SHANGHAI JIAO TONG UNIVERSITY

PROGRAMMING LANGUAGES (CS383)

COURSE PROJECT REPORT

---

# An Interpreter for SimPL

---


*Instructor:*
Kenny Q. Zhu

*T.A.:*
Xusheng Luo

*Author:*
Weiming Bao

*Student ID:*
5140219191


December 21, 2016

# Contents

# 1 Overview

## 1.1 Introduction

This is a report for the course project of *Programming Languages (CS383)*.

As specified, the target of this project is to implement an interpreter for SimPL, a simplified variant of the programming language ML. A well-designed skeleton on Java is provided along with the detailed specifications of SimPL.

Basic requirements, as illustrated in Fig. 1 mainly include the implementation of the typing and evaluation system. Additional requirements include the implementation of garbage collection, polymorphic types, lazy evaluation, etc.

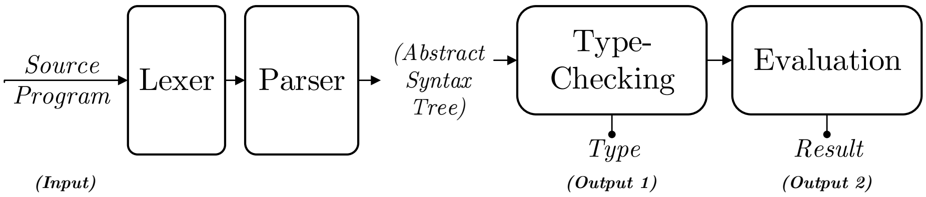


Figure 1: Flowchart illustrating the basic processes of a SimPL interpreter

## 1.2 Achievements

I have met the basic requirements for a SimPL interpreter, which performs correctly on all test programs provided in the skeleton package.

I have also implemented a *Mark-and-Sweep* garbage collector, polymorphic types, and the evaluation strategy of *Call-by-Need (lazy evaluation)*.

The implementation of the SimPL interpreter is a challenging task, especially the part regarding the typing system. Thanks to the project, I have gained a deeper understanding on concepts of programming languages, more specifically, type inference, $\lambda$ calculus, references, garbage collection, evaluation strategies and so on.

## 1.3 Report Organization

The rest of the report is organized as follows. The implementation approach is presented in Section 2.1. The implementation of the typing system and the evaluation system are elaborated respectively in Section 3 and Section 4. The additional parts (bonuses) are discussed in Section 5.

# 2 Implementation Approach

Lexical analysis and syntactical analysis (Lexer and Parser - as illustrated in Fig. 1) have already been implemented in the project skeleton utilizing JFlex and JavaCUP.

Method `parser()` serves as the entrance of lexical and syntactical analysis. Failure in these processes will report `Syntax Error`.

In this section, I will first present a brief introduction to the implicit implementation approach in the provided skeleton.

## 2.1 Abstract Syntax Tree (AST)

After the lexical and syntactical analysis, an *Abstract Syntax Tree* (AST) is generated for the lexically and syntactically correct source program with the root node `Expr`.

`Expr` is maintained as an abstract class, the class diagram of which is illustrated in Fig. 2. All the different expressions are extended from this class. Binary and unary operations are packaged respectively in `BinaryExpr` and `UnaryExpr` respectively.

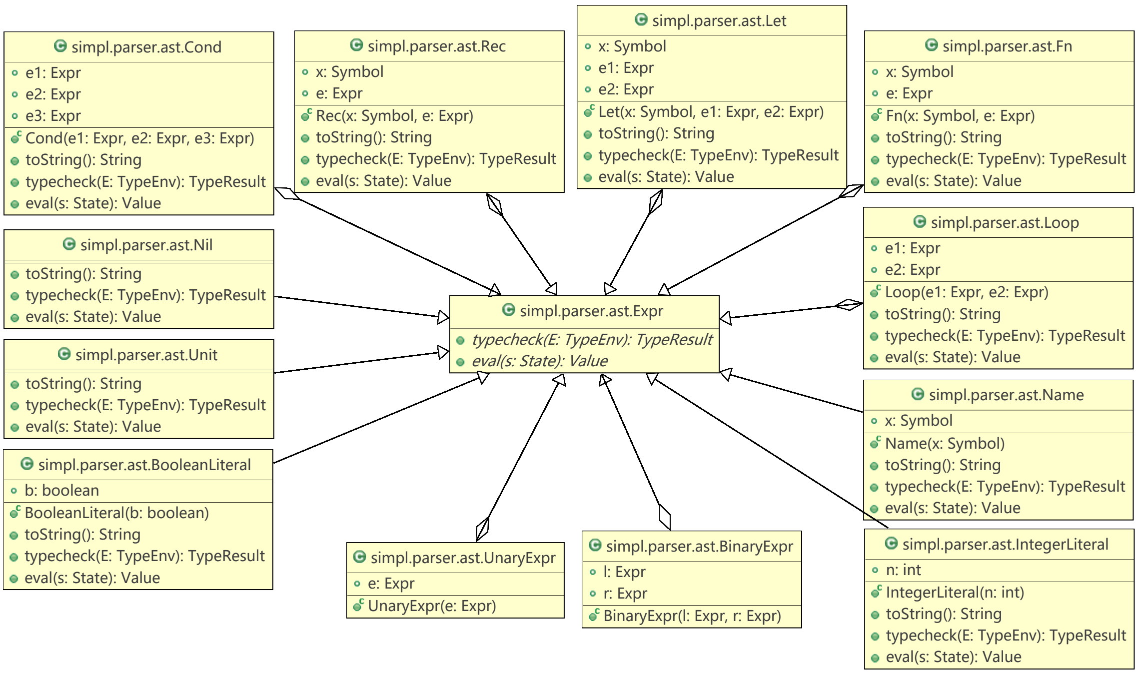


Figure 2: Java Class Diagram of `Expr` in the implementation

Binary operation expressions include `ArithExpr`, `Cons`, `Pair`, `AndAlso`, `OrElse`, `Seq`, `App` and `RelExpr`.

Unary operation expressions include `Group`, `Not`, `Ref`, `Deref` and `Neg`.

## 2.2 Java Methods

In the classes corresponding to AST "nodes", method `typecheck()` and `eval()` serves receptively as the interfaces for type checking and evaluation, which are the main challenges in the basic implementation.

# 3 Typing

## 3.1 Introduction

The type checking system is mainly relying on Type Inference. The main purpose is to type-check the nested expressions and generate the type of the untyped source program if possible.

From a general perspective, the main steps include adding the typing scheme, generate constraints, solving or simplifying constraints and applying the solution.

More specifically, first, walk over the program keep track of the type equations t1 = t2 that must hold in order to type check the expressions according to the normal typing rules and introduce new type variables for unknown types whenever necessary. Then, the set of constraints are solved utilizing the unification algorithm (involving substitution and composition). And the principal solutions are applied to the nested expressions.

## 3.2 Implementation

### 3.2.1 Classes for Type Scheme

The Java class diagram of the type scheme is illustrated in Fig. 3. Extended from the abstract class `Type`, there are known types `BoolType`, `ArrowType`, `PairType`, `ListType`, `RefType`, `UnitType`, `IntType`, and type variables `TypeVar`.

The following methods are overridden from the abstract class:

1. `isEqualityType()` - returns a boolean value.
2. `replace()` - replaces a TypeVar to a known Type and returns the known type.
3. `contains()` - returns a boolean value.
4. `unify()` - returns a substitution given a known type.

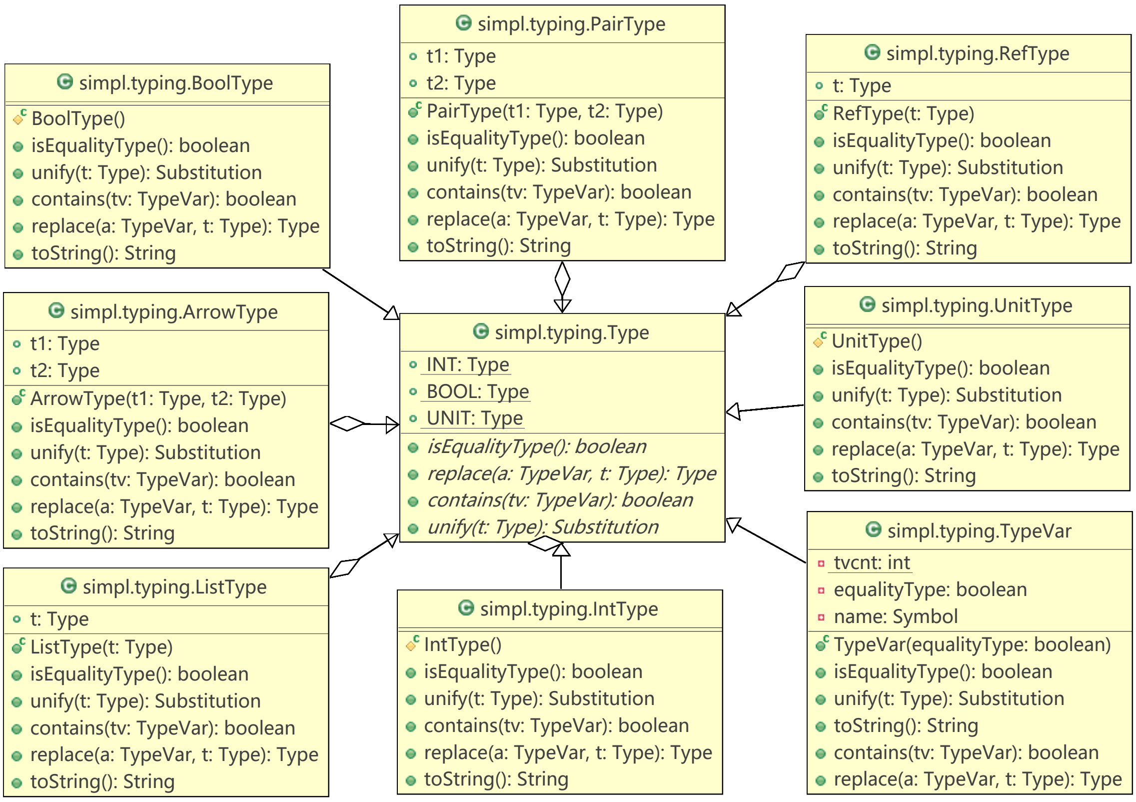


Figure 3: Java Class Diagram of the types in the implementation

### 3.2.2 Type-checking Entrance

Type-checking begins with the `Expr` node with the default type environment, involving two classes `TypeEnv` and `DefaultTypeEnv`, the java class diagram of which is illustrated in Fig. 4.

The `TypeEnv` is an abstract class. The class `DefaultTypeEnv` extends `TypeEnv` containing the initialized type environment with predefined names `fst()`, `snd()`, `hd()`, `tl()`, `iszero()`, `pred()` and `succ()`, as specified in the Specfication document.

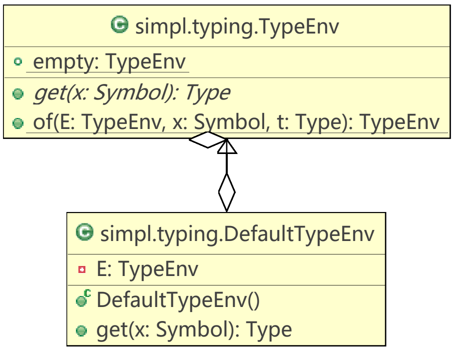


Figure 4: Java Class Diagram of the Environment types in the implementation

The constructor of `DefaultTypeEnv` is shown as below.

```java
public DefaultTypeEnv() {
    TypeVar tv1 = new TypeVar(false);
    TypeVar tv2 = new TypeVar(false);
    E = TypeEnv.of(
            TypeEnv.of(
                TypeEnv.of(
                    TypeEnv.of(
                        TypeEnv.of(
                            TypeEnv.of(
                                TypeEnv.of(TypeEnv.empty, Symbol.symbol
                                    ("fst"), new ArrowType(new PairType
                                    (tv1, tv2), tv1)),
                            Symbol.symbol("snd"), new ArrowType(new
                                PairType(tv1, tv2), tv2)),
                        Symbol.symbol("hd"), new ArrowType(new ListType
                            (tv1), tv1)),
                    Symbol.symbol("tl"), new ArrowType(new ListType(tv1
                        ), new ListType(tv1))),
                Symbol.symbol("iszero"), new ArrowType(new IntType(),
                    new BoolType())),
            Symbol.symbol("pred"), new ArrowType(new IntType(), new
                IntType())),
        Symbol.symbol("succ"), new ArrowType(new IntType(),new IntType
            ())
    );
}
```

The constraints are generated automatically under the recursive type-checking process with the help of the AST nodes.

### 3.2.3 Solving Constraints

A substitution implemented in class `substitution()` is a method / function from type variables TypeVar to type schemes Type. The corresponding Java class diagram is in Fig. 5.

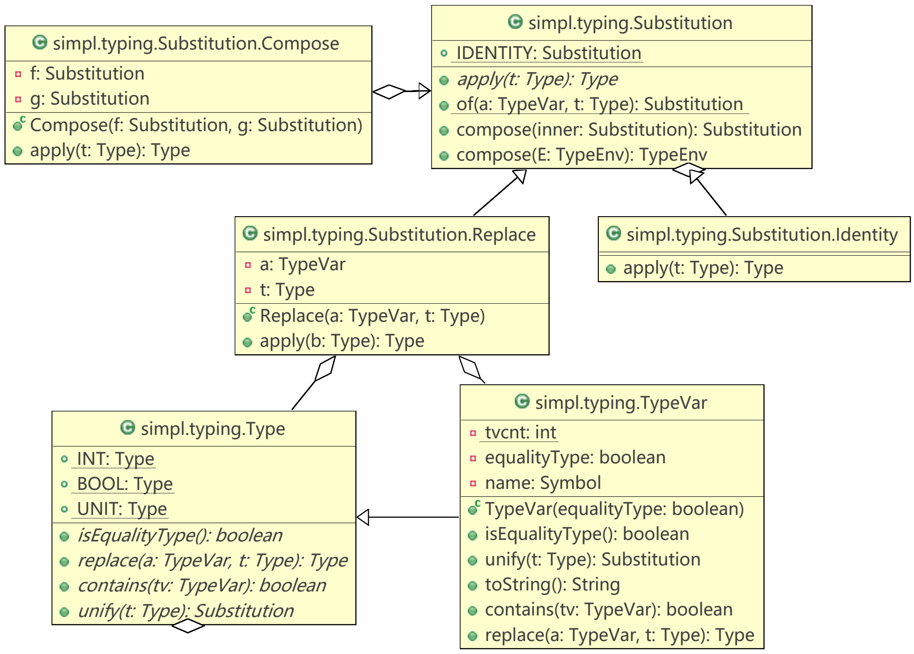


Figure 5: Java Class Diagram of Substitution in the implementation of typing system

The composition of two substitutions is specified in subclass `Compose`. The order of composed substitutions is illustrated as

$$(U \circ S)(a) = U(S(a)),$$

where the substitution S is first applied and then comes the substitution U.

Unification systematically simplifies a set of constraints, yielding a substitution to get the principal solution. In this project, unification is implemented in method `unify()` for every type class mentioned in Section 3.2.1.

For unknown types, the key method is `unify()` in class `TypeVar`, the implementation of which is as follows:

```java
public Substitution unify(Type t) throws TypeCircularityError {
    if (t.contains(this)) {
        if (t instanceof TypeVar && ((TypeVar)t).name.equals(this.name)
        )
```

```
            // a = a;
            return Substitution.IDENTITY;
        else
            // Circularity (EG: a = b->a);
            throw new TypeCircularityError();
    }
    return Substitution.of(this,t);
}
```

For `BoolType`, `IntType` and `UnitType`, the implementation of the method `unify()` is as follows:

```
public Substitution unify(Type t) throws TypeError {
    if(t instanceof TypeVar){
        return t.unify(this);
    }
    if(t instanceof BoolType){
        return Substitution.IDENTITY;
    }
    throw new TypeMismatchError();
}
```

Similar to `ArrowType`, the implementation of `unify()` in `PairType` is as follows:

```
public Substitution unify(Type t) throws TypeError {
    if (t instanceof TypeVar) {
        return t.unify(this);
    }
    if (t instanceof ArrowType) {
        Substitution s1 = ((ArrowType)t).t2.unify(this.t2);
        Substitution s2 = s1.apply(((ArrowType)t).t1).unify(s1.apply(
            this.t1));
        return s2.compose(s1);
    }
    throw new TypeMismatchError();
}
```

Similar to `ListType`, the implementation of `unify()` in `RefType` is as follows:

```
public Substitution unify(Type t) throws TypeError {
    if(t instanceof TypeVar){
        return t.unify(this);
    }
    if(t instanceof RefType){
        return this.t.unify(((RefType)t).t);
    }
    throw new TypeMismatchError();
}
```

### 3.2.4 Type-Checking Result

If the solving process failed, `Type Error` is reported, where there are two specific kinds of type errors, as illustrated in Fig. 6.

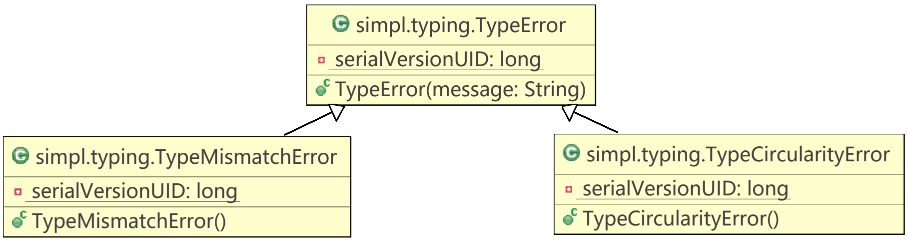


Figure 6: Java Class Diagram of the type errors in the implementation

If there is no type error, the solving process terminated properly in a known type class instance. `System.out.println()` will automatically call the corresponding overridden `toString()` method and print the type.

### 3.2.5 Selected Implementations of `typecheck()`

The naming policy on temporal variables in `typecheck()` is to compose the lower-case initial letters of the words. For instance, `ltr` indicates the left type result and `rt` indicates the right type.

The implementation of `typecheck()` in `ArithExpr`:

```java
public TypeResult typecheck(TypeEnv E) throws TypeError {
    // Get the left type result ltr and the right type result rtr
    TypeResult ltr = l.typecheck(E);
    TypeResult rtr = r.typecheck(E);
    // Compose and add the substitution to TypeEnv
    Substitution substitution = rtr.s.compose(ltr.s);
    // Generate the corresponding type
    Type lt = substitution.apply(ltr.t);
    Type rt = substitution.apply(rtr.t);
    // Unify lt and rt with Type.INT
    substitution = rt.unify(Type.INT).compose(lt.unify(Type.INT)).
        compose(substitution);
    // return the type result
    return TypeResult.of(substitution,Type.INT);
}
```

- First, the left type and the right type in the current type environment are checked. If correct, the type results will be saved in `ltr` and `rtr` respectively.

- The substitution is composed and added to `TypeEnv E` in the specified order. `substitution` gets the returned composed substitutions.
- Since `ArithExpr` applies to two int type, we then get the corresponding `Type` instance `lt` and `rt` and unify `lt` and `rt` with `Type.Int` in the previous order.
- Finally, `TypeResult` is returned with the new substitutions and `Type.Int`.

The implementation of `typecheck()` in `TypeVar`:

```java
public Substitution unify(Type t) throws TypeCircularityError {
    if (t.contains(this)) {
        if (t instanceof TypeVar && ((TypeVar)t).name.equals(this.name)
            )
            return Substitution.IDENTITY; // a = a;
        else
            throw new TypeCircularityError(); // Circularity (EG: a = b
                ->a);
    }
    return Substitution.of(this,t);
}
```

The circularity error during the typing process is detected here, when a constraint like `a = b->a` occurs. While `a = a` is the most intuitively case. The implementation of this part has cost me a lot of time.

Several other typical implementations of `typecheck()` are listed below:

The implementation of `typecheck()` in `Cond`:

```java
public TypeResult typecheck(TypeEnv E) throws TypeError {
    //          TODO
    TypeResult tr1 = e1.typecheck(E);
    TypeResult tr2 = e2.typecheck(E);
    TypeResult tr3 = e3.typecheck(E);
    Substitution substitution = tr3.s.compose(tr2.s).compose(tr1.s);
    Type t1 = substitution.apply(tr1.t);
    Type t2 = substitution.apply(tr2.t);
    Type t3 = substitution.apply(tr3.t);
    substitution = t1.unify(Type.BOOL).compose(substitution);
    t2 = substitution.apply(t2);
    t3 = substitution.apply(t3);
    TypeVar rt = new TypeVar(false);
    substitution = t2.unify(rt).compose(substitution);
    t3 = substitution.apply(t3);
    substitution = t3.unify(rt).compose(substitution);
    return TypeResult.of(substitution,substitution.apply(rt));
}
```

The implementation of `typecheck()` in `EqExpr`:

```
public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeResult ltr = l.typecheck(E);
    TypeResult rtr = r.typecheck(E);
    Substitution substitution = rtr.s.compose(ltr.s);
    Type lt = substitution.apply(ltr.t);
    Type rt = substitution.apply(rtr.t);

    if(lt instanceof ListType || rt instanceof ListType){
        Type tv = new TypeVar(false);
        substitution = lt.unify(new ListType(tv)).compose(substitution)
            ;
        tv = substitution.apply(tv);
        substitution = rt.unify(new ListType(tv)).compose(substitution)
            ;
    }else if(lt instanceof PairType || rt instanceof PairType){
        Type ltv = new TypeVar(false);
        Type rtv = new TypeVar(false);
        substitution = lt.unify(new PairType(ltv,rtv)).compose(
            substitution);
        ltv = substitution.apply(ltv);
        rtv = substitution.apply(rtv);
        substitution = rt.unify(new PairType(ltv,rtv)).compose(
            substitution);
    }else if(lt instanceof RefType || rt instanceof RefType){
        Type tv = new TypeVar(false);
        substitution = lt.unify(new RefType(tv)).compose(substitution);
        tv = substitution.apply(tv);
        substitution = rt.unify(new RefType(tv)).compose(substitution);
    }
    else if(lt instanceof TypeVar && rt instanceof TypeVar){
        Type tv = new TypeVar(false);
        substitution = lt.unify(tv).compose(substitution);
        tv = substitution.apply(tv);
        substitution = rt.unify(tv).compose(substitution);
    } else if(lt.equals(Type.INT) || rt.equals(Type.INT)){
        substitution = rt.unify(Type.INT).compose(lt.unify(Type.INT)).
            compose(substitution);
    }else if(lt.equals(Type.BOOL) || rt.equals(Type.BOOL)){
        substitution = rt.unify(Type.BOOL).compose(lt.unify(Type.BOOL))
            .compose(substitution);
    }
    return TypeResult.of(substitution,Type.BOOL);
}
```

# 4 Evaluation

## 4.1 Introduction

Intuitively, SIMPL programs are nested expressions. After the type-checking process, the evaluation system deals with the nested expressions recursively.

If evaluation failed, (e.g. divided by 0), `Runtime Error` is reported. Otherwise, the final value instance is returned.

Given the possible nested expressions, a call stack need to be maintained for set of bindings (environment `Env`) in form of `state` for `Ref` cells.

## 4.2 Implementation

### 4.2.1 Classes for Values

The Java class diagram of the Values is illustrated in Fig. 7. Extended from the abstract class `Value`, there are known types `BoolValue`, `FunValue`, `ConsValue`, `PairValue`, `RefValue`, `UnitValue`, `IntValue`, `NilValue`, and `RecValue`.

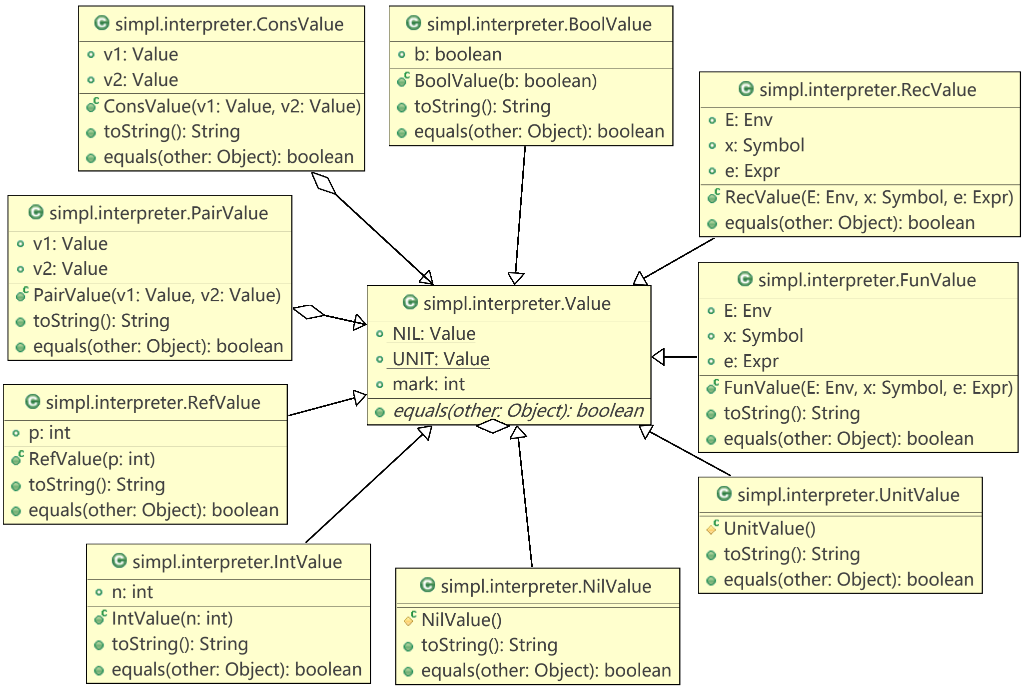


Figure 7: Java Class Diagram of the Values in the implementation

The method `equals()` that returns a boolean value is overridden from the abstract class, which is intuitively simple to implement.

### 4.2.2 Evaluation Entrance

Evaluation begins with the `Expr` node with the initial state, involving two classes `InitialState` and `State`. The related java class diagram is illustrated in Fig. 8.

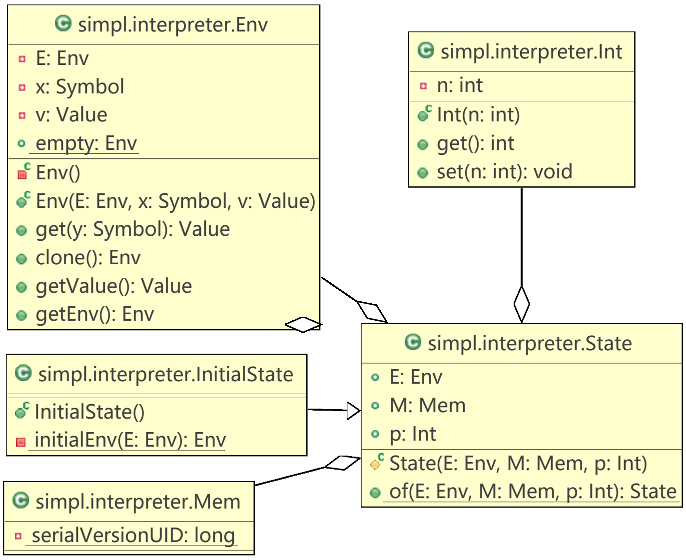


Figure 8: Java Class Diagram of the states in the implementation

The initialization of `InitialState` is shown as below.

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

The constructor of the predefined names are called during the initialization.

Take `hd` as an example, the constructor is implemented as follows, with the evaluation rules specified in the Specification document.

```
public hd() {
    super(Env.empty, Symbol.symbol("hd arg"), new Expr() {
        @Override
        public TypeResult typecheck(TypeEnv E) throws TypeError {
            return TypeResult.of(new TypeVar(true));
        }
        @Override
        public Value eval(State s) throws RuntimeError {
            Value v = s.E.get(Symbol.symbol("hd arg"));
            if (v == Value.NIL) {
                // hd(nil) = error;
                throw new RuntimeError("ERROR: hd(nil)");
            }else
            // hd(cons,v1,v2) = v1;
            return ((ConsValue)v).v1;
        }
    });
}
```

#### 4.2.3 Environment `Env`

An environment is a set of bindings containing (variable, value) mappings. Class `Env` is nested for implementing nested function calls. During evaluation process, each time a symbol comes, we will check whether it is already in the current `Env`. If not, we will create a new `Env` instance of the current `Env` to add the bindings.

Take a function with multiple parameters as an example. The implementation of `eval()` in `App` are as follows:

```
public Value eval(State s) throws RuntimeError {
    // E-App
    FunValue fv = (FunValue)l.eval(s);
    Value v = r.eval(s);
    return fv.e.eval(State.of(new Env(fv.E, fv.x, v),s.M,s.p));
}
```

After the evaluation, a new state is provided with a new Env where all "`x`" is binding to `v`. With the help of `Env`, when the evaluation of the first parameter is finished, the second parameters are then to be added to the `Env`, which is, to be evaluated.

This is exactly the *Call-by-Need* evaluation strategy (Lazy Evaluation), which will be discussed further in Section 5.3

The implementation of `Env` class is as follows:

```
public Value get(Symbol y) {
    if (y.toString().equals(x.toString()))
        return v;
    else
        return E.get(y);
}
```

### 4.2.4 Memory Reference `Ref`

`Ref` is the only instruction related to memory (heap) assignments. During the evaluation process, each time `Ref` comes, the Value is put into the heap and a corresponding pointer is returned.

The pointer `p` and the memory `Mem` along with the corrent `Env` are maintained by class `State`. A call stack can be implemented just by creating new states with modified member variable `p`.

The evaluation method `eval()` of `Ref` is implemented as follows:

```
public Value eval(State s) throws RuntimeError {
    // E-Ref
    // Obtain the current p
    Int p = new Int(s.p.get());
    // Increase the block index by 1
    s.p.set(s.p.get()+1);
    // Evaluate the Value under the new s
    Value v = e.eval(s);
    // Put the Value into the corresponding block
    s.M.put(p.get(), v);
    // Return the new RefValue / pointer
    return new RefValue(p.get());
}
```

- First, obtain the current `p` and increase the block index by 1.
- Evaluate the `Value` and put it into the corresponding block.
- Finally, the new `RefValue` / pointer is returned.

### 4.2.5 Selected Implementations of `eval()`

The naming policy on temporal variables in `eval()` is to compose the lower-case initial letters. For instance, `lv` indicates the left value.

Similar to other arithmetic expressions, the implementation of `eval()` in `Add`:

```java
public Value eval(State s) throws RuntimeError {
    // E-Add
    IntValue v1 = (IntValue)l.eval(s);
    IntValue v2 = (IntValue)r.eval(s);
    return new IntValue(v1.n + v2.n);
}
```

Several other typical implementations of `eval()` are listed below:

The codes of `eval()` in `Assign` are as follows:

```java
public Value eval(State s) throws RuntimeError {
    // E-Assign
    // Evaluate the left value and right value
    RefValue lv = (RefValue)l.eval(s);
    Value rv = r.eval(s);
    // Put the value into corresponding block
    s.M.put(lv.p, rv);
    return Value.UNIT;
}
```

The codes of `eval()` in `Eq` are as follows:

```java
public Value eval(State s) throws RuntimeError {
    // E-Eq1 & E-Eq2
    Value lv = l.eval(s);
    Value rv = r.eval(s);
    return lv.equals(rv) ? new BoolValue(true) : new BoolValue(false);
}
```

The codes of `eval()` in `Not` are as follows:

```java
public Value eval(State s) throws RuntimeError {
    // E-Not1 & E-Not2
    BoolValue v = (BoolValue)e.eval(s);
    return new BoolValue(!(v.b));
}
```

# 5 Bonus

## 5.1 Garbage Collection

In modern Program Languages Garbage Collection is a vital task.

A garbage is a block of heap memory that cannot be accessed by the program. There are two types of garbages: *orphans* and *widows*.

Although SimPL is simple, there exists the possibility for *orphans* to be created, due to assignments and memory references.

I have implemented a *Mark and Sweep* garbage collector in the project. Basically, the garbage collector is triggered when the heap is full. All the referenced cells are marked and the unmarked *orphans* are swept.

### 5.1.1 Implementation

Given that garbage can only be created during the evaluation of `Ref`. The implementation of Mark and Sweep garbage collector is illustrated as follows:

```
public Value eval(State s) throws RuntimeError {
    Int p = new Int(s.p.get());

    // ============== GC: Mark and Sweep ==============
    int HEAPSIZE = 1; // heap size
    if(p.get() > HEAPSIZE)
    {
        // Mark
        Env env = s.E;
        while(env != Env.empty)
        {
            Value val = env.getValue();
            while(val instanceof RefValue && val.mark == 0)
            {
                val.mark = 1;
                val = s.M.get(((RefValue)val).p);
            }
            val.mark = 1;
            env = env.getEnv();
        }
        System.out.println("--- Before: ---");
        for (int i = 0; i < p.get(); i++)
        {
            System.out.println(s.M.get(i));
```

```
        }
        // Sweep
        for (int i = 0; i < p.get(); i++)
        {
            if (s.M.get(i) != null && s.M.get(i).mark == 0)
            {
                s.M.put(i, null); // Utilizing the nature of Java
                    HashMap
                System.out.println("collect" + i);
            }
        }
        System.out.println("--- After: ---");
        for (int i = 0; i < p.get(); i++)
        {
            System.out.println(s.M.get(i));
        }
        System.out.println("GC completed");
    }
    // ============================= END =============================
    s.p.set(s.p.get()+1);
    Value v = e.eval(s);
    s.M.put(p.get(), v);
    return new RefValue(p.get());
}
```

A member variable indicating the reference status is added into `Value`:

```
public int mark = 0;
```

Two relevant member methods are added into `Env`:

```
public Value getValue()
{
    return v;
}
public Env getEnv()
{
    return E;
}
```

The memory is implemented by a Java HashMap in the provided skeleton. During the sweep operations, we only need to put `null` into the corresponding garbage cell.

In my implementation, I restricted the heap size to be 2, which is an extreme condition only for illustration on the following testcase.

### 5.1.2 Testcase

Take a sample SimPL program as the example, where I have met great obstacles to find it:

```
let f = fn x => ref 1 in
    let y = ref 2 in
        !(f 1) + !y + !(f 1)
    end
end
```

Analyzing the above test SimPL program, during the evaluation of two let expression, value 1 and value 2 are put into the heap before the evaluation of its right expression.

While during the evaluation of its right expression `!(f 1) + !y + !(f 1)`, orphans are generated right before the evaluation of the second `!(f 1)` in *Call-by-Need* evaluation strategy.

Set the heap size to be 2 to trigger the Mark and Sweep garbage collector. Then the output of the sample program will be as follows:

```
int
--- Before: ---
2
1
collect0
collect1
--- After: ---
null
null
GC completed
4
```

## 5.2 Polymorphic types

The implementation of polymorphic types has been accomplished, given the type system implements Type Inference.

When an unknown type comes, it is represented with `Type.IDENTIFY` and a name `tv + tvcount`. `tvcount` will increase every time an unknown type is encountered, as is illustrated in the implementation code for `TypeVar` below.

```
public TypeVar(boolean equalityType) {
    this.equalityType = equalityType;
    name = Symbol.symbol("tv" + ++tvcnt);
}
```

Take the following SimPL source program as the example:

```
(* using polymorphic types *)
rec map =>
    fn f => fn l =>
        if l=nil
        then nil
        else (f (hd l))::(map f (tl l))
```

The type-checking result is

`((tv33 -> tv34) -> (tv33 list -> tv34 list)).`

## 5.3 Lazy Evaluation

In Lazy Evaluation strategy, an expression will never be evaluated until its value is needed, which is also called *Call-by-Need* evaluation strategy.

The implementation of Lazy Evaluation is tightly related to the implementation of `Env`, regarding nested functions, as is elaborated in Section 4.2.3.

As is specified in the evaluation rules, a *Short-Circuit-Evaluation* like policy is applied to `AndAlso` and `OrElse`.

The codes of `eval()` in `AndAlso` are as follows:

```
public Value eval(State s) throws RuntimeError {
    BoolValue lv = (BoolValue)l.eval(s);
    if (!lv.b)
    // E-AndAlso2
        return new BoolValue(false);
    else {
        BoolValue rv = (BoolValue)r.eval(s);
        // E-AndAlso1
        return new BoolValue(rv.b);
    }
}
```

If the left value `lv` is `false`, the right value `rv` will not be evaluated.

The codes of `eval()` in `OrElse` are as follows:

```java
public Value eval(State s) throws RuntimeError {
    BoolValue lv = (BoolValue)l.eval(s);
    if (lv.b == true)
    // E-OrElse1
    return new BoolValue(true);
    else {
        // E-OrElse2
        BoolValue rv = (BoolValue)r.eval(s);
        return new BoolValue(rv.b);
    }
}
```

## 6 Acknowledgment

Despite the well-designed skeleton, the project is still really challenging. Due to my poor performance in the killing final exam, I devoted heart and soul to this project.

During the project, I reviewed the course slides and reference books and gained a deeper understanding of concepts related to *Programming Languages*. Thanks for my classmates for discussing confusing concepts.

Finally, I would like to express my gratitude for Prof. Kenny Q. Zhu and dear T.A. Xusheng Luo for providing a chance for me to overcome challenges and obtain fruitful knowledges.