X-Fiber: A Language with Exceptions, Functions, Integers, Booleans, Eagerness, and Recursion

CS320 Programming Languages Project #2 (Due: June 7, 2020)

1 INTRODUCTION

X-Fiber is a toy language for the second project of the CS320 course. X-Fiber stands for a language with e**x**ceptions, **f**unctions, **i**ntegers, **b**ooleans, **e**agerness, and **r**ecursion. As the name implies, it extends Fiber. Like Fiber, it is an eager language and features integers, booleans, first-class functions, and recursive functions. In addition, it provides first-class continuations, exceptions, and exception handlers. More precisely, X-Fiber supports the following features (the bold parts are the features not in Fiber):

- integers and booleans
- basic arithmetic operators, including negation, addition, subtraction, multiplication, division, and modulo
- basic relational operators, including equal-to, not-equal-to, less-then, less-then-or-equal-to, greater-then, and greater-then-or-equal-to.
- basic boolean operators, including negation, conjunction, and disjunction
- conditional expressions (if-else expressions)
- tuples of arbitrary lengths greater than one
- projections for tuples
- lists, which are cons or nil
- primitives for lists: isEmpty, nonEmpty, head, and tail
- immutable local variables
- immutable local variable binding via pattern matching on tuples
- first-class functions and function application
- anonymous functions
- mutually recursive functions
- dynamic type tests
- **first-class continuations**
- **return expressions**
- **exceptions and exception handlers**

This document defines X-Fiber and provides a guide to the project. First, it gives the syntax of X-Fiber: Section 2 describes the concrete syntax; Secion 3 formalizes the desugaring rules; Section 4 shows the abstract syntax. Second, it describes the semantics of X-Fiber in Section 5. Finally, Section 6 explains the directory structure and rules for implementation. In addition, Appendix A shows the small-step semantics of X-Fiber.

2 CONCRETE SYNTAX

The concrete syntax of X-Fiber is written in the extended Backus–Naur form. To improve the readability, we use different colors for different kinds of objects. Syntactic elements of the extended Backus–Naur form, rather than X-Fiber, are written in purple. For example, we use `=`, `|`, and `;`. Note that `{ }` denotes a repetition of zero or more times, and `[ ]` denotes an optional existence. Nonterminals are written in blue. For example, `expr` is a nonterminal denoting expressions. Any other objects written in black are terminals. For instance, `"true"` and `"false"` are terminals representing boolean literals.

The parsing phase belongs to the given portion of the interpreter. Therefore, you do not need either to implement a parser or to deal with the concrete syntax directly. However, the concrete syntax helps you write your own test cases, and we recommend you to understand the concrete syntax briefly even though you are free to skip tedious details.

The following is the concrete syntax of X-Fiber (the parts in boxes are the cases not in Fiber.):

```
ltr  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L"
     | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
     | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
     | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v"
     | "w" | "x" | "y" | "z" ;
pdgt = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
dgt  = "0" | pdgt ;

sch  = ltr | "_" ;
ch   = sch | dgt ;
id   = sch {ch} ;

idx  = pdgt {dgt} ;
num  = ["-"] dgt {dgt} ;

expr = id | num | "true" | "false" | "-" expr | "!" expr
     | expr "+"  expr | expr "-"  expr | expr "*"  expr | expr "/"  expr
     | expr "%"  expr | expr "==" expr | expr "!=" expr | expr "<"  expr
     | expr "<=" expr | expr ">"  expr | expr ">=" expr | expr "&&" expr
     | expr "||" expr | "if" "(" expr ")" expr "else" expr
     | "(" expr "," expr {"," expr} ")" | expr "." "_" idx
     | "Nil" | expr "::" expr | expr "." "isEmpty"
     | expr "." "nonEmpty" | expr "." "head" | expr "." "tail"
     | "val" id "=" expr ";" expr
     | "val" "(" id "," id {"," id} ")" "=" expr ";" expr
     | "vcc" id ";" expr
     | "(" ")" "=>" expr | id "=>" expr | "(" id {"," id} ")" "=>" expr
     | fdef {fdef} expr
     | expr "(" ")" | expr "(" expr {"," expr} ")"
     | expr "." "isInstanceOf" "[" type "]"
     | "return" expr
     | "throw" expr
     | "try" expr "catch" expr
     | "(" expr ")"| "{" expr "}" ;

fdef = "def" id "(" ")" "=" expr ";"
     | "def" id "(" id {"," id} ")" "=" expr ";" ;

type = "Int" | "Boolean" | "Tuple" | "List" | "Function" ;
```

Note that whitespaces, such as `' '`, `'\t'`, and `'\n'`, are omitted from the above specification. You can insert any kinds of whitespaces between any two terminals to make a valid program. For

`*` `/` `%`	`+` `-`	`==` `!=` `<` `<=` `>` `>=`	`&&`	`||`	`::`

← higher lower

Fig. 1. Operator Precedence

$$
\begin{aligned}
\llbracket \texttt{-}e \rrbracket &= \llbracket e \rrbracket\ \texttt{* -1} \\
\llbracket \texttt{!}e \rrbracket &= \texttt{if (}\llbracket e \rrbracket\texttt{) false else true} \\
\llbracket e_1\ \texttt{-}\ e_2 \rrbracket &= \llbracket e_1 \rrbracket\ \texttt{+}\ \llbracket \texttt{-}e_2 \rrbracket \\
\llbracket e_1\ \texttt{!=}\ e_2 \rrbracket &= \llbracket \texttt{!(}\llbracket e_1 \rrbracket\ \texttt{==}\ \llbracket e_2 \rrbracket\texttt{)} \rrbracket \\
\llbracket e_1\ \texttt{<=}\ e_2 \rrbracket &= \texttt{val}\ \underline{x_1}\ \texttt{=}\ \llbracket e_1 \rrbracket\texttt{;} \\
&\ \texttt{val}\ \underline{x_2}\ \texttt{=}\ \llbracket e_2 \rrbracket\texttt{;} \\
&\ \llbracket x_1\ \texttt{==}\ x_2\ \texttt{||}\ x_1\ \texttt{<}\ x_2 \rrbracket \\
\llbracket e_1\ \texttt{>}\ e_2 \rrbracket &= \llbracket \texttt{!(}\llbracket e_1 \rrbracket\ \texttt{<=}\ \llbracket e_2 \rrbracket\texttt{)} \rrbracket \\
\llbracket e_1\ \texttt{>=}\ e_2 \rrbracket &= \llbracket \texttt{!(}\llbracket e_1 \rrbracket\ \texttt{<}\ \llbracket e_2 \rrbracket\texttt{)} \rrbracket \\
\llbracket e_1\ \texttt{\&\&}\ e_2 \rrbracket &= \texttt{if (}\llbracket e_1 \rrbracket\texttt{)}\ \llbracket e_2 \rrbracket\ \texttt{else false} \\
\llbracket e_1\ \texttt{||}\ e_2 \rrbracket &= \texttt{if (}\llbracket e_1 \rrbracket\texttt{) true else}\ \llbracket e_2 \rrbracket \\
\llbracket e\texttt{.nonEmpty} \rrbracket &= \llbracket \texttt{!(}\llbracket e \rrbracket\texttt{.isEmpty)} \rrbracket \\
&\boxed{\llbracket \texttt{return}\ e \rrbracket = \texttt{return(}\llbracket e \rrbracket\texttt{)}} \\
\llbracket \texttt{(}e\texttt{)} \rrbracket &= \llbracket e \rrbracket \\
\llbracket \texttt{\{}e\texttt{\}} \rrbracket &= \llbracket e \rrbracket
\end{aligned}
$$

$$
\boxed{\begin{aligned}
&\llbracket \texttt{(}x_1\texttt{,}\cdots\texttt{,}x_i\texttt{)}\ \texttt{=>}\ e \rrbracket = \\
&\quad \texttt{(}x_1\texttt{,}\cdots\texttt{,}x_i\texttt{)}\ \texttt{=>} \\
&\qquad \texttt{vcc return;}\ \llbracket e \rrbracket
\end{aligned}}
$$

$$
\boxed{\begin{aligned}
&\llbracket \texttt{def}\ x\texttt{(}x_1\texttt{,}\cdots\texttt{,}x_i\texttt{)}\ \texttt{=}\ e\texttt{;} \rrbracket = \\
&\quad \texttt{def}\ x\texttt{(}x_1\texttt{,}\cdots\texttt{,}x_i\texttt{)}\ \texttt{=} \\
&\qquad \texttt{vcc return;}\ \llbracket e \rrbracket\texttt{;}
\end{aligned}}
$$

$$
\begin{aligned}
&\llbracket \texttt{val (}x_1\texttt{,}\cdots\texttt{,}x_i\texttt{)}\ \texttt{=}\ e_1\texttt{;}\ e_2 \rrbracket = \\
&\quad \texttt{val}\ \underline{x}\ \texttt{=}\ \llbracket e_1 \rrbracket\texttt{;} \\
&\quad \texttt{val}\ x_1\ \texttt{=}\ x\texttt{.\_1;} \\
&\quad \cdots \\
&\quad \texttt{val}\ x_i\ \texttt{=}\ x\texttt{.\_}i\texttt{;} \\
&\quad \llbracket e_2 \rrbracket
\end{aligned}
$$

Any other cases recursively desugar their subexpressions.

Fig. 2. Desugaring Rules

example, since we have `expr = num | "-" expr`, if one parses `-1` and `- 1`, then both will succeed, and the results will be the same. On the other hand, because you cannot insert whitespaces at the middle of terminals, `tr ue` cannot be parsed while `true` can be parsed correctly.

Figure 1 shows operator precedence. One appearing earlier in the table precedes one appearing later. In X-Fiber, all the binary operators except `::` are left-associative. Only `::` is right-associative.

3 DESUGARING

To simplify the implementation of the interpreting phase, the parsing phase of the interpreter desugars a given expression. Desugaring rewrites some subexpressions with other expressions. Due to desugaring, the abstract syntax of X-Fiber consists of less sorts of expressions than the concrete syntax.

Like the concrete syntax, you do not need to care about desugaring in detail. However, understanding desugaring helps you find how test cases are transformed.

Figure 2 defines desugaring of X-Fiber expressions (the parts in boxes are the rules not in Fiber). Let e and x respectively denote an expression and an identifier. An expression e is desugared to $\llbracket e \rrbracket$.

$$
\begin{array}{rcll}
\text{Identifier} & x & \in & \mathit{Id} \\
\text{Index} & i & \in & \mathbb{Z}^{+} \\
\text{Number} & n & \in & \mathbb{Z} \\
\text{Boolean} & b & ::= & \mathsf{true} \\
& & | & \mathsf{false} \\
\text{Function} & d & ::= & \mathsf{def}\ x(x,\cdots,x) = e \\
\text{Types} & \tau & ::= & \mathsf{Int} \\
& & | & \mathsf{Boolean} \\
& & | & \mathsf{Tuple} \\
& & | & \mathsf{List} \\
& & | & \mathsf{Function}
\end{array}
$$

$$
\begin{array}{rcll}
\text{Expression} & e & ::= & x & \text{(variable)} \\
& & | & n & \text{(integer)} \\
& & | & b & \text{(boolean)} \\
& & | & e + e & \text{(addition)} \\
& & | & e \times e & \text{(multiplication)} \\
& & | & e \div e & \text{(division)} \\
& & | & e\ \mathsf{mod}\ e & \text{(modulo)} \\
& & | & e = e & \text{(equal-to)} \\
& & | & e < e & \text{(less-then)} \\
& & | & \mathsf{if}\ e\ e\ e & \text{(conditional)} \\
& & | & (e,\cdots,e) & \text{(tuple; length > 1)} \\
& & | & e.i & \text{(projection)} \\
& & | & \mathsf{Nil} & \text{(nil)} \\
& & | & \mathsf{Cons}\ e\ e & \text{(cons)} \\
& & | & e.\mathsf{isEmpty} & \text{(is-empty)} \\
& & | & e.\mathsf{head} & \text{(head)} \\
& & | & e.\mathsf{tail} & \text{(tail)} \\
& & | & \mathsf{val}\ x{=}e\ \mathsf{in}\ e & \text{(local variable)} \\
& & \boxed{|} & \boxed{\mathsf{vcc}\ x\ \mathsf{in}\ e} & \boxed{\text{(continuation)}} \\
& & | & \lambda x \cdots x.e & \text{(anonymous function)} \\
& & | & d \cdots d\ e & \text{(recursive function)} \\
& & | & e(e,\cdots,e) & \text{(function application)} \\
& & | & e\ \mathsf{is}\ \tau & \text{(type test)} \\
& & \boxed{|} & \boxed{\mathsf{throw}\ e} & \boxed{\text{(exception)}} \\
& & \boxed{|} & \boxed{\mathsf{try}\ e\ \mathsf{catch}\ e} & \boxed{\text{(exception handler)}}
\end{array}
$$

Fig. 3. Abstract Syntax

4 ABSTRACT SYNTAX

Figure 3 describes the abstract syntax of X-Fiber (the parts in boxes are the cases not in Fiber). Metavariable x ranges over identifiers; i ranges over indices of tuples, which are positive integers; n ranges over integers; b ranges over boolean literals, which are either true or false; e ranges over expressions; d ranges over recursive function definitions; τ ranges over types, which are either Int, Boolean, Tuple, List, or Function.

The following briefly describes expressions:

- $e_1 + e_2$, $e_1 \times e_2$, $e_1 \div e_2$, e_1 mod e_2, $e_1 = e_2$, and $e_1 < e_2$ are binary operations on integers.
- if e_1 e_2 e_3 is a conditional expression.
- $(e_1, \cdots, e_i)$ creates a tuple of length i. Length i must be greater than one.
- $e.i$ is a projection from a tuple. The beginning index is one.
- Nil creates the empty list.
- Cons e_1 e_2 creates a nonempty list.
- e.isEmpty, e.head, and e.tail are unary operations on a list.
- val x=e_1 in e_2 defines a local variable whose name is x and scope is e_2.
- vcc x in e defines a local variable whose name is x and scope is e. The current continuation is bound to x.

- $\lambda x_1 \cdots x_i.e$ defines an anonymous function whose parameters are $x_1, \cdots, x_i$ and body is e. The names of the parameters must be distinct from each other.
- def $x(x_1, \cdots, x_i)$ $=e$ defines a (possibly recursive) function whose name is x, parameters are $x_1, \cdots, x_i$, and body is e. The names of the parameters must be distinct from each other.
- $d_1 \cdots d_i$ e defines functions from d_1 to d_i. The names of the functions must be distinct from each other. They can be mutually recursive and used in e.
- $e(e_1, \cdots, e_i)$ is a function application. e is a function; $e_1, \cdots, e_i$ are arguments.
- e is τ tests the type of a given value.
- throw e throws an exception.
- try e_1 catch e_2 registers an exception handler.

5 SEMANTICS

This section explains the semantics of X-FIBER in a natural language. See Appendix A to find the formal small-step semantics.

To explain the semantics, we need the definition of a value. A value is one of the following:

- an integer
- a boolean
- a tuple whose length is greater than one and elements are values
- the empty list
- a nonempty list, which consists of a value and a (empty or nonempty) list
- a closure, which is a function with an environment
- a continuation, which denotes the remaining computation at some point of execution

In this section, we use the following metavariables and terminologies:

- Metavariable v ranges over values.
- Metavariable σ ranges over environments, which are maps from identifiers to values.
- If we say "the result is v" while explaining evaluation of e, then e results in v.
- If we say "throw an exception" while explaining evaluation of e, then e causes the exception.
- An exception always carries a single value. If an exception carries v, then we call it an exception carrying v. If the value is unimportant, we can omit the "carrying v" part.
- We use the word "must" to represent requirements. If a requirement is violated, then a run-time error occurs. A run-time error differs from an exception. Any occurrence of a run-time error immediately terminates the execution.

The following explains how each expression is evaluated.

Case x:

(1) Let σ be the current environment.
(2) x must be in the domain of σ.
(3) The result is $\sigma(x)$.

Case n:

(1) The result is n.

Case b:

(1) The result is b.

Case $e_1 \oplus e_2$:

(∗) Suppose that $\oplus \in \{+, \times, \div, \text{mod}, =, <\}$.
(1) Evaluate e_1.
(2) If e_1 causes an exception, then
  (a) Throw the same exception.
(3) Else if e_1 results in v_1, then
  (a) v_1 must be an integer.
  (b) Evaluate e_2.
  (c) If e_2 causes an exception, then
    (i) Throw the same exception.
  (d) Else if e_2 results in v_2, then
    (i) v_2 must be an integer.
    (ii) (v_1, v_2) must be in the domain of $\oplus$. Note that $+, \times \in (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}$, $\div, \text{mod} \in (\mathbb{Z}, \mathbb{Z} \setminus \{0\}) \rightarrow \mathbb{Z}$, and $=, < \in (\mathbb{Z}, \mathbb{Z}) \rightarrow \{\text{true}, \text{false}\}$.
    (iii) The result is $v_1 \oplus v_2$.

Case if e_1 e_2 e_3:

(1) Evaluate e_1.
(2) If e_1 causes an exception, then
  (a) Throw the same exception.
(3) Else if e_1 results in v_1, then
  (a) v_1 must be a boolean.
  (b) If v_1 is true, then
    (i) Evaluate e_2.
    (ii) If e_2 causes an exception, then
      (A) Throw the same exception.
    (iii) Else if e_2 results in v_2, then
      (A) The result is v_2.
  (c) Else if v_1 is false, then
    (i) Evaluate e_3.
    (ii) If e_3 causes an exception, then
      (A) Throw the same exception.
    (iii) Else if e_3 results in v_3, then
      (A) The result is v_3.

Case $(e_1, \cdots, e_i)$:

(1) Evaluate e_1.
(2) If e_1 causes an exception, then
  (a) Throw the same exception.
(3) Else if e_1 results in v_1, then
  (a) Evaluate e_{k+1} in the same manner after evaluating e_k.
  (b) Repeat (a) until e_i is evaluated.
  (c) The result is a tuple consisting of the values from v_1 to v_i.

Case $e.i$:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) v must be a tuple whose length is greater than or equal to i.
  (b) The result is the ith element of v. Note the the beginning index is one.

Case Nil:

(1) The result is the empty list.

Case Cons e_1 e_2:

(1) Evaluate e_1.
(2) If e_1 causes an exception, then
  (a) Throw the same exception.
(3) Else if e_1 results in v_1, then
  (a) Evaluate e_2.
  (b) If e_2 causes an exception, then
    (i) Throw the same exception.
  (c) Else if e_2 results in v_2, then
    (i) v_2 must be either the empty list or a nonempty list.
    (ii) The result is a nonempty list whose head is v_1 and tail is v_2.

Case e.isEmpty:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) v must be either the empty list or a nonempty list.
  (b) If v is the empty list, then
    (i) The result is true.
  (c) Else if v is a nonempty list, then
    (i) The result is false.

Case e.head:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) v must be a nonempty list.
  (b) The result is the head of v.

Case e.tail:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) v must be a nonempty list.
  (b) The result is the tail of v.

Case val x=e_1 in e_2:

(1) Evaluate e_1.
(2) If e_1 causes an exception, then
  (a) Throw the same exception.
(3) Else if e_1 results in v_1, then
  (a) Add mapping from x to v_1 to the current environment.
  (b) Let σ_{new} be the new environment.
  (c) Evaluate e_2 under σ_{new}.
  (d) If e_2 causes an exception, then
    (i) Throw the same exception.
  (e) Else if e_2 results in v_2, then
    (i) The result is v_2.

Case vcc x in e:

(1) Let v_c be the current continuation.

(2) Add mapping from x to v_c to the current environment.
(3) Let σ_{new} be the new environment.
(4) Evaluate e under σ_{new}.
(5) If e causes an exception, then
  (a) Throw the same exception.
(6) Else if e results in v, then
  (a) The result is v.

Case $\lambda x_1 \cdots x_i.e$:

(1) Let σ be the current environment.
(2) The result is a closure whose parameters are from x_1 to x_i, body is e, and environment is σ.

Case $d_1 \cdots d_i\ e$:

(1) Let $x_1, \cdots, x_i$ be the names of $d_1, \cdots, d_i$.
(2) Let $v_1, \cdots, v_i$ be the closures of $d_1, \cdots, d_i$.
(3) Add mapping from x's to v's to the current environment.
(4) Let σ_{new} be the new environment.
(5) The environment of every v_k needs to be σ_{new}.
(6) Evaluate e under σ_{new}.
(7) If e causes an exception, then
  (a) Throw the same exception.
(8) Else if e results in v, then
  (a) The result is v.

Case $e(e_1, \cdots, e_i)$:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) v must be either a closure or a continuation.
  (b) Evaluate e_1.
  (c) If e_1 causes an exception, then
    (i) Throw the same exception.
  (d) Else if e_1 results in v_1, then
    (i) Evaluate e_{k+1} in the same manner after evaluating e_k.
    (ii) Repeat (i) until e_i is evaluated.
    (iii) If v is a closure, then
      (A) The number of parameters must equal the number of arguments.
      (B) Let $x_1, \cdots, x_i$ be the names of the parameters of v.
      (C) Let e_c be the body of v.
      (D) Let σ_c be the environment of v.
      (E) Add mapping from x's to v's to σ_c.
      (F) Let σ_{new} be the new environment.
      (G) Evaluate e_c under σ_{new}.
      (H) If e_c causes an exception, then
        ① Throw the same exception.
      (I) Else if e_c results in v_c, then
        ① The result is v_c.
    (iv) Else if v is a continuation, then
      (A) There must be a single argument.
      (B) Call v with v_1 as an argument.

Case e is τ:

(∗) The type of a value is as the following:
  - The type of an integer is Int.
  - The type of a boolean is Boolean.
  - The type of a tuple is Tuple.
  - The type of the empty list is List.
  - The type of a nonempty list is List.
  - The type of a closure is Function.
  - The type of a continuation is Function.

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) If the type of v is τ, then
    (i) The result is true.
  (b) If the type of v is not τ, then
    (i) The result is false.

Case throw e:

(1) Evaluate e.
(2) If e causes an exception, then
  (a) Throw the same exception.
(3) Else if e results in v, then
  (a) Throw an exception carrying v.

Case try e_1 catch e_2:

(1) Evaluate e_1.
(2) If e_1 causes an exception carrying v_e, then
  (a) Evaluate e_2.
  (b) If e_2 causes an exception, then
    (i) Throw the same exception.
  (c) Else if e_2 results in v_2, then
    (i) v_2 must be either a closure or a continuation.
    (ii) If v_2 is a closure, then
      (A) There must be a single parameter.

(B) Let x be the name of the parameter of v_2.
(C) Let e_c be the body of v_2.
(D) Let σ_c be the environment of v_2.
(E) Add mapping from x to v_e to σ_c.
(F) Let σ_{new} be the new environment.
(G) Evaluate e_c under σ_{new}.
(H) If e_c causes an exception, then
  1 Throw the same exception.
(I) Else if e_c results in v_c, then
  1 The result is v_c.

(iii) If v_2 is a continuation, then
  (A) Call v_2 with v_e as an argument.

(3) Else if e_1 results in v_1, then
  (a) The result is v_1.

6 A GUIDE TO THE PROJECT

6.1 The Directory Structure

To start the project, you should place the `proj02` directory under the `src/main/scala/cs320` directory of your exercise #1 project. The `proj02` directory contains three files: `Project02.scala`, `package.scala`, and `proj02.pdf`.

The `Project02.scala` file defines the abstract syntax of X-Fiber. In addition, it implements the parser and the desugarer of X-Fiber. **DO NOT** modify the `Project02.scala` file.

The `package.scala` file contains the empty `interp` function and given test cases. You should complete the `interp` function. Since passing all the given test cases does not guarantee that your interpreter is perfect, we highly recommend you to write your own test cases. As we provide a reference interpreter of X-Fiber, you can find the correct results of your new test cases.

```
cs320
├── build.sbt
├── project
└── src/main/scala/cs320
    ├── Main.scala
    ├── Homework.scala
    ├── proj02
    │   ├── Project02.scala
    │   ├── package.scala
    │   └── proj02.pdf
    ├── ex01
    └── ...
```

6.2 Rules for Implementation

- You can define helper functions.
- If e results in v under the empty environment, then `interp(e, Map(), x => x, None)` must equal to v.
- If execution of e under the empty environment terminates with a run-time error, then `interp(e, Map(), x => x, None)` must terminate by calling the `error` function. Error messages can be any strings.
- Do not import anything.
- Do not use `while` loops, `for` loops, and `break` statements.
- Do not use mutable variables and mutable collections. However, you can mutate the `env` fields of `CloV` instances in the `RecFuns` case.
- Do not modify `Project02.scala`.

A SMALL-STEP SEMANTICS

$$
\begin{array}{rrcl}
\text{Value} & v & \in & \mathbb{V} \\
 & v & ::= & n \mid b \mid (v,\cdots,v) \mid \mathsf{Nil} \mid \mathsf{Cons}\ v\ v \mid \langle \lambda x \cdots x.e, \sigma \rangle \mid \langle k, s \rangle \\
\text{Environment} & \sigma & \in & Id \xrightarrow{\text{fin}} \mathbb{V} \\
\text{Handler} & H & ::= & \cdot \mid \langle k, s \rangle \\
\text{Continuation} & k & ::= & \square \mid \sigma, H \vdash e :: k \mid (+) :: k \mid (\times) :: k \mid (\div) :: k \mid (\mathsf{mod}) :: k \\
 & & \mid & (=) :: k \mid (<) :: k \mid (\sigma, H, e, e) :: k \mid ((i)) :: k \mid (.i) :: k \\
 & & \mid & (\mathsf{Cons}) :: k \mid (\mathsf{isEmpty}) :: k \mid (\mathsf{head}) :: k \mid (\mathsf{tail}) :: k \\
 & & \mid & (x, \sigma, H, e) :: k \mid (@i) :: k \mid ([\tau]) :: k \mid (\leftrightarrow) :: k \\
\text{Stack} & s & ::= & \blacksquare \mid v :: s
\end{array}
$$

Fig. 4. Definitions for Semantics

$\boxed{type(v) = \tau}$

$$
\begin{array}{rclrcl}
type(n) & = & \mathsf{Int} & type(\mathsf{Nil}) & = & \mathsf{List} \\
type(b) & = & \mathsf{Boolean} & type(\mathsf{Cons}\ v_1\ v_2) & = & \mathsf{List} \\
type((v_1,\cdots,v_i)) & = & \mathsf{Tuple} & type(\langle \lambda x_1 \cdots x_i.e, \sigma \rangle) & = & \mathsf{Function} \\
 & & & type(\langle k, s \rangle) & = & \mathsf{Function}
\end{array}
$$

Fig. 5. Types of Values

$$\boxed{k \mid\mid s \longrightarrow k \mid\mid s}$$

$$\frac{x \in Domain(\sigma)}{\sigma, H \vdash x :: k \mid\mid s \longrightarrow k \mid\mid \sigma(x) :: s}$$

$$\sigma, H \vdash n :: k \mid\mid s \longrightarrow k \mid\mid n :: s$$

$$\sigma, H \vdash b :: k \mid\mid s \longrightarrow k \mid\mid b :: s$$

$$\sigma, H \vdash e_1 + e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (+) :: k \mid\mid s$$

$$(+) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 + n_2 :: s$$

$$\sigma, H \vdash e_1 \times e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\times) :: k \mid\mid s$$

$$(\times) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 \times n_2 :: s$$

$$\sigma, H \vdash e_1 \div e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\div) :: k \mid\mid s$$

$$\frac{n_2 \neq 0}{(\div) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 \div n_2 :: s}$$

Fig. 6. Evaluation of Expressions (1/3)

$$\sigma, H \vdash e_1 \text{ mod } e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\text{mod}) :: k \mid\mid s$$

$$\frac{n_2 \neq 0}{(\text{mod}) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 \text{ mod } n_2 :: s}$$

$$\sigma, H \vdash e_1 = e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (=) :: k \mid\mid s$$

$$(=) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 = n_2 :: s$$

$$\sigma, H \vdash e_1 < e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (<) :: k \mid\mid s$$

$$(<) :: k \mid\mid n_2 :: n_1 :: s \longrightarrow k \mid\mid n_1 < n_2 :: s$$

$$\sigma, H \vdash \text{if } e_1 \ e_2 \ e_3 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: (\sigma, H, e_2, e_3) :: k \mid\mid s$$

$$(\sigma, H, e_1, e_2) :: k \mid\mid \text{true} :: s \longrightarrow \sigma, H \vdash e_1 :: k \mid\mid s$$

$$(\sigma, H, e_1, e_2) :: k \mid\mid \text{false} :: s \longrightarrow \sigma, H \vdash e_2 :: k \mid\mid s$$

$$\sigma, H \vdash (e_1, \cdots, e_i) :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \ \cdots \ :: \sigma, H \vdash e_i :: ((i)) :: k \mid\mid s$$

$$((i)) :: k \mid\mid v_i :: \ \cdots \ :: v_1 :: s \longrightarrow k \mid\mid (v_1, \cdots, v_i) :: s$$

$$\sigma, H \vdash e.i :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: (.i) :: k \mid\mid s$$

$$(.i) :: k \mid\mid (v_1, \cdots, v_i, \cdots, v_j) :: s \longrightarrow k \mid\mid v_i :: s$$

$$\sigma, H \vdash \text{Nil} :: k \mid\mid s \longrightarrow k \mid\mid \text{Nil} :: s$$

$$\sigma, H \vdash \text{Cons } e_1 \ e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\text{Cons}) :: k \mid\mid s$$

$$\frac{type(v_2) = \text{List}}{(\text{Cons}) :: k \mid\mid v_2 :: v_1 :: s \longrightarrow k \mid\mid \text{Cons } v_1 \ v_2 :: s}$$

$$\sigma, H \vdash e.\text{isEmpty} :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: (\text{isEmpty}) :: k \mid\mid s$$

$$(\text{isEmpty}) :: k \mid\mid \text{Nil} :: s \longrightarrow k \mid\mid \text{true} :: s$$

$$(\text{isEmpty}) :: k \mid\mid \text{Cons } v_1 \ v_2 :: s \longrightarrow k \mid\mid \text{false} :: s$$

$$\sigma, H \vdash e.\text{head} :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: (\text{head}) :: k \mid\mid s$$

$$(\text{head}) :: k \mid\mid \text{Cons } v_1 \ v_2 :: s \longrightarrow k \mid\mid v_1 :: s$$

$$\sigma, H \vdash e.\text{tail} :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: (\text{tail}) :: k \mid\mid s$$

$$(\text{tail}) :: k \mid\mid \text{Cons } v_1 \ v_2 :: s \longrightarrow k \mid\mid v_2 :: s$$

Fig. 7. Evaluation of Expressions (2/3)

$$\sigma, H \vdash \mathsf{val}\ x\!=\!e_1\ \mathsf{in}\ e_2 :: k \mid\mid s \longrightarrow \sigma, H \vdash e_1 :: (x, \sigma, H, e_2) :: k \mid\mid s$$

$$(x, \sigma, H, e) :: k \mid\mid v :: s \longrightarrow \sigma[x \mapsto v], H \vdash e :: k \mid\mid s$$

$$\sigma, H \vdash \mathsf{vcc}\ x\ \mathsf{in}\ e :: k \mid\mid s \longrightarrow \sigma[x \mapsto \langle k, s\rangle], H \vdash e :: k \mid\mid s$$

$$\sigma, H \vdash \lambda x_1 \cdots x_i.e :: k \mid\mid s \longrightarrow k \mid\mid \langle \lambda x_1 \cdots x_i.e, \sigma\rangle :: s$$

$$\frac{\begin{array}{c} d_1 = \mathsf{def}\ x_1(x_{11}, \cdots, x_{1j_1}) = e_1 \quad \cdots \quad d_i = \mathsf{def}\ x_i(x_{i1}, \cdots, x_{ij_i}) = e_i \\ v_1 = \langle \lambda x_{11} \cdots x_{1j_1}.e_1, \sigma'\rangle \quad \cdots \quad v_i = \langle \lambda x_{i1} \cdots x_{ij_i}.e_i, \sigma'\rangle \qquad \sigma' = \sigma[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i] \end{array}}{\sigma, H \vdash d_1 \cdots d_i\ e :: k \mid\mid s \longrightarrow \sigma', H \vdash e :: k \mid\mid s}$$

$$\sigma, H \vdash e(e_1, \cdots, e_i) :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: \sigma, H \vdash e_1 :: \cdots :: \sigma, H \vdash e_i :: (@i) :: k \mid\mid s$$

$$(@i) :: k \mid\mid v_i :: \cdots :: v_1 :: \langle \lambda x_1 \cdots x_i.e, \sigma\rangle :: s \longrightarrow \sigma[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i], H \vdash e :: k \mid\mid s$$

$$(@1) :: k \mid\mid v :: \langle k', s'\rangle :: s \longrightarrow k' \mid\mid v :: s'$$

$$\sigma, H \vdash e\ \mathsf{is}\ \tau :: k \mid\mid s \longrightarrow \sigma, H \vdash e :: ([\tau]) :: k \mid\mid s$$

$$([\tau]) :: k \mid\mid v :: s \longrightarrow k \mid\mid \mathit{type}(v) = \tau :: s$$

$$\sigma, \langle k', s'\rangle \vdash \mathsf{throw}\ e :: k \mid\mid s \longrightarrow \sigma, \langle k', s'\rangle \vdash e :: k' \mid\mid s'$$

$$\sigma, H \vdash \mathsf{try}\ e_1\ \mathsf{catch}\ e_2 :: k \mid\mid s \longrightarrow \sigma, \langle \sigma, H \vdash e_2 :: (\leftrightarrow) :: (@1) :: k, s\rangle \vdash e_1 :: k \mid\mid s$$

$$(\leftrightarrow) :: k \mid\mid v_2 :: v_1 :: s \longrightarrow k \mid\mid v_1 :: v_2 :: s$$

Fig. 8. Evaluation of Expressions (3/3)

$$k \mid\mid s \longrightarrow^* k \mid\mid s \qquad\qquad \frac{k_1 \mid\mid s_1 \longrightarrow^* k_2 \mid\mid s_2 \quad k_2 \mid\mid s_2 \longrightarrow k_3 \mid\mid s_3}{k_1 \mid\mid s_1 \longrightarrow^* k_3 \mid\mid s_3}$$

Fig. 9. Reflexive Transitive Closure

A.1 Rules for Implementation

(1) If $\emptyset, \cdot \vdash e :: \square \mid\mid \blacksquare \longrightarrow^* \square \mid\mid v :: \blacksquare$, then `interp(`$e$`, Map(), x => x, None)` must equal to v.
(2) If $\forall (k, s) \in \{(k, s) : \emptyset, \cdot \vdash e :: \square \mid\mid \blacksquare \longrightarrow^* k \mid\mid s\}.\exists k'.\exists s'.k \mid\mid s \longrightarrow k' \mid\mid s'$, then `interp(`$e$`, Map(), x => x, None)` must not terminate.
(3) If $\nexists v.\emptyset, \cdot \vdash e :: \square \mid\mid \blacksquare \longrightarrow^* \square \mid\mid v :: \blacksquare$ and (2) is not the case, then `interp(`e`, Map(), x => x, None)` must throw an exception with the `error` function.