CSE 3302: Simpl Interpreter Project

Mai Tran

November 28, 2023

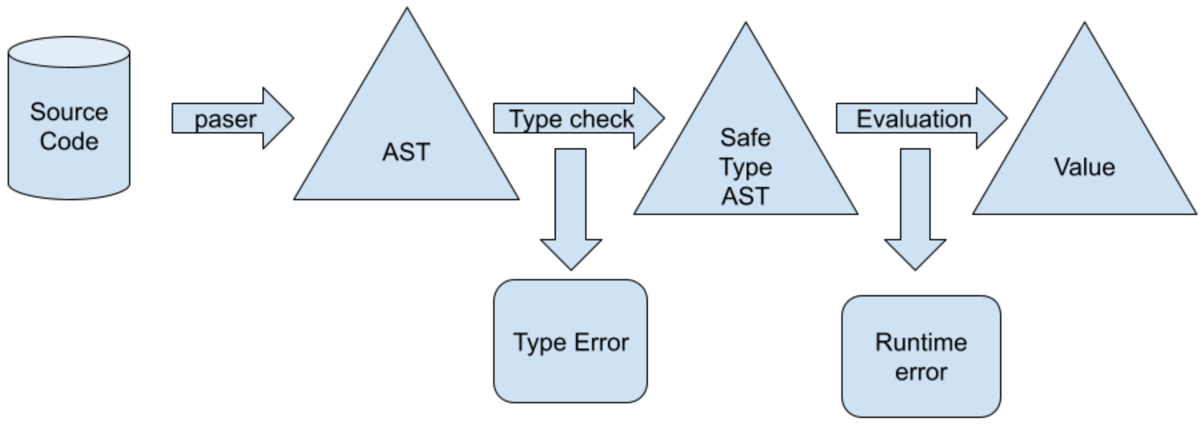
1 Academic Integrity

I have neither given nor received unauthorized assistance on this work. I will not post the project description and the solution online.

Sign: Mai Tran Date:November 28, 2023

2 Project Structure

I began by examining the entry point of the project, which is the Interpreter class within the Simpl.interpreter package. I constructed a simplified model (not necessarily accurate) of how Simpl interprets the source code as outlined below



Vestibulum congue

2.1 Parser

The fully implemented parser in the skeleton code transforms the source code into an abstract syntax tree (AST). For instance, if the input plain text is something like ”iszero 30”, it will be parsed into an “application” node and the parser will automatically generate an App object corresponding to this node. It is my responsibility to define TypeCheck and Evaluation rules for this App class, as well as for all other symbols in the AST.

2.2 Type Check

The TypeCheck process begins from the root expression. For example, if the root expression is an App object, TypeCheck is performed using the TypeCheck method of the App class. This is made possible thanks to Java’s Subtyping Polymorphism, as each different symbol (or node in the AST) will have a different implementation of the TypeCheck method. If a type error is detected, the interpreter halts, raises a type error, and stops the evaluation process. If the type of check is successful, indicating that this program is type-safe, the process proceeds to evaluation step.

2.3 Evaluation

The Evaluation procedure is quite similar to the TypeCheck process. The correct implementation of the Eval method is called, once again, benefiting from subtyping. If any of these evaluations encounter an impasse, a runtime error is thrown. If the evaluations complete successfully, the final evaluated value is printed, which can be a constant or an abstract value like ”fun”.

3 Getting started

3.1 Fill out typing package

I begin by completing the typing package. There 4 Classes that need to be understanding before implementing. These include “TypeEnv”, “DefaultTypeEnv”, “Substitution”, and “TypeResult”

3.1.1 TypeEnv

In brief, a TypeEnv is a recursively defined object where each instance contains both another object of the same type and a mapping between symbols (e.g., ”x”) and their corresponding types (e.g., ”t”). When you request the type of a symbol ”x” from the current TypeEnv object, it first checks if there is a mapping for ”x” at the current level. If a mapping is found, it returns the associated type; otherwise, it invokes the previous TypeEnv in the chain to search for the mapping. The base case is an ”empty” TypeEnv, which returns NULL for all symbols ”x.”

3.1.2 DefaultTypeEnv

The DefaultTypeEnv can be left as an empty type environment because all the predefined types, such as integer literals or Boolean literals, are detected by the parser and automatically instantiated with their corresponding types. However, for built-in functions like iszero, head, tail, and others whose types are not instantiated by the parser, I need to manually assign types to them within the default type environment.

3.1.3 Substitution

Substitution shares similarities with TypeEnv as it also relies on a recursion. A substitution is essentially a composition of multiple substitutions. The simplest form of substitution is Identity, which essentially means that it results in no change. In contrast, Replace is a ”single” substitution, involving a sole rule for mapping type variables denoted as ”a” to a specific type ”t.”. One crucial aspect to grasp is that the “apply” method of Replace does not perform the actual replacement process itself; instead, it supplies the mapping to the type ”A” on which it is invoked. This approach makes sense because types can take on various forms, with some, like Arrow Types, being combinations of two types. Therefore, the substitution should be carried out on both of these components. In other words, Substitution functions as a database that provides information, and how the actual "substitution" is implemented will be determined by the specific Type passed in as the parameter.

3.1.4 TypeResult

TypeResult is a product of the TypeCheck method, which will be explained in detail later in this report. In general, a TypeResult is a combination of both a Type and a Substitution. The rationale behind this definition is the notion that when performing type checking on an untyped expression, we should obtain two pieces of information: the first being the returned type (which can either be a concrete type or a type variable), and the second being the set of constraints gathered during the type checking process. The presence of substitution in the return value is critical because without it, the return type might not be a valid type, it is hyphothesis under which we can state that the expression will have type 't'. .The implementation of the TypeCheck method will be placed in the Abstract Syntax Tree (AST) of each node, and this choice is justified by the close association between the Type we generate and the syntactic constraints of the program.

3.1.5 Implementation Procedure

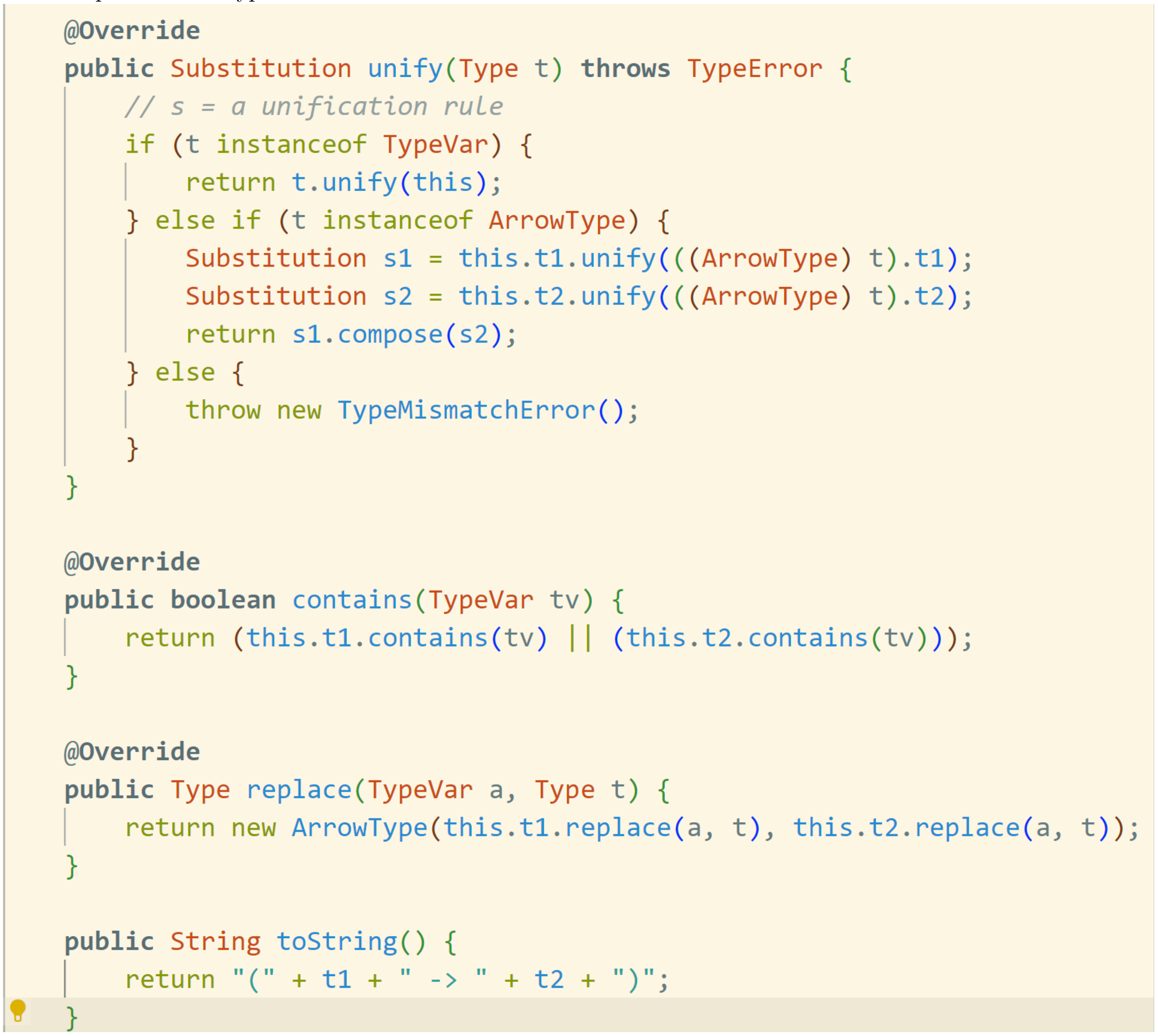
The process of completing the typing package becomes quite straightforward once the concepts outlined in the preceding section are grasped. When implementing the unification method, there are three crucial considerations:

1. When invoking unify on a type variable, it is essential to reverse the direction. This ensures that the type variable calls the unify method on a more specific type, aligning with the u-var 2 rule in the lecture slides.

2. If unify is called on a type variable of the same shape, the unification method can be invoked on the smaller type if the current type make up of other types or begins replacing the type variable if they are small enough.

3. If neither of the previous cases succeeds, it indicates either a type mismatch (such as calling unify with INT on BOOL) or a Type circularity error in the case of TypeVar.

Example of ArrowType



```
@Override
public Substitution unify(Type t) throws TypeError {
    // s = a unification rule
    if (t instanceof TypeVar) {
        return t.unify(this);
    } else if (t instanceof ArrowType) {
        Substitution s1 = this.t1.unify(((ArrowType) t).t1);
        Substitution s2 = this.t2.unify(((ArrowType) t).t2);
        return s1.compose(s2);
    } else {
        throw new TypeMismatchError();
    }
}

@Override
public boolean contains(TypeVar tv) {
    return (this.t1.contains(tv) || (this.t2.contains(tv)));
}

@Override
public Type replace(TypeVar a, Type t) {
    return new ArrowType(this.t1.replace(a, t), this.t2.replace(a, t));
}

public String toString() {
    return "(" + t1 + " -> " + t2 + ")";
}
```

3.2 Fill out Interpreter package

If the typing package denotes the return type of the program's expression after type-checking steps. The interpreter package comprises classes that represent the return value after invoking the eval method. Similar to the typing package, there are four key classes that need to be comprehended before completing the remaining class. These classes are Env.java, State.java, InitialState.java, and Mem.java.

3.2.1 Env.java

Its definition closely resembles that of TypeEnv in the typing package. However, instead of mapping between symbols and types, Env is responsible for mapping symbols to their corresponding values.

These values can be FunValue, RefValue, and others, which will be implemented subsequently.

3.2.2 Mem.java

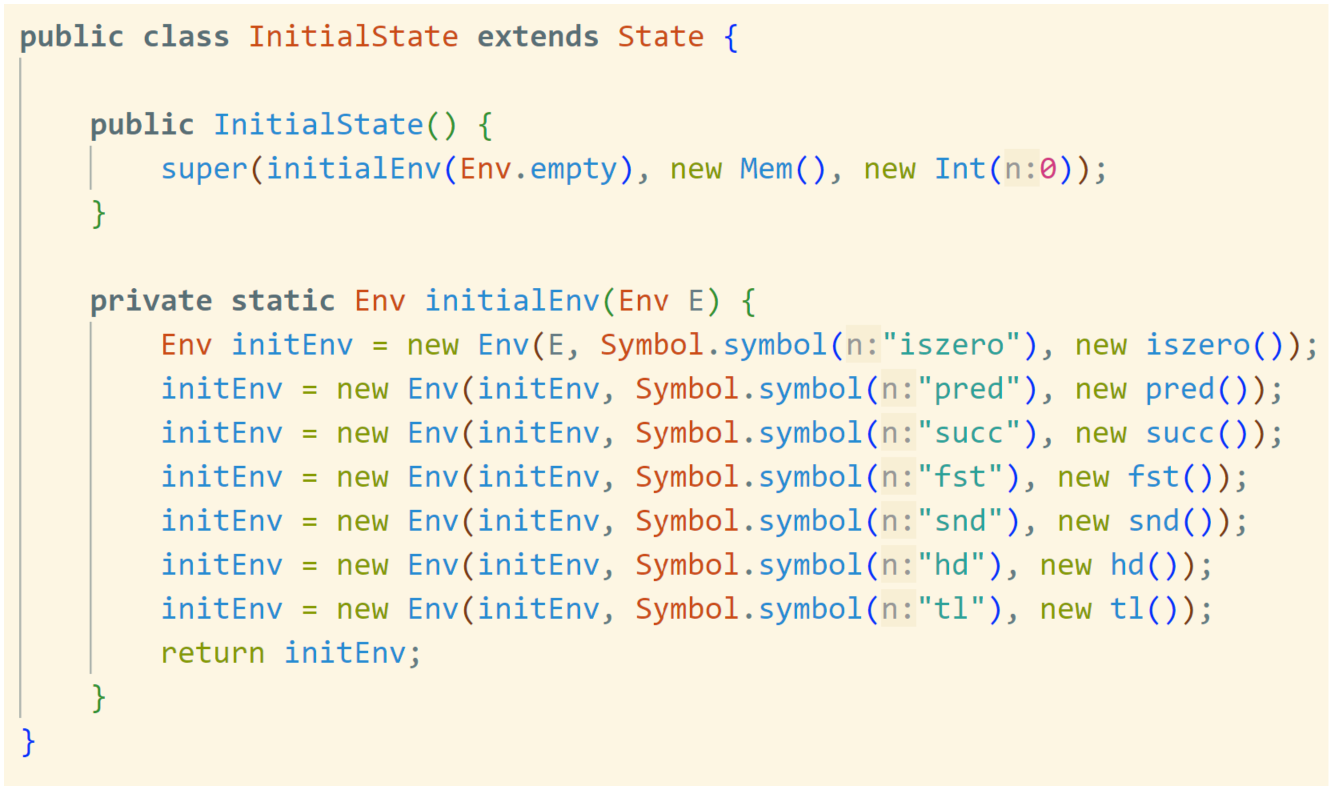
This extends the Java hashmap and serves as a simulation of the heap in a real programming language. To achieve this simulation, specific methods such as read (equivalent to hashmap's get), write (equivalent to hashmap's put), and alloc (creating a new entry in the hashmap table) must be implemented. Mem includes a final static int called ptr, ensuring a unique value throughout the program's runtime. This integer guarantees that the next allocated memory cell will never be repeated.

3.2.3 State.java

This class consists of three fields: Env (implemented in Env.java), Mem (implemented in Mem.java), and p (an integer pointer, as explained above). Its purpose is to maintain the program's state, as depicted in the lecture.

3.2.4 InitialState.java

As briefly mentioned in DefaultTypeEnv, when the parser encounters a defined symbol like 5, it automatically assigns this symbol the IntType (implemented in the typing package). The same concept applies here: when the parser encounters 5, it also instantiates an IntValue attached to it. However, predefined functions like "iszero," "succ," etc., still need a value attached to them. InitialState.java is where we add these Value mappings to the initial state.



```java
public class InitialState extends State {

    public InitialState() {
        super(initialEnv(Env.empty), new Mem(), new Int(n:0));
    }

    private static Env initialEnv(Env E) {
        Env initEnv = new Env(E, Symbol.symbol(n:"iszero"), new iszero());
        initEnv = new Env(initEnv, Symbol.symbol(n:"pred"), new pred());
        initEnv = new Env(initEnv, Symbol.symbol(n:"succ"), new succ());
        initEnv = new Env(initEnv, Symbol.symbol(n:"fst"), new fst());
        initEnv = new Env(initEnv, Symbol.symbol(n:"snd"), new snd());
        initEnv = new Env(initEnv, Symbol.symbol(n:"hd"), new hd());
        initEnv = new Env(initEnv, Symbol.symbol(n:"tl"), new tl());
        return initEnv;
    }
}
```

3.2.5 Procedure for Implementation

Once again, the process of completing the interpreter package becomes straightforward after understanding the concepts outlined in the preceding section. When implementing the equals method, certain patterns need to be considered:

1. First, check if the other object is of the same type; if not, simply return false.
2. If they are of the same value type, check if the corresponding values of this particular value type are equal. This might involve recursive calls, as seen in the case of ConsValue (checking if the heads are equal and subsequently calling recursive equal on the rest of the array).
3. For special values like unit, we can simply return true if the object passed in as an argument is also of the Unit value.

Example of PairValue implementation

```
public class PairValue extends Value {

    public final Value v1, v2;

    public PairValue(Value v1, Value v2) {
        this.v1 = v1;
        this.v2 = v2;
    }

    public String toString() {
        return "pair@" + v1 + "@" + v2;
    }

    @Override
    public boolean equals(Object other) {
        return (other instanceof PairValue) && (this.v1.equals(((PairValue) other).v1))
                && (this.v2.equals(((PairValue) other).v2));
    }
}
```

3.3 Fill out Parser package

This is the package where we need to determine how to correctly invoke the type check and eval methods, update the type environment and state accurately, ensuring that Simple will interpret the source code correctly. With the typing and interpreter components already completed, the primary challenge lies in simply calling the appropriate methods in all the classes included in the AST folder. There are two basic pattern that can be follow for typeCheck and eval

1. Type Check

    (a) Invoke the type check on smaller expressions (if possible) to collect type results.

    (b) Generate a combined substitution by composing all the collected substitutions together.

    (c) Reapply the new substitution to the type returned by the type check in step 1.

    (d) Now, we have consistent types under the same constraints; initiate the unification process based on the syntactic rules learned in class.

    (e) Incrementally expand the combined substitution by composing it with the new constraints returned by the unification step.

    (f) With the final substitution obtained, apply it to the type that is supposed to be the return type.

    (g) Construct a type result to return, including the final return type and the final substitution.

2. Evaluation

   (a) Invoke the eval method on the smaller expression (if possible) to collect the value result.

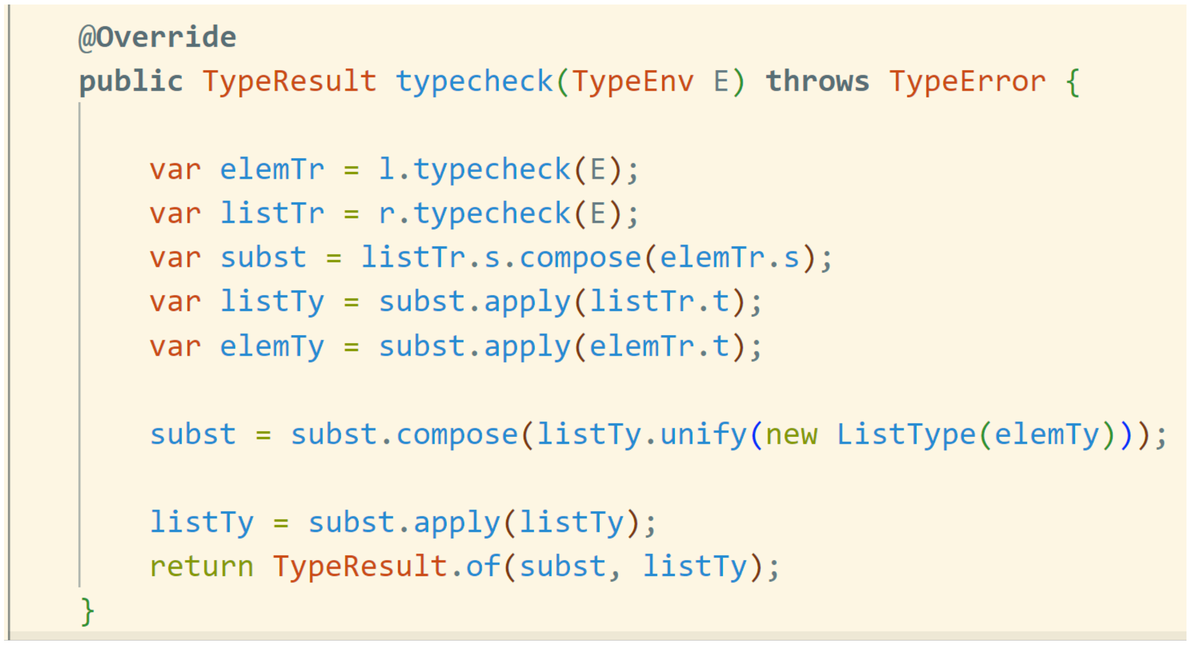
   (b) Use an if condition to double-check the return value and validate it based on syntactic rules. For example, if the current evaluated expression is Add, the evaluated value of lhs must be of type IntValue, and the evaluated value of rhs must also be of type IntValue. Although this may not be necessary since the program has already passed the type-check step, it is included for the completeness of runtime error handling.

   (c) After validating that the evaluated results are of the correct value type, return a new Value based on syntactic rules. For example, if the current expression is Add, return a new IntValue whose n is the sum of lhs.n and rhs.n.

3.3.1 Procedure for Implementation

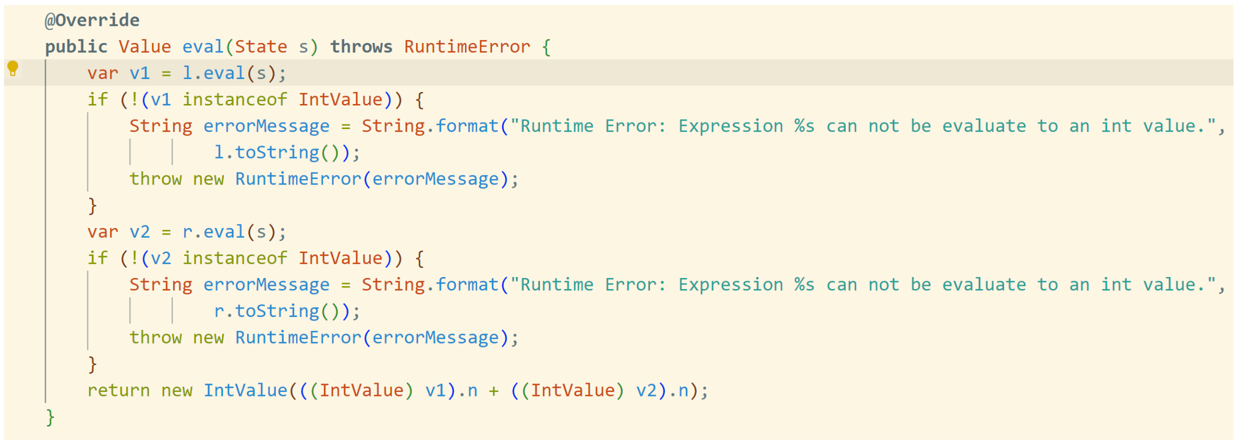
Following the explained pattern above, here are examples of its code for Cons type check and Add evaluation:

Cons typeCheck:



```java
@Override
public TypeResult typecheck(TypeEnv E) throws TypeError {

    var elemTr = l.typecheck(E);
    var listTr = r.typecheck(E);
    var subst = listTr.s.compose(elemTr.s);
    var listTy = subst.apply(listTr.t);
    var elemTy = subst.apply(elemTr.t);

    subst = subst.compose(listTy.unify(new ListType(elemTy)));

    listTy = subst.apply(listTy);
    return TypeResult.of(subst, listTy);
}
```

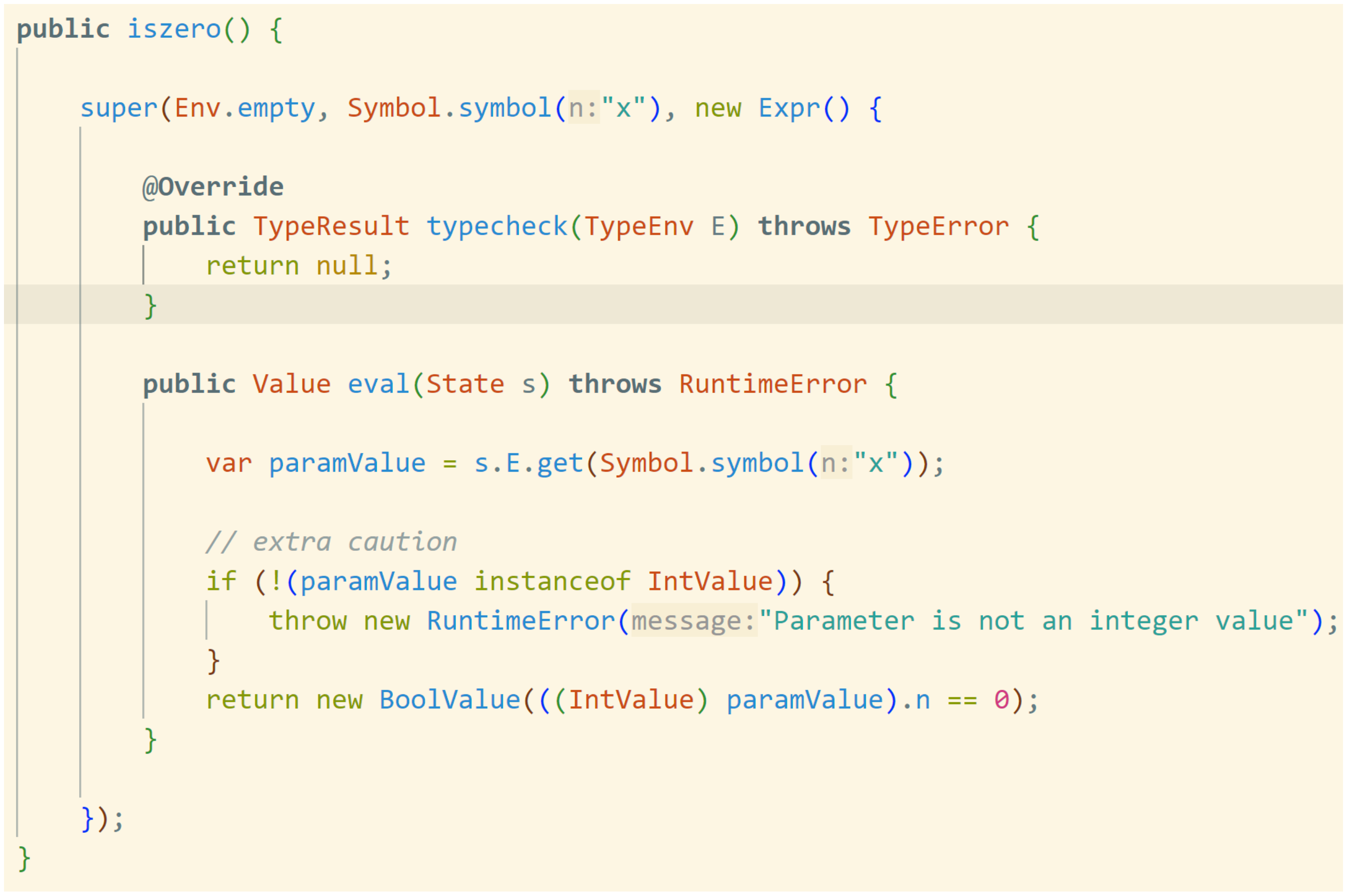
Add Evaluation:



```
@Override
public Value eval(State s) throws RuntimeError {
    var v1 = l.eval(s);
    if (!(v1 instanceof IntValue)) {
        String errorMessage = String.format("Runtime Error: Expression %s can not be evaluate to an int value.",
                l.toString());
        throw new RuntimeError(errorMessage);
    }
    var v2 = r.eval(s);
    if (!(v2 instanceof IntValue)) {
        String errorMessage = String.format("Runtime Error: Expression %s can not be evaluate to an int value.",
                r.toString());
        throw new RuntimeError(errorMessage);
    }
    return new IntValue(((IntValue) v1).n + ((IntValue) v2).n);
}
```

3.4 Completing Predefined Functions

Having already added predefined function types and values to the initialState and default Env, the remaining task involves completing their evaluation methods. There is a pattern to follow for evaluation: Given that these functions are extensions of FunValue, we can be certain that they will appear in the lhs of the "App" symbol. Assuming that App type check is successful and the evaluation is completed, we will have the binding between the parameter "x" and the evaluated value of the Application rhs. All that remains is to search for that value and make the correct decision based on it. An example is attached below with the code for iszero.



```java
public iszero() {

    super(Env.empty, Symbol.symbol(n:"x"), new Expr() {

        @Override
        public TypeResult typecheck(TypeEnv E) throws TypeError {
            return null;
        }

        public Value eval(State s) throws RuntimeError {

            var paramValue = s.E.get(Symbol.symbol(n:"x"));

            // extra caution
            if (!(paramValue instanceof IntValue)) {
                throw new RuntimeError(message:"Parameter is not an integer value");
            }
            return new BoolValue(((IntValue) paramValue).n == 0);
        }

    });
}
```

3.5 Bonus Features

3.5.1 Type Error detailed output

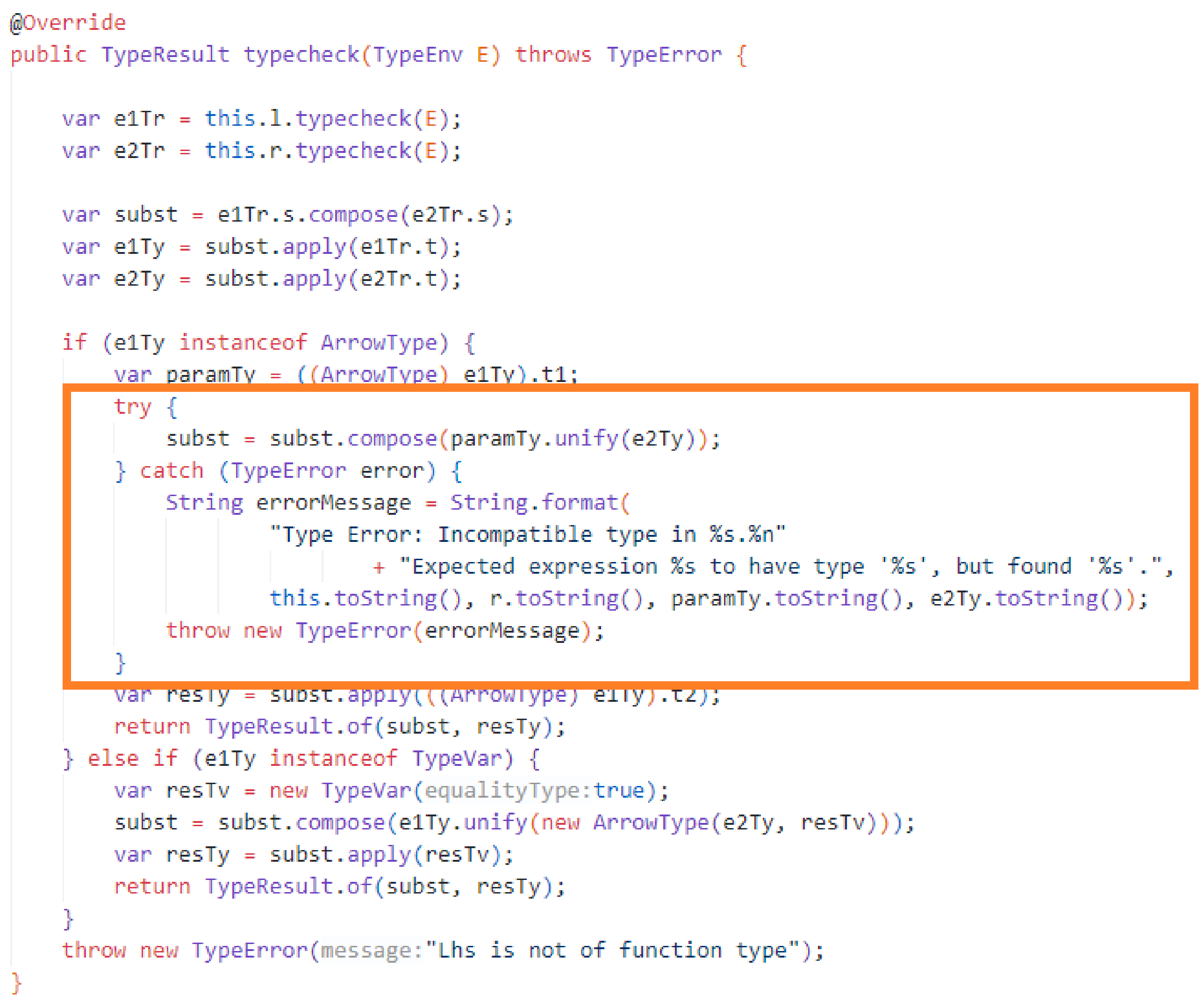
Detecting type errors by having the program generate a corresponding error message is a valuable practice. However, as the program's size increases, this approach becomes excessively abstract. Programmers may encounter situations where they are uncertain about the specific location in the source code responsible for the type error.

To tackle this challenge, it is recommended to incorporate this functionality into all Abstract Syntax Tree (AST) classes of the parser package. This decision is justified for the following reasons:

1. The AST class encapsulates information about the expression's structure. This not only enables the detection of type errors but also facilitates the identification of the exact expression causing the error.

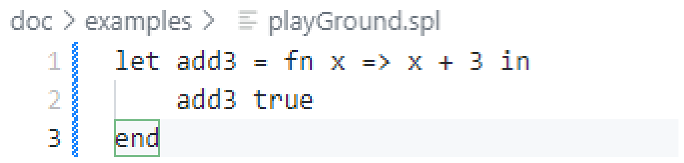
2. In the event of a type error, the unification algorithm will generate a corresponding error message. The AST classes serve as optimal locations to capture these error messages, enhance them with additional structural information, and then propagate the improved error message back to the Interpreter class.

Example of implementation in Let.java:



```java
@Override
public TypeResult typecheck(TypeEnv E) throws TypeError {

    var e1Tr = this.l.typecheck(E);
    var e2Tr = this.r.typecheck(E);

    var subst = e1Tr.s.compose(e2Tr.s);
    var e1Ty = subst.apply(e1Tr.t);
    var e2Ty = subst.apply(e2Tr.t);

    if (e1Ty instanceof ArrowType) {
        var paramTy = ((ArrowType) e1Ty).t1;
        try {
            subst = subst.compose(paramTy.unify(e2Ty));
        } catch (TypeError error) {
            String errorMessage = String.format(
                    "Type Error: Incompatible type in %s.%n"
                            + "Expected expression %s to have type '%s', but found '%s'.",
                    this.toString(), r.toString(), paramTy.toString(), e2Ty.toString());
            throw new TypeError(errorMessage);
        }
        var resTy = subst.apply(((ArrowType) e1Ty).t2);
        return TypeResult.of(subst, resTy);
    } else if (e1Ty instanceof TypeVar) {
        var resTv = new TypeVar(equalityType:true);
        subst = subst.compose(e1Ty.unify(new ArrowType(e2Ty, resTv)));
        var resTy = subst.apply(resTv);
        return TypeResult.of(subst, resTy);
    }
    throw new TypeError(message:"Lhs is not of function type");
}
```

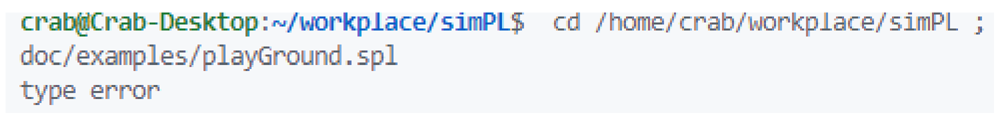
Let's consider an example of a type error version and a modified version for this SimPl program:



doc > examples > playGround.spl

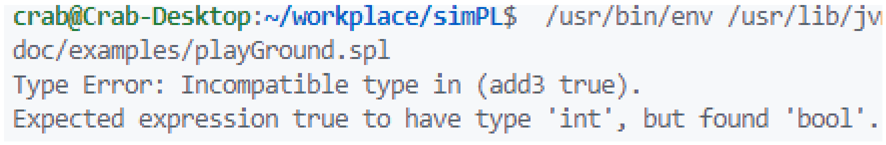
```
let add3 = fn x => x + 3 in
    add3 true
end
```

Without detailed output:



```
crab@Crab-Desktop:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
doc/examples/playGround.spl
type error
```

With detailed output:



```
crab@Crab-Desktop:~/workplace/simPL$  /usr/bin/env /usr/lib/jv
doc/examples/playGround.spl
Type Error: Incompatible type in (add3 true).
Expected expression true to have type 'int', but found 'bool'.
```

3.5.2 Garbage Collection

Garbage collection is a compelling illustration of the essence of software engineering, where it practically provides no interface for users or programmers to acknowledge its existence. Behind the scenes, the garbage collector (GC) efficiently collects all unused or lost memory cells and returns the reclaimed space to the heap. In our SimPl project, the only place where we allocate additional memory space (by adding a new entry to the hash table) is in `Mem.java` within the `interpreter` package. Therefore, to introduce GC features, modifications are required only in `Mem.java`.

I have chosen to implement a copying collection GC strategy as taught in class. To begin, there are a couple of necessary changes:

1. To traverse the entire program heap (`Env` in the case of our SimPl implementation), we need a method in `Mem` class that allows access to the entire `Env` structure. To simplify the code, I have changed the visibility of `Env` in the `Env` class to public. Similarly, to access the current value for each symbol, I have added an additional public method called `getVal` to retrieve the current environment value.

2. To test if GC is working correctly, we need a way to simulate the limit of heap space. I have achieved this by adding a new class name Features.java that included Boolean field to turn on/off features as well as memory size limit.

After making all the necessary visibility changes, the remaining task is to implement GC by adding additional code to the `alloc` method of `Mem`. The algorithm breakdown is as follows:

1. **Phase 1**

   (a) Create a temporary `ArrayList` of type `Value`, essentially acting as the `to_Space` introduced in class during the Memory Management lecture.

   (b) Create an index pointer to keep track of the next position to insert into the `to_Space`. This pointer simulates (and ensures) the elimination of fragmentation by incrementing by one after each iteration.

   (c) Begin looping through the `Env` (base case: `Env == NULL`, incremental step: `Env = Env.E`). For each iteration, check whether the current value is a `Ref` value. If yes, store that `Ref` value in the temporary array and set the pointer of the current value to the index, then increase the index by one. If no, continue. This step is essentially a forwarding address in Cheney's algorithm; we update the address to the new continuous address space, but the actual contents at that address are not ready yet (currently stored in the temporary array).

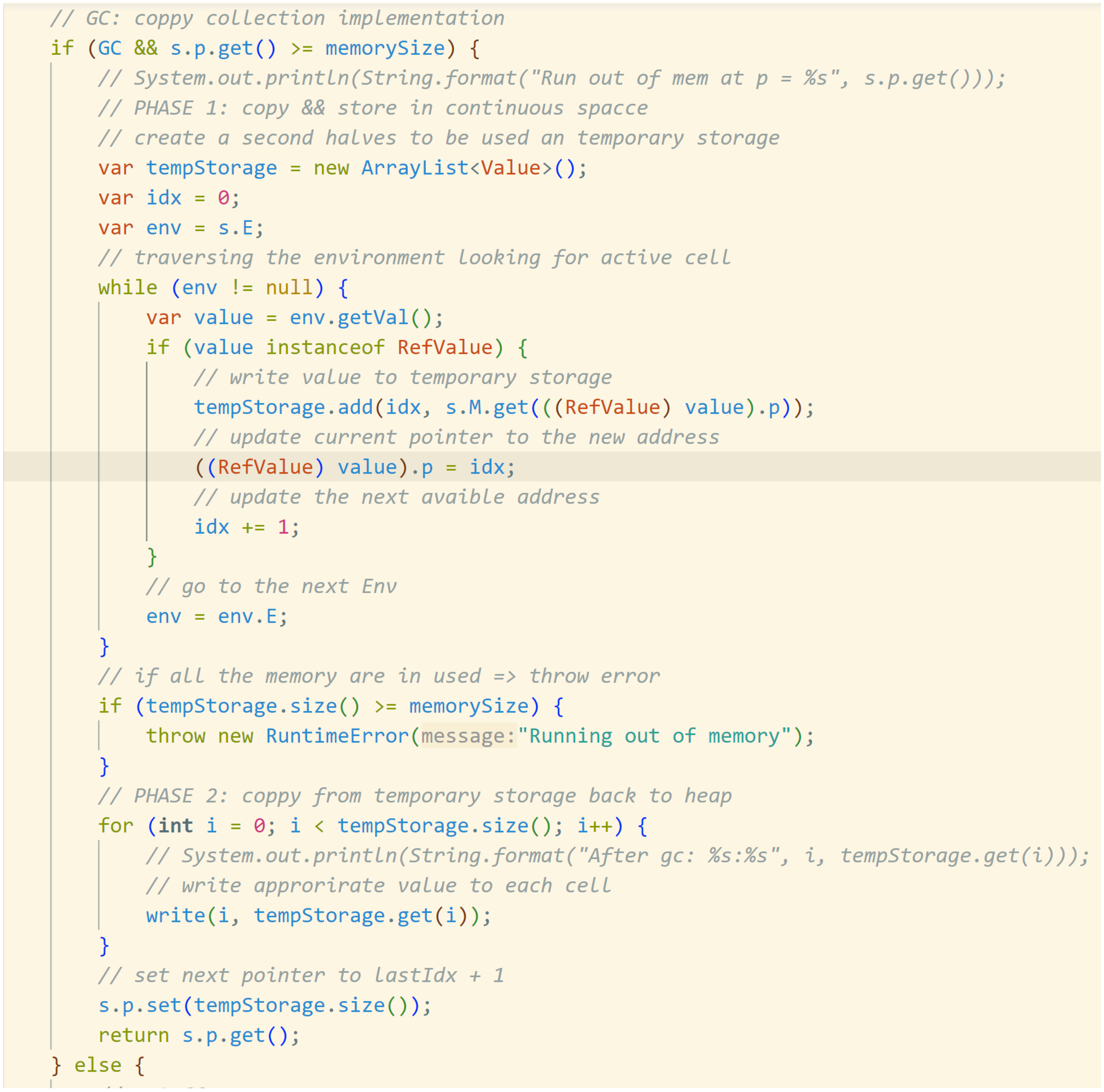
2. **Out of Memory Check:** After phase 1, use an if condition to check the size of the temporary array. If its size is still greater than `memorySize`, we know that all the memory is currently in use, and in fact, this is the scenario where we have nothing to collect due to insufficient memory. If so, throw an out-of-memory error. If not, proceed to phase 2.

3. **Phase 2: Copy from Temporary Array Back to Heap**

   (a) Use a for loop to iterate through the temporary array and write the correct values back to the intended address.

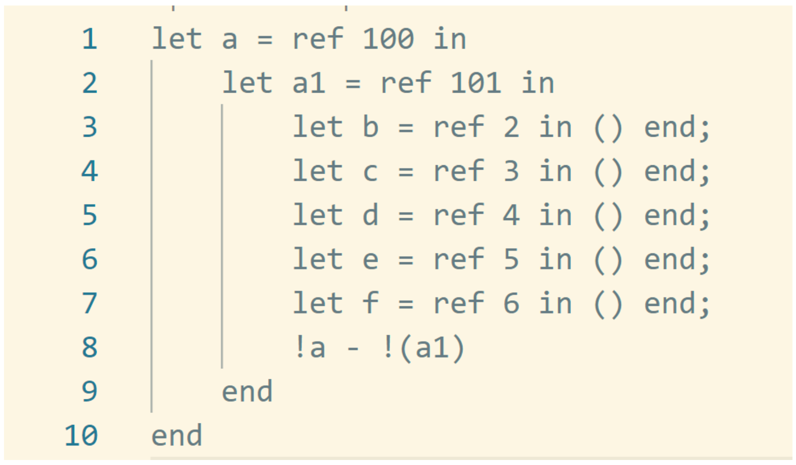
   (b) After finishing writing, the last index + 1 is the next memory cell that can be allocated. Set the next pointer (`p`) to this position and return `s.p.get` to the caller.

Example of implementation in Mem.java



```
// GC: coppy collection implementation
if (GC && s.p.get() >= memorySize) {
    // System.out.println(String.format("Run out of mem at p = %s", s.p.get()));
    // PHASE 1: copy && store in continuous spacce
    // create a second halves to be used an temporary storage
    var tempStorage = new ArrayList<Value>();
    var idx = 0;
    var env = s.E;
    // traversing the environment looking for active cell
    while (env != null) {
        var value = env.getVal();
        if (value instanceof RefValue) {
            // write value to temporary storage
            tempStorage.add(idx, s.M.get(((RefValue) value).p));
            // update current pointer to the new address
            ((RefValue) value).p = idx;
            // update the next avaible address
            idx += 1;
        }
        // go to the next Env
        env = env.E;
    }
    // if all the memory are in used => throw error
    if (tempStorage.size() >= memorySize) {
        throw new RuntimeError(message:"Running out of memory");
    }
    // PHASE 2: coppy from temporary storage back to heap
    for (int i = 0; i < tempStorage.size(); i++) {
        // System.out.println(String.format("After gc: %s:%s", i, tempStorage.get(i)));
        // write approrirate value to each cell
        write(i, tempStorage.get(i));
    }
    // set next pointer to lastIdx + 1
    s.p.set(tempStorage.size());
    return s.p.get();
} else {
```

Test the result with this straightforward program:



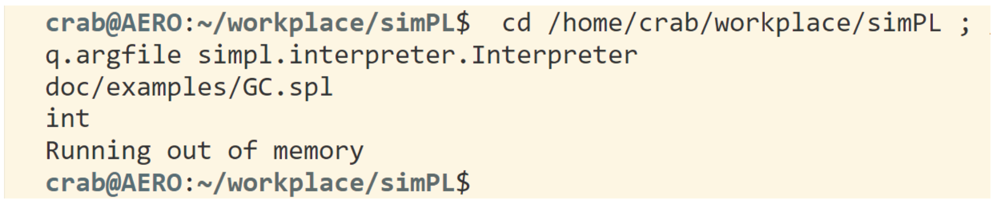
```
1   let a = ref 100 in
2       let a1 = ref 101 in
3           let b = ref 2 in () end;
4           let c = ref 3 in () end;
5           let d = ref 4 in () end;
6           let e = ref 5 in () end;
7           let f = ref 6 in () end;
8           !a - !(a1)
9       end
10  end
```

I tested by set memory cell limit of 3. As the program advance to line 4 and attempt to allocate one more cell for 'c', it will encounter the 3-memory cell limit (cell of 'a' + cell of 'a1' + cell of 'b'). Consequently, we run out of memory. However, with 'b' going out of scope, a proper garbage collection (GC) implementation should reclaim the memory cell from 'b' and assign it to 'c'. This would result in just enough memory cells for the correct evaluation of the entire program. Therefore, the expected results are:

1. 3 memory cells without GC: running out of memory
2. 3 memory cells with GC: correctly evaluated to the end
3. 2 memory cells with GC: running out of memory (we need at least 3 since a and a1 alway in scope)

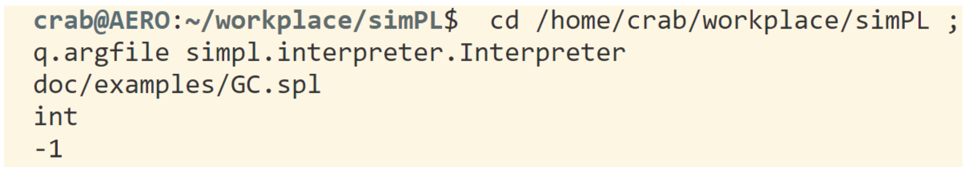
Testing result:

Without GC and memory size set to 3:



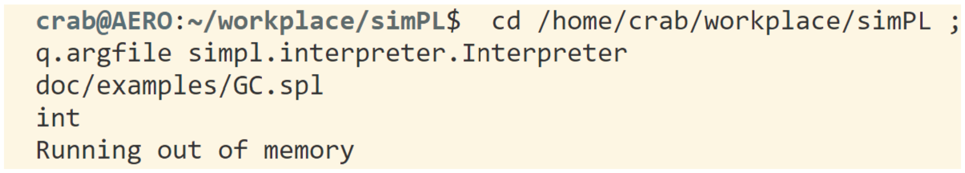
```
crab@AERO:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
q.argfile simpl.interpreter.Interpreter
doc/examples/GC.spl
int
Running out of memory
crab@AERO:~/workplace/simPL$
```

With GC and memory size set to 3:



```
crab@AERO:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
q.argfile simpl.interpreter.Interpreter
doc/examples/GC.spl
int
-1
```

With GC and memory size set to 2:


```
crab@AERO:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
q.argfile simpl.interpreter.Interpreter
doc/examples/GC.spl
int
Running out of memory
```

3.5.3 Lazy Evaluation

The concept underlying lazy evaluation stems from a fundamental question: why evaluate an expression that will never be used? Specifically, in the context of the SimPL interpreter, when a variable is bound to an expression (potentially reducible to a value), the evaluation process is deferred until the first invocation of that expression within the associated context or environment. The term "first" is crucial here, as the intentional delay in the evaluation process necessitates the use of a placeholder value. Consequently, if this placeholder value appears multiple times in the body expression, the same expression may need to be re-evaluated repeatedly, which is a wasteful of resource. In essence, the correct implementation of lazy evaluation requires achieving two key objectives:

1. Postponing the evaluation process whenever there is a binding between two entities (in the case of SimPL, between a variable and an expression).

2. Implementing a mechanism to ensure that the expression is evaluated at most once (no evaluation if never used).

To address these objectives, I chosed to introduce a special runtime value called a "thunk," (idea from Haskell's lazy evaluation). This decision offers several advantages:

1. As I implemented ThunkValue as an subtype of Value, ThunkValue can be used wherever a Value type is required. Although a thunk is not technically a Value (as it is not yet evaluated), it is treated as such in all other semantic evaluation classes, requiring no modifications to those classes.

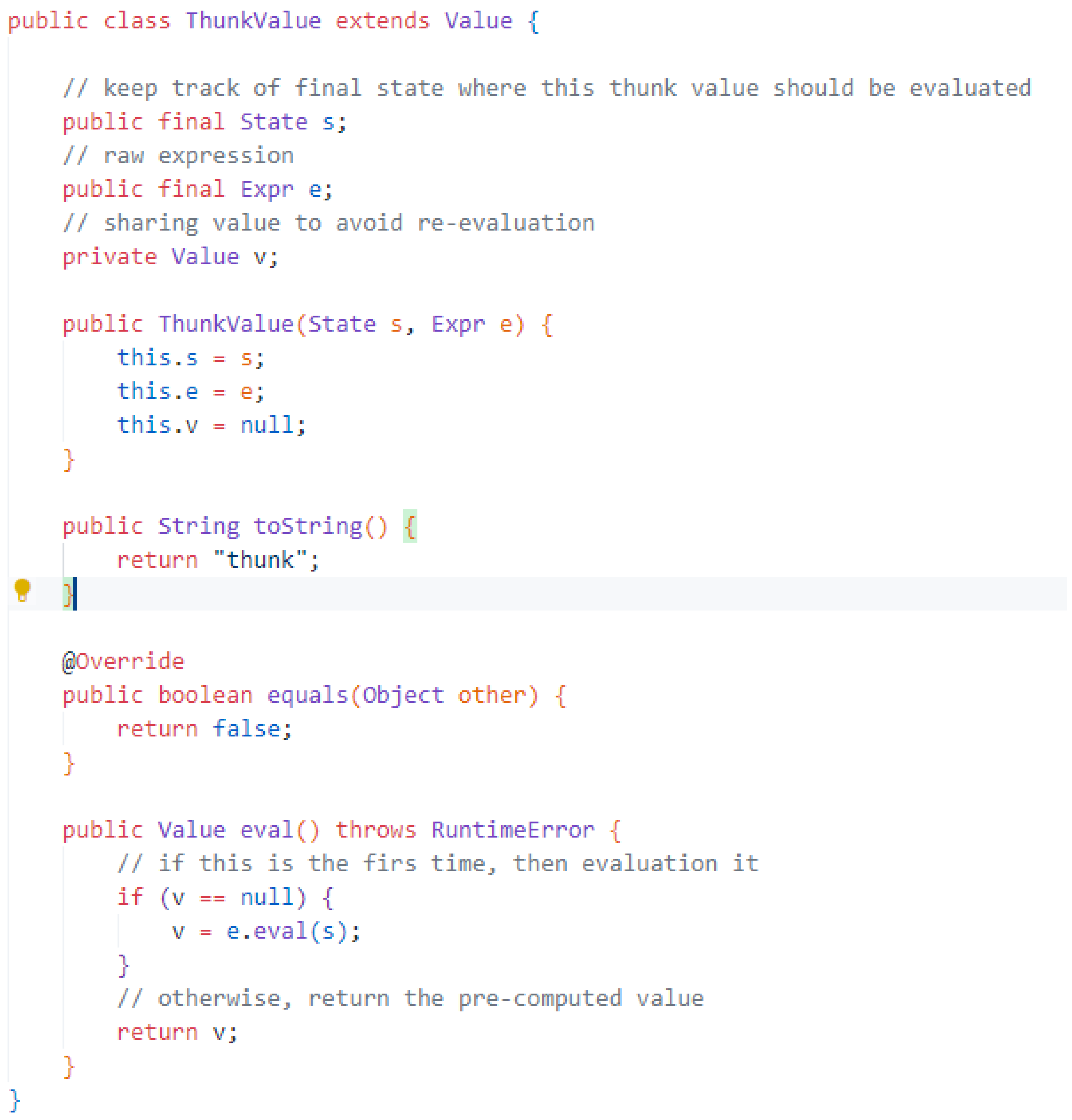
2. Given the use of lazy evaluation, there is a need to capture the current program state to evaluate the expression at a later point. This is achieved by adding a State field into the ThunkValue class and fixing it as the initialization point.

3. Memorization is crucial to avoid re-evaluating the same expression multiple times. To achieve this, I added a Value field (v) to the ThunkValue class. Initially set to NULL, it serves as an

indicator for the first evaluation. After the initial evaluation, the result is cached in v, preventing redundant re-evaluations.

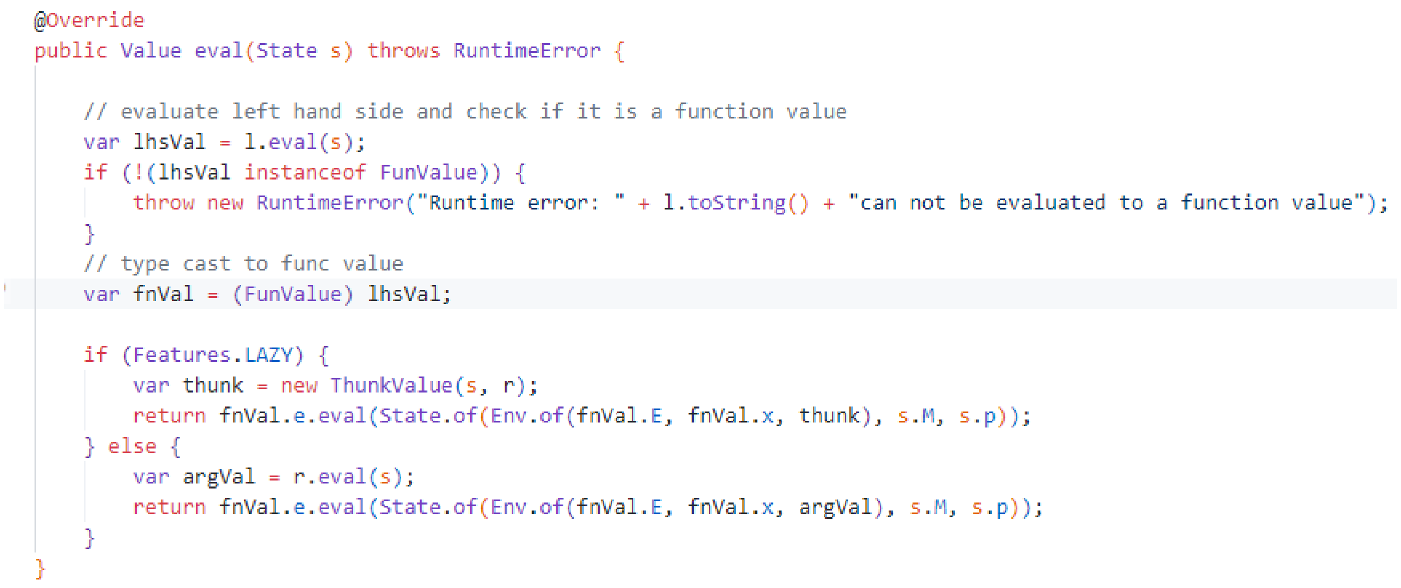
After implementing these modifications, along with minor adjustments in Name.java and predefined functions, the remaining changes involve adjusting the evaluation order in Let and App. Instead of evaluating the parameter immediately, a new thunk value is created, capturing the parameter's expression and the current program state. Subsequently, the body is evaluated in the new environment.

ThunkValue.java Class:



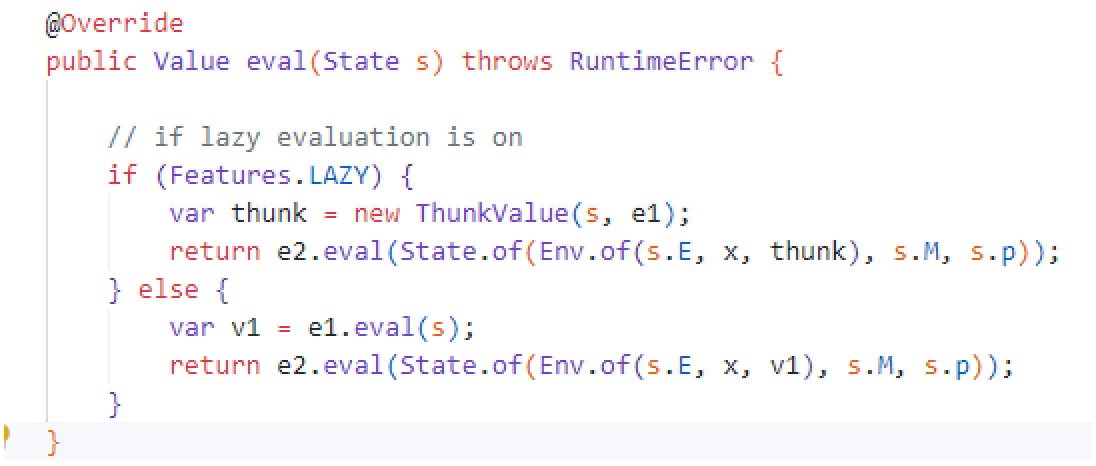
```java
public class ThunkValue extends Value {

    // keep track of final state where this thunk value should be evaluated
    public final State s;
    // raw expression
    public final Expr e;
    // sharing value to avoid re-evaluation
    private Value v;

    public ThunkValue(State s, Expr e) {
        this.s = s;
        this.e = e;
        this.v = null;
    }

    public String toString() {
        return "thunk";
    }

    @Override
    public boolean equals(Object other) {
        return false;
    }

    public Value eval() throws RuntimeError {
        // if this is the firs time, then evaluation it
        if (v == null) {
            v = e.eval(s);
        }
        // otherwise, return the pre-computed value
        return v;
    }
}
```

Modified App.java:



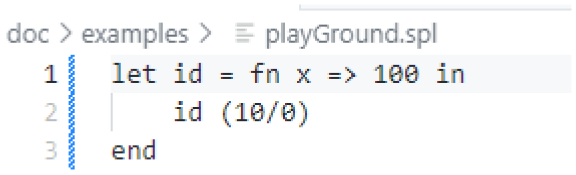
```java
@Override
public Value eval(State s) throws RuntimeError {

    // evaluate left hand side and check if it is a function value
    var lhsVal = l.eval(s);
    if (!(lhsVal instanceof FunValue)) {
        throw new RuntimeError("Runtime error: " + l.toString() + "can not be evaluated to a function value");
    }
    // type cast to func value
    var fnVal = (FunValue) lhsVal;

    if (Features.LAZY) {
        var thunk = new ThunkValue(s, r);
        return fnVal.e.eval(State.of(Env.of(fnVal.E, fnVal.x, thunk), s.M, s.p));
    } else {
        var argVal = r.eval(s);
        return fnVal.e.eval(State.of(Env.of(fnVal.E, fnVal.x, argVal), s.M, s.p));
    }
}
```

Modified Let.java:



```java
@Override
public Value eval(State s) throws RuntimeError {

    // if lazy evaluation is on
    if (Features.LAZY) {
        var thunk = new ThunkValue(s, e1);
        return e2.eval(State.of(Env.of(s.E, x, thunk), s.M, s.p));
    } else {
        var v1 = e1.eval(s);
        return e2.eval(State.of(Env.of(s.E, x, v1), s.M, s.p));
    }
}
```

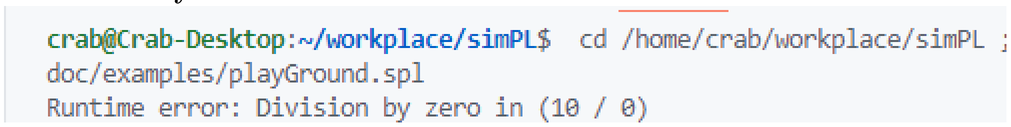
With these modifications applied, the program is set for testing using a simple example. Without lazy evaluation, the parameter expression (10/0) would be evaluated before being passed into the "id" function. This would lead to a division by 0 error, causing the evaluation process to halt. However, with correct implementation of lazy evaluation, the program output will be 100. This is because the parameter never appears in the body of "id," and as a result, it is never evaluated.



doc > examples > playGround.spl

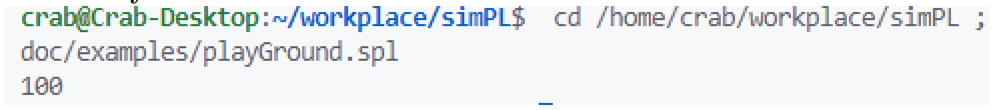
```
let id = fn x => 100 in
    id (10/0)
end
```

Without Lazy Evaluation



```
crab@Crab-Desktop:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
doc/examples/playGround.spl
Runtime error: Division by zero in (10 / 0)
```

With Lazy Evaluation



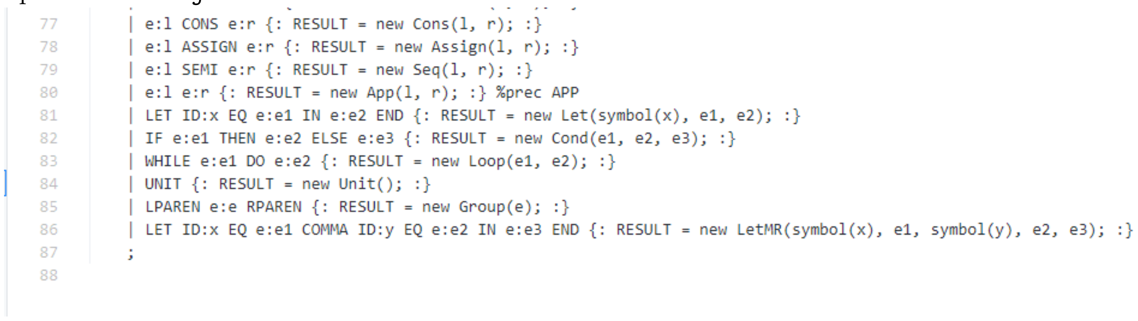
```
crab@Crab-Desktop:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
doc/examples/playGround.spl
100
```

3.5.4 Mutual Recursion

As introduced in the Ocaml lecture, mutual recursion enables programmers to two functions definition that can mutually drive each other downward. This is in contrast to relying on "rec" to drive itself. In the context of the simPl interpreter, while technically it is possible to use nested Let expressions to implement mutual recursion, type checking can easily manipulated by assigning a type variable to a "not yet existed" function. Evaluation, on the other hand, is more complicated. With lazy evaluation enabled, the first mutually defined function fixes its environment concretely at the time of evaluation. Unfortunately, at this point, we don't have the value of the other function in the current environment yet.

Therefore, I decided to create explicit syntax for mutual recursion: `Let x = e1, y = e2 in e3`. In order to achieve this, several things need to be changed:

1. Modification in the program syntax. Since no new keywords are introduced, the only place that needs to be changed is `simpl.grm`. After making necessary changes, I run the make file again to update `Parser.java`.



```
    | e:l CONS e:r {: RESULT = new Cons(l, r); :}
    | e:l ASSIGN e:r {: RESULT = new Assign(l, r); :}
    | e:l SEMI e:r {: RESULT = new Seq(l, r); :}
    | e:l e:r {: RESULT = new App(l, r); :} %prec APP
    | LET ID:x EQ e:e1 IN e:e2 END {: RESULT = new Let(symbol(x), e1, e2); :}
    | IF e:e1 THEN e:e2 ELSE e:e3 {: RESULT = new Cond(e1, e2, e3); :}
    | WHILE e:e1 DO e:e2 {: RESULT = new Loop(e1, e2); :}
    | UNIT {: RESULT = new Unit(); :}
    | LPAREN e:e RPAREN {: RESULT = new Group(e); :}
    | LET ID:x EQ e:e1 COMMA ID:y EQ e:e2 IN e:e3 END {: RESULT = new LetMR(symbol(x), e1, symbol(y), e2, e3); :}
    ;
```

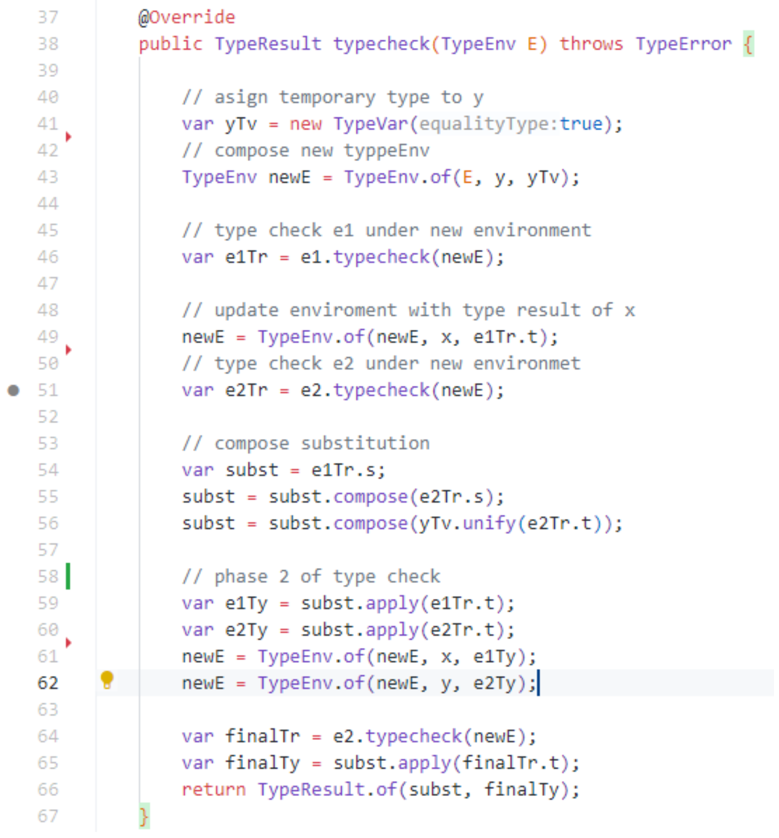
2. Since I defined a new node in the abstract syntax tree (AST), a new class related to the new node needs to be created in the `AST` folder, which in my case, LetMR.java.

After these changes, the remaining task is to figure out how to implement type checking and evaluation for `LetMR`. As mentioned earlier, type checking is straightforward with a type variable as a placeholder. After the first round of type checking, run type check a second time to obtain a unified type. Evaluation is a bit trickier, as it is easier to evaluate `e3` in an extended environment when `x` is bound to `v1` (the evaluated value of `e1`) and `y` is bound to `v2`. However, we eventually have to consult `v1`, whose body consists of `y`, and its environment doesn't have `y` yet.

To achieve this, I change the value of the `Environment` field in `FunValue` to be assignable, allowing us to update FunValue environment easily. With that done, the principle of evaluation for `LetMR` is as follows:

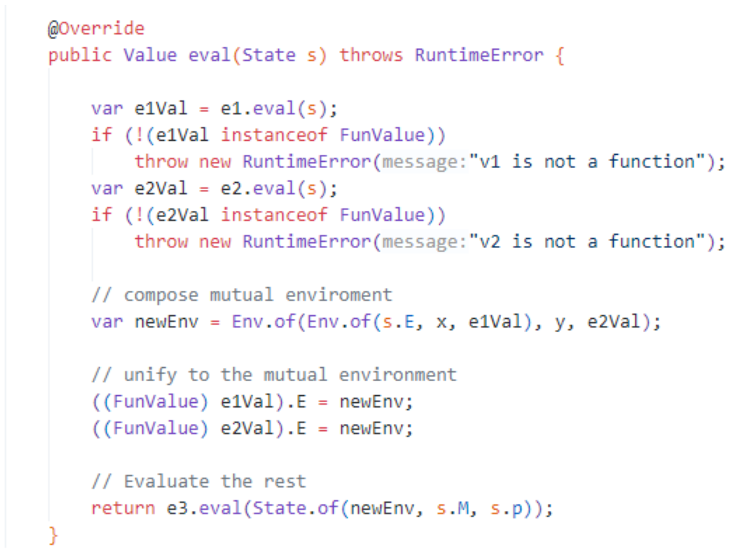
1. Evaluate `e1` to a value, presumably a function value. If not, throw a runtime error.
2. Evaluate `e2` to a value, presumably a function value. If not, throw a runtime error.
3. Enlarge the current environment by binding `x` to the value of `e1` and `y` to the value of `e2`.
4. Update the environments of `v1` and `v2` to the new enlarged environment so that they both contain the definition of the other.
5. Evaluate `e3` in the new enlarged environment.

Type check



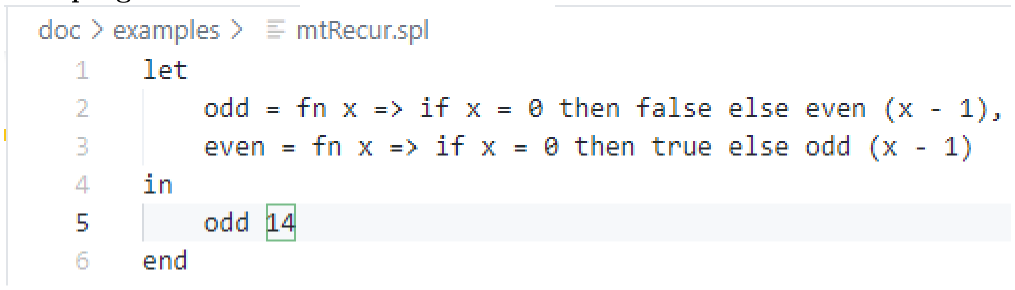
```
    @Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {

        // asign temporary type to y
        var yTv = new TypeVar(equalityType:true);
        // compose new typpeEnv
        TypeEnv newE = TypeEnv.of(E, y, yTv);

        // type check e1 under new environment
        var e1Tr = e1.typecheck(newE);

        // update enviroment with type result of x
        newE = TypeEnv.of(newE, x, e1Tr.t);
        // type check e2 under new environmet
        var e2Tr = e2.typecheck(newE);

        // compose substitution
        var subst = e1Tr.s;
        subst = subst.compose(e2Tr.s);
        subst = subst.compose(yTv.unify(e2Tr.t));

        // phase 2 of type check
        var e1Ty = subst.apply(e1Tr.t);
        var e2Ty = subst.apply(e2Tr.t);
        newE = TypeEnv.of(newE, x, e1Ty);
        newE = TypeEnv.of(newE, y, e2Ty);

        var finalTr = e2.typecheck(newE);
        var finalTy = subst.apply(finalTr.t);
        return TypeResult.of(subst, finalTy);
    }
```

Eval



```java
@Override
public Value eval(State s) throws RuntimeError {

    var e1Val = e1.eval(s);
    if (!(e1Val instanceof FunValue))
        throw new RuntimeError(message:"v1 is not a function");
    var e2Val = e2.eval(s);
    if (!(e2Val instanceof FunValue))
        throw new RuntimeError(message:"v2 is not a function");

    // compose mutual enviroment
    var newEnv = Env.of(Env.of(s.E, x, e1Val), y, e2Val);

    // unify to the mutual environment
    ((FunValue) e1Val).E = newEnv;
    ((FunValue) e2Val).E = newEnv;

    // Evaluate the rest
    return e3.eval(State.of(newEnv, s.M, s.p));
}
```

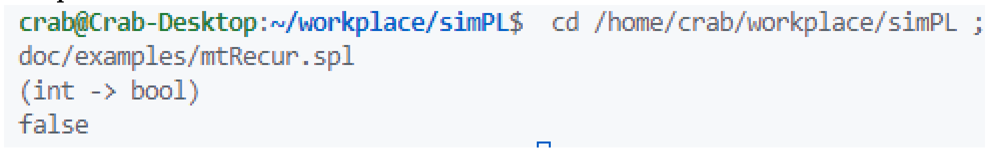
Test program



doc > examples > mtRecur.spl

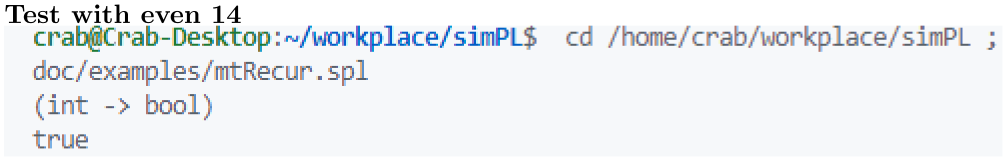
```
let
    odd = fn x => if x = 0 then false else even (x - 1),
    even = fn x => if x = 0 then true else odd (x - 1)
in
    odd 14
end
```

Output



```
crab@Crab-Desktop:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
doc/examples/mtRecur.spl
(int -> bool)
false
```

Test with even 14



```
crab@Crab-Desktop:~/workplace/simPL$  cd /home/crab/workplace/simPL ;
doc/examples/mtRecur.spl
(int -> bool)
true
```