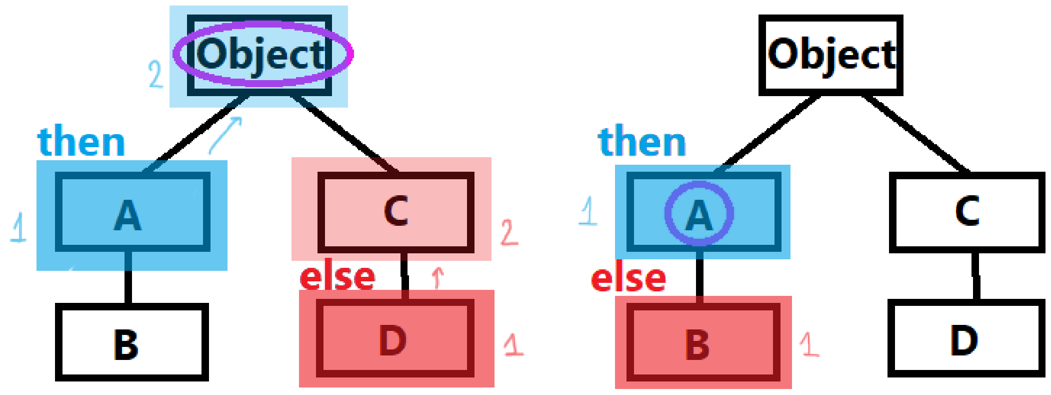


I determined this by considering something similar to the above graph. Firstly, I knew that we could only implicitly convert (or use) some type X in the place of some type Y if X is a subtype of Y. This is because, at the simplest level, as a subtype of Y, X is guaranteed to have the same amount or more information than Y. Thus, we can simply disregard certain information from X to achieve something of the form Y. Thus, the object that was the closest to both the types of then and else while being a supertype of both is the most accurate type. This type shall hence be referred to as the "least upper bound" or LUB.

Before I even began implementing the case for if0I in typecheck-recur I implemented a way to find the LUB of two types as I knew I would need it. The functions find-lub and find-lub-recur are my solution to this problem. The former is a wrapper class for the latter which is recursive. To begin with find-lub, it takes in two types t1 and t2 as well as the second expression e2 (for errors) and a list of typed classes t-classes. Using type cases, if both types are not of objT then we merely see if they are equal. If they are we return t1 (although either works), otherwise we throw a type-error. This works as for any primitive types, the LUB of two is the types themselves and there is no notion of sub or supertypes and they must be equal. When typechecking if t1 or t2 is an objT and the other a primitive type we also return a type error as we will never be able to implicitly convert between these. Finally if t2 and t1 are both objT, we call find-lub-recur.

Before I describe how we call find-lub-recur let me describe what the function parameters are and how it functions. It has four class parameters c1, c2, super-c1, and super-c2, and then t-classes. In each loop, c1 and c2 always represent the classes associated with types t1 and t2. On the other hand super-c1 and super-c2 are the highest classes we have traversed up the tree at each loop. Consequently, as we havent climbed the tree when we call find-lub-recur from find-lub, both c1 and super-c1 are the class associated with ObjT t1 and both c2 and super-c2 are the class associated with ObjT t2.

In find-lub-recur, we first check if c1 is a subclass of super-c2, if it is then we return super-c2 as it is the LUB. Otherwise We check if c2 is a subclass of super-c1, if it is we return super-c1 as it is the LUB. Otherwise, We increment super-c1 and super-c2 by getting the super class of the current super-c1 and super-c2 and calling the function recursively with those as our new super-c1 and super-c2.



This image displays how the function works. First it checks the current types as c1 and super-c1 are the same and c2 and super-c2 are the same. On the right tree, this works and A is found to be the LUB and the recursion is over immediately. On the left tree neither are reachable by going downward from super-c1 (A) or super-c2 (D) so we go into a second pass. This time super-c1 is (Object) and super-c2 is (C). We know c1 (A) is reachable from (Object) by definition, but since c1 (D) is now reachable, we deduce that (Object) is the LUB.

So with this function working, I wrote the type-case for typecheck-expr. First I ensured that the conditional (c) was of type numT, otherwise returning an error. Then I ran find-lub on the types found by recurring on the then and else expressions and that was the result. Thus, this finished my implementation.

6. Arrays:

I started implementing arrays in inherit.rkt by creating the ExpI variants for the 3 listed functions newarrayI, arrayrefI, and arraysetI. As I had already moved type definitions to a new file types.rkt, it was trivial to allow a field of type Type as an input to newarrayI. Converting these to Exp's in exp-I->c was trivial and the only thing of note done in this process was ignoring the type of the array as it was only necessary for typechecking. At this time I also setup the input parsing for these I expressions in inherit-parse.rkt with the only notable thing being that parse-type was used on the second part of the newarray s-expression to convert it into a type to be passed into newarrayI. This was the simplest way of transforming this information as the function for parsing types was already implemented for me.

With that out of the way, I began to fully consider typechecking. First, in types.rkt I added a new type called array and implemented the type parsing for it. In this version of curly, arrays are represented by the following syntax < Type >. It was simple to implement this and all that the arrayT type needed to record was array-type or the type stored between the angle brackets. Back in typed-class.rkt I wanted to define how is-subtype would work for array types now that they were implemented. At first I thought that it would make sense to say that arrays were subtypes if the type of their elements are subtypes. Then I read part 7, realized that it required run-time checking and rewrote my typechecker to merely see if two array types were identical or not.

With that decided and implemented I moved onto typecheck-expr where I implemented typechecking for newarrayI, arrayrefI, and arraysetI. Firstly, for each of these I needed to ensure that their respective numT inputs were indeed type numT. I did this by recurring on the index or length for each case respectively and checking if that type was equal to numT. If it was I moved on and if it wasn't I raised a type-error.

For newarrayI, after checking that the length was numT, I ensured that the type of the expression to be used for initialization was a subtype of the <Type> passed in. if is-subtype? Failed I raised a type error, and if it succeeded I returned the type indicated by the <Type> value initially passed in.

For arrayrefI, after verifying the index was a numT, I needed to ensure that the array expression was some type of array. Consequently I typecased on the recur of the array I-expression and if it was an arrayT returned this type otherwise I raised a type-error.

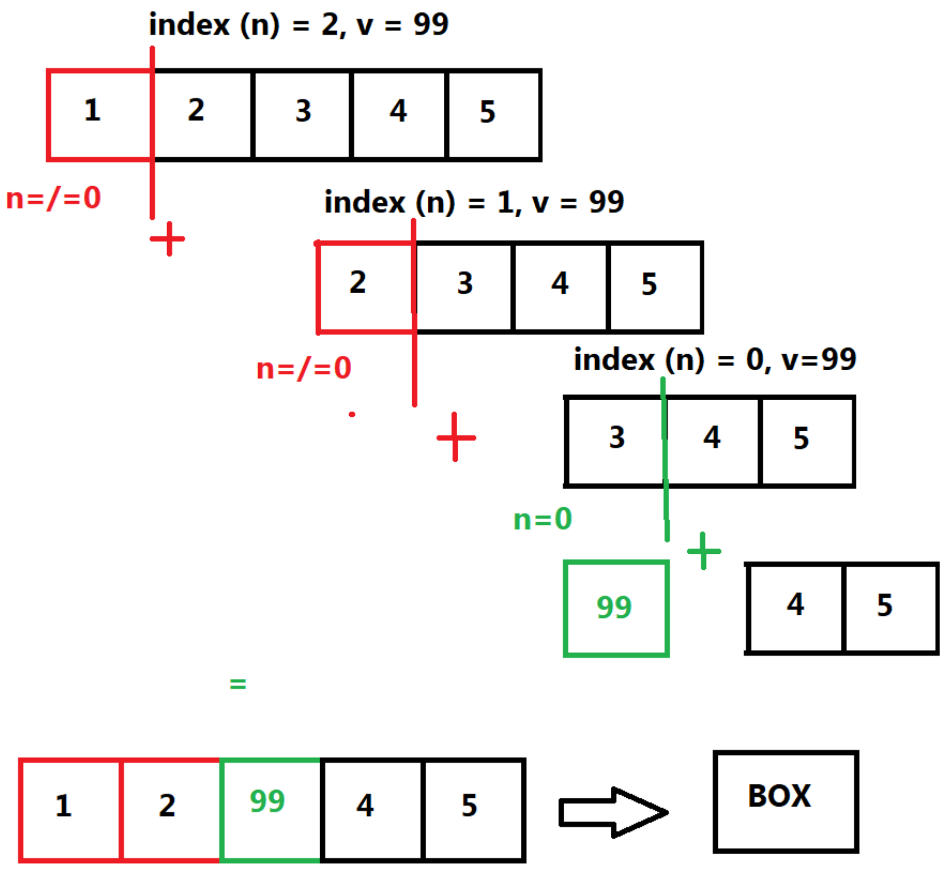
Finally, for arraysetI, after once again checking that the index was a numT, I just needed to verify that the type element held by the initial array expression (a-t) was an array type and subsequently supertype of the type of the expression being added to the array (v-t). Thus, I once again typecased on the type (a-t) found by recurring on the array expression a, raised an error if it was not of arrayT and continued onward if it was. Then, if (v-t) was a subtype of the type held by the arrayT (a-t) hence called (t), I returned (numT) as we return zero from arrayset otherwise I would throw another type-error.

Moving on to class.rkt and the interp function, I first defined newarrayE, arrayrefE, and arraysetE. Additionally, I added a new variant of value called arrayV which contains a field called values which has a (Boxof (Listof Value)) to track the items in the array and be able to set their values imperatively. Then I added cases for each of the relevant Expressions to the interp function.

In the newarray case, I merely ensured that the value of the length was a number via type-case and created an arrayV with values equal to the result of a call to a new function called construct-array which I promptly boxed. Construct-array takes a number referring to size, an empty list to be built upon, and a value and creates a list that of length n that is comprised all of the value given. It does this by recursively adding the value “v” onto the empty list “a” while decrementing the size “s” until s equals 0 at which point it returns the constructed list. I had to box this list for arrayV so that arraysetE can actually store the values inside this box instead of returning them allowing updates to be imperative.

For arrayref, I added a case to interp that first recurs on the array expression and ensures that we are attempting to access an array (otherwise providing an error) then ensures the index is a number by recurring on the index expression (once again providing an error if it is not a numV). Next, I chose to use list-ref to implement arrayref as it implements the same functionality for lists which back the array in this version of curly. However, I also chose to first catch my own out of bounds error if the index was less than zero or greater than the largest index in our list.

Finally, for arrayset, I once again ensured that the array and index were interpreted into the correct values (as with array ref) then updated the value of the list stored in the box with set-box! The new value stored in the box needed to be the list but with the value at index given replaced with the new value. Thus, a helper function "arr-list-set" was created and employed here. "Arr-list-set" takes in the current list of values stored in the box (vs), the index to be updated (n), and the new value (v) to replace the old value with. It works recursively by checking at each step that it is within the bounds of the list. If the index n is 0, the first item in vs it appends (v) on to the rest of (vs). Otherwise, it appends the first element in (vs) on to the result of "arr-list-set" for the rest of (vs) where the new n is smaller by one. This correctly replaces a given value in (vs) recursively by continually decreasing the size of the active list and shifting the index accordingly, consequently decreasing the scale of the problem at each step until the index is reached. Once again, as this uses a set-box! to update the values stored in arrayV this is an imperative update to this value.



The only other part of the implementation of arrays that is worth mentioning is testing this imperative update. Because I chose not to implement the local binding this was not a trivial task. To do this I realized that the only way to truly store a value was in an object's fields. Additionally, I realized that the simplest way to call two functions in order would be to implement begin, that way the 0 result of an arrayset is discarded and we can once-again get the field of the object we are using.

So I quickly implemented begin as beginI in the parser/typecheker, and beginE in the interpreter. Begin took two expressions (f) and (s), evaluated (f), then evaluated (s) keeping this result. Importantly, I ensured the typechecker returned the value of the type of (s) for the beginI case. In class.rkt, for the case in Interp, used plait's begin to implement this. I recurred on (f) discarded the result then recurred on (s) keeping this result.

With begin now implemented and a few tests written I went ahead and wrote a class to be used in testing called Arrays-set. The class Arrays-set has one field x that stores an array and two functions update-nine! and update-ref-four!. Update-nine! Begins by using arrayset to update the value of the x array to 9 at the index provided in the argument then returns the value of x. Update-ref-4 Begins similarly by using arrayset to update the value of the x array to 4 at the index provided in the argument then returning the value at that same index using arrayrefE. All of this allowed me to finally test that arrayset used an imperative update. This worked because begin allowed me to run two statements without using the result of the first and this object allowed me to store the value of x in a way that it could be retrieved in the (s) expression of begin.

8. Null:

My general implementation strategy for null was to implement Null on the type-checking end as objT 'null. I chose to do this because in this version of curly Null can only truly stand for an object as it can't be used as an array or number. Consequently, having null be represented inside of an object type eases the implementation and doesn't prevent what we need to do with null. Specifically it implicitly allows null to be used as an instance of an object without any modification. At this point I just needed to prevent other operations using null. I consequently just prevented classes from being named null, made the typechecker treat items with objT 'null slightly differently from others, made the interpreter return errors when attempting to call get and send on null values, and converted nullI into nullE into a nullV value despite not having a nullT type. Specifically I prevented the use of send and get for null objects in both the interpreter and typechecker.

To begin my implementation, I started in inherit.rkt where I defined nullI and converted it to nullE within the context of exp-i->c. Then in class I defined nullE. Neither nullI or nullE need any fields as null is a singular concept with no variation. Below this I defined a variant of Value called nullV once again with no fields. In the interp definition, we simply return nullV when we interpret a nullE. Still in interp, I now prevented null objects from being like regular objects in get, send and ssend using type-case. If the interpreted value of what their object they were called on was nullV I simply returned an error "object as null" thus, this prevents using these functions on a null object as all null objects will interpret to nullV.

After running some tests I wrote in class I moved on to typed-class. Here I firstly modified is-subtype? to essentially make it recognize null as a subtype and supertype of all other objects. Firstly it is important to note that since null is a type that is defined as an object it still cannot be used in the place of a number or array as far as typechecker is concerned. The reason that null should be a subtype and supertype of all other objects is so that it can be used in place of any object when testing object equality. So, to do this I essentially wrote that if both types were objT, if either type contained the name ‘null, then the subtype statement was true. As I did this for both t1 and t2 which are on opposing sides of the subtype operation, this means that objT ‘null is always correctly recognized as the subtype and supertype of any other Object type.

Next I used typecheck-expr to define the type of nullI I-Expressions by setting it to return objT ‘null. This implicitly gives all instances of the null expression the object type as I intended to define it. Then, similar to class’s interp, in typecheck-expr I checked getI and sendI for having object expressions where the resulting type had the ‘null class-name. If the object that get and send were trying to be called on were null, then I simply returned that the “object was null and thus had no definition.”

Finally, I finished the implementation of null in typed-parse and inherit-parse by doing the same in both. I firstly disallowed null as a class name in their specific parse-class methods, returning invalid input if null was found as a class name. This was necessary as implementing a null class would prevent such a class from being used as all calls to get and send would be denied. This made sense as java uses null as a keyword and thus it cannot be the name of a class under any normal circumstances. Additionally, I added a case for their respective interp-prog functions that returns the s-expression `null as a result for a nullV value being output. Thus this concluded the implementation of null.

This all works as in the typechecker curly treats nullI objects as having objT ‘null thus allowing them to be used in the place of other objects since null is a subtype and supertype of every other object. The only places where they are not allowed are in get and send where they are specifically banned from use. Otherwise the typechecker sees them as valid no matter what. For interp, on the other hand since a nullE returns nullV we simply return an error from get and send when this value is interpreted. Otherwise a nullE is just outputted as nullV a null value.