# FIBER: A Language with Functions, Integers, Booleans, Eagerness, and Recursion

CS320 Programming Languages Project #1 (Due: May 3, 2020)

## 1 INTRODUCTION

FIBER is a toy language for the first project of the CS320 course. FIBER stands for a language with **f**unctions, **i**ntegers, **b**ooleans, **e**agerness, and **r**ecursion. As the name implies, it features integers, booleans, first-class functions, and recursive functions. In addition, it is an eager language. (*Eagerness* denotes the most usual function application semantics, that the arguments of a function application are evaluated before the function body is evaluated. A later lecture will cover lazy langauges, which are not eager.) More precisely, FIBER supports the following features:

- 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

This document defines FIBER and provides a guide to the project. First, it gives the syntax of FIBER: Section 2 describes the concrete syntax; Secion 3 formalizes the desugaring rules; Section 4 shows the abstract syntax. Second, it defines the operational semantics of FIBER in Section 5. Finally, Section 6 explains the project files and a recommended strategy for the project.

## 2 CONCRETE SYNTAX

The concrete syntax of 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 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 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
     | "(" ")" "=>" expr | id "=>" expr | "(" id {"," id} ")" "=>" expr
     | fdef {fdef} expr
     | expr "(" ")" | expr "(" expr {"," expr} ")"
     | expr "." "isInstanceOf" "[" type "]"
     | "(" 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 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.

Unlike the various kinds of concrete syntax shown in the lectures, the concrete syntax of Fiber is *ambiguous*. It means that a single string can be parsed in multiple ways. For example, `1 + 2 * 3` can result in both Tree a1 and Tree a2 in Figure 1.

To resolve the ambiguity of the concrete syntax, we define *precedence* between binary operators. If $op_1$ precedes $op_2$, then $e_1$ $op_1$ $e_2$ $op_2$ $e_3$ can result in only Tree b1. On the other hand, if $op_2$ precedes $op_1$, Tree b2 is the only possible result.

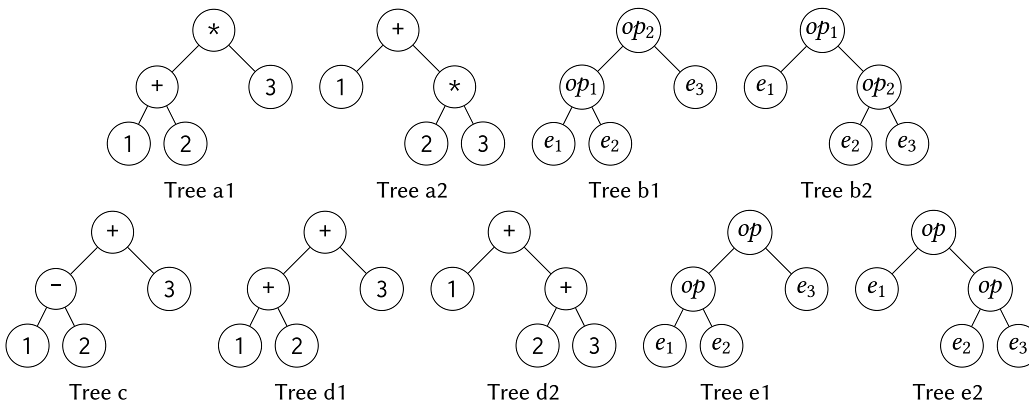


Fig. 1. Parse Trees

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

← higher lower

Fig. 2. Operator Precedence

Figure 2 shows precedence. One appearing earlier in the table precedes one appearing later. For example, since `*` precedes `+`, `1 + 2 * 3` is parsed to only Tree a2. Operators in the same box of the table have the same precedence. If they appear in a single expression, then one appearing first in the expression has the higher precedence in the expression. For instance, `1 - 2 + 3` results in Tree c because `-` and `+` have the same precedence, but `-` appears first in the expression.

Alas, precedence is not enough to resolve the ambiguity. We have problems when an operator appears more than once in an expression. For example, `1 + 2 + 3` can result in both Tree d1 and Tree d2.

We introduce *associativity* of binary operators to solve the problem. A binary operator can be either left-associative or right-associative. If $op_1$ is left-associative, then $e_1$ $op$ $e_2$ $op$ $e_3$ can result in only Tree e1. On the other hand, if $op$ is right-associative, Tree e2 is the only possible result. In Fiber, all the binary operators except `::` are left-associative. Only `::` is right-associative. Thus, `1 + 2 + 3` is parsed to only Tree d1.

## 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 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.

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

$$
\begin{aligned}
\llbracket e_1 \texttt{ \&\& } e_2 \rrbracket &= \texttt{if (}\llbracket e_1 \rrbracket\texttt{) } \llbracket e_2 \rrbracket \texttt{ else false} \\
\llbracket e_1 \texttt{ || } e_2 \rrbracket &= \texttt{if (}\llbracket e_1 \rrbracket\texttt{) true else } \llbracket e_2 \rrbracket \\
\llbracket e\texttt{.nonEmpty} \rrbracket &= \llbracket \texttt{!(}\llbracket e \rrbracket\texttt{.isEmpty)} \rrbracket \\
\llbracket \texttt{val (}x_1\texttt{, } \cdots \texttt{,} x_i\texttt{) = } e_1\texttt{; } e_2 \rrbracket &= \texttt{val } \underline{x} \texttt{ = } \llbracket e_1 \rrbracket\texttt{;} \\
&\quad\ \texttt{val } x_1 \texttt{ = } x\texttt{.\_1;} \\
&\quad\ \texttt{...} \\
&\quad\ \texttt{val } x_i \texttt{ = } x\texttt{.\_}i\texttt{;} \\
&\quad\ \llbracket e_2 \rrbracket \\
\llbracket \texttt{(}e\texttt{)} \rrbracket &= \llbracket e \rrbracket \\
\llbracket \texttt{\{}e\texttt{\}} \rrbracket &= \llbracket e \rrbracket
\end{aligned}
$$

Fig. 3. Desugaring Rules

Figure 3 defines desugaring of Fiber expressions. Let $e$ and $x$ respectively denote an expression and an identifier. An expression $e$ is desugared to $\llbracket e \rrbracket$. For example, after desugaring, `-(1 + 2)` becomes `(1 + 2) * -1`. Note that lines under identifiers imply that the identifiers must be fresh, i.e. have different names from existing ones. For instance, if the entire program is `1 <= 2`, then below is a valid desugaring result.

```
val x = 1;
val y = 2;
x == y || x < y
```

Since the desugaring rules are defined recursively, they can handle complex programs correctly. For example, `--(1 + 2)` becomes `(1 + 2) * -1 * -1` instead of `-(1 + 2) * -1` after desugaring.

## 4 ABSTRACT SYNTAX

Figure 4 describes the abstract syntax of 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; $\tau$ ranges over types, which are either Int, Boolean, Tuple, List, or Function.

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.

Expression if $e_1$ $e_2$ $e_3$ is a conditional expression. $e_1$ is the condition; $e_2$ is the true branch; $e_3$ is the false branch.

Expression $(e_1, \cdots, e_i)$ is a tuple of length $i$. Length $i$ must be greater than one. Expression $e.i$ is a projection from a tuple. The starting index of a tuple is one.

Expressions Nil and $e_1 :: e_2$ create lists. Nil creates the empty list; $e_1 :: e_2$ creates nonempty lists. Expressions $e$.isEmpty, $e$.head, and $e$.tail are operations on lists.

Expression val $x$=$e_1$ in $e_2$ defines a local variable whose name is $x$ and scope is $e_2$.

Expression $\lambda x_1 \cdots x_i.e$ 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. Function definition 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. Expression $d_1 \cdots d_i$ $e$ 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 $e$. Expression $e(e_1, \cdots, e_i)$ is a function application. $e$ should be a function; $e_1$, $\cdots$, $e_i$ are arguments.

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

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

Fig. 4. Abstract Syntax

$$
\begin{array}{rccl}
\text{Value} & v & \in & \mathbb{V} \\
 & v & ::= & n \mid b \mid (v, \cdots, v) \mid \mathsf{Nil} \mid v :: v \mid \langle \lambda x \cdots x.e, \sigma \rangle \\
\text{Environment} & \sigma & \in & \mathit{Id} \xrightarrow{\text{fin}} \mathbb{V}
\end{array}
$$

Fig. 5. Values and Environments

Expression $e$ is $\tau$ checks whether the value of $e$ is $\tau$ at run time.

## 5 OPERATIONAL SEMANTICS

Figure 5 defines values and environments of FIBER. Metavariable $v$ ranges over values; $\mathbb{V}$ denotes the set of every value; metavariable $\sigma$ ranges over environments.

A value is either an integer, a boolean value, a tuple, a list, or a closure. Value $(v_1, \cdots, v_i)$ denotes a tuple whose elements are $v_1, \cdots, v_n$. Nil is the empty list, and $v_1 :: v_2$ is a nonempty list whose head is $v_1$ and tail is $v_2$. $\langle \lambda x_1 \cdots x_i.e, \sigma \rangle$ is a closure of a function whose parameters are $x_1, \cdots, x_i$ and body is $e$ created under environment $\sigma$.

An environment is a finite map from identifiers to values.

Figure 6 and 7 show the operational semantics of FIBER. $\sigma \vdash e \Rightarrow v$ implies that if one evaluates $e$ under $\sigma$, then the result is $v$. The following briefly describes the rules:

- $x$ results in $\sigma(x)$ if $x$ is in the domain of $\sigma$, which is the current environment.
- $n$ results in $n$.

$$\boxed{\sigma \vdash e \Rightarrow v} \qquad \frac{x \in \mathit{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \sigma \vdash n \Rightarrow n \qquad \sigma \vdash b \Rightarrow b$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \div e_2 \Rightarrow n_1 \div n_2} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \ \mathsf{mod}\ e_2 \Rightarrow n_1 \ \mathsf{mod}\ n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 = e_2 \Rightarrow n_1 = n_2} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow \mathsf{true} \quad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash \mathsf{if}\ e_1\ e_2\ e_3 \Rightarrow v} \qquad \frac{\sigma \vdash e_1 \Rightarrow \mathsf{false} \quad \sigma \vdash e_3 \Rightarrow v}{\sigma \vdash \mathsf{if}\ e_1\ e_2\ e_3 \Rightarrow v}$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \sigma \vdash e_i \Rightarrow v_i}{\sigma \vdash (e_1, \cdots, e_i) \Rightarrow (v_1, \cdots, v_i)} \qquad \frac{\sigma \vdash e \Rightarrow (v_1, \cdots, v_i, \cdots, v_{i'})}{\sigma \vdash e.i \Rightarrow v_i}$$

$$\sigma \vdash \mathsf{Nil} \Rightarrow \mathsf{Nil} \qquad \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2 \quad \mathit{type}(v_2) = \mathsf{List}}{\sigma \vdash e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Fig. 6. Evaluation of Expressions (1/2)

- $b$ results in $b$.
- Suppose that $\oplus \in \{+, \times, \div, \mathsf{mod}, =, <\}$. For the evaluation of $e_1 \oplus e_2$, $e_1$ and $e_2$ must be evaluated, and the results have to be integers. Let the integers be $n_1$ and $n_2$. The result of $e_1 \oplus e_2$ is $n_1 \oplus n_2$. Note that $+, \times \in (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$, $\div, \mathsf{mod} \in (\mathbb{Z}, \mathbb{Z} \setminus \{0\}) \to \mathbb{Z}$, and $=, < \in (\mathbb{Z}, \mathbb{Z}) \to \{\mathsf{true}, \mathsf{false}\}$.
- For the evaluation of if $e_1$ $e_2$ $e_3$, $e_1$ must be evaluated, and the result has to be a boolean. If the result is true, then the result of if $e_1$ $e_2$ $e_3$ equals the result of $e_2$. Otherwise, the result equals the result of $e_3$.
- For the evaluation of $(e_1, \cdots, e_i)$, all of $e_1, \cdots, e_i$ must be evaluated. Let their results be $v_1$, $\cdots$, $v_i$. Then the result of $(e_1, \cdots, e_i)$ is $(v_1, \cdots, v_i)$.
- For the evaluation of $e.i$, $e$ must be evaluated, and the result has to be a tuple whose length is greater than or equal to $i$. Let the tuple be $(v_1, \cdots, v_{i'})$. Then the result of $e.i$ is $v_i$.
- Nil results in Nil.
- For the evaluation of $e_1 :: e_2$, $e_1$ and $e_2$ must be evaluated. Let the results be $v_1$ and $v_2$. $v_2$ has to be a list. The result of $e_1 :: e_2$ is $v_1 :: v_2$.
- For the evaluation of $e$.isEmpty, $e$ must be evalauted, and the result has to be a list. If the list is Nil, then the result of $e$.isEmpty is true. Otherwise, the result is false.
- For the evaluation of $e$.head, $e$ must be evalauted, and the result has to be a nonempty list. Let the list be $v_1 :: v_2$. The result of $e$.head is $v_1$.
- For the evaluation of $e$.tail, $e$ must be evalauted, and the result has to be a nonempty list. Let the list be $v_1 :: v_2$. The result of $e$.tail is $v_2$.

$$\frac{\sigma \vdash e \Rightarrow \mathsf{Nil}}{\sigma \vdash e.\mathsf{isEmpty} \Rightarrow \mathsf{true}} \qquad \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{isEmpty} \Rightarrow \mathsf{false}}$$

$$\frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{head} \Rightarrow v_1} \qquad \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{tail} \Rightarrow v_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \mathsf{val}\ x\mathsf{=}e_1\ \mathsf{in}\ e_2 \Rightarrow v_2} \qquad \sigma \vdash \lambda x_1 \cdots x_i.e \Rightarrow \langle \lambda x_1 \cdots x_i.e, \sigma \rangle$$

$$\frac{\begin{array}{c} d_1 = \mathsf{def}\ x_1(x_{11}, \cdots, x_{1i_1}) \mathsf{=} e_1 \quad \cdots \quad d_i = \mathsf{def}\ x_i(x_{i1}, \cdots, x_{ii_i}) \mathsf{=} e_i \\ v_1 = \langle \lambda x_{11} \cdots x_{1i_1}.e_1, \sigma' \rangle \quad \cdots \quad v_i = \langle \lambda x_{i1} \cdots x_{ii_i}.e_i, \sigma' \rangle \\ \sigma' = \sigma[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i] \quad \sigma' \vdash e \Rightarrow v \end{array}}{\sigma \vdash d_1 \cdots d_i\ e \Rightarrow v}$$

$$\frac{\begin{array}{c} \sigma \vdash e \Rightarrow \langle \lambda x_1 \cdots x_i.e', \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \sigma \vdash e_i \Rightarrow v_i \\ \sigma'[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i] \vdash e' \Rightarrow v \end{array}}{\sigma \vdash e(e_1, \cdots, e_i) \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash e\ \mathsf{is}\ \tau \Rightarrow type(v) = \tau}$$

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

- For the evaluation of val $x$=$e_1$ in $e_2$, $e_1$ must be evaluated. Let the result be $v_1$. Add mapping from $x$ to $v_1$ to the current environment, and call the new environment $\sigma_{new}$. The result of val $x$=$e_1$ in $e_2$ is the result of evaluating $e_2$ under $\sigma_{new}$.
- $\lambda x_1 \cdots x_i.e$ results in a closure that captures the current environment.
- Consider $d_1 \cdots d_i\ e$. Let the names of functions defined by $d_1, \cdots, d_i$ be $x_1, \cdots, x_i$ and the closures of the functions be $v_1, \cdots, v_i$. Add mapping from $x$'s to $v$'s to the current environment, and call the new environment $\sigma_{new}$. $v$'s must capture the new environment $\sigma_{new}$ instead of the old environment. The result of $d_1 \cdots d_i\ e$ equals the result of evaluating $e$ under $\sigma_{new}$.
- For the evaluation of $e(e_1, \cdots, e_i)$, $e$, $e_1, \cdots, e_i$ must be evaluated, and the result of $e$ must be a closure that has $i$ parameters. Let the names of the parameters be $x_1, \cdots, x_i$ and the results of $e_1, \cdots, e_i$ be $v_1, \cdots, v_i$. Add mapping from $x$'s to $v$'s to the environment of the closure, and call the new environment $\sigma_{new}$. The result of $e(e_1, \cdots, e_i)$ equals the result of evaluating the body of the closure under $\sigma_{new}$.
- For the evaluation of $e$ is $\tau$, $e$ must be evaluated. Let the result be $v$. If the type of $v$ equals $\tau$, then the result of $e$ is $\tau$ is true. Otherwise, the result is false. Figure 8 defines the types of values.

## 6 A GUIDE TO THE PROJECT

### 6.1 The Directory Structure

To start the project, you should place the `proj01` directory under the `src/main/scala/cs320` directory of your exercise #1 project. The `proj01` directory contains three files: `Project01.scala`, `package.scala`, and `proj01.pdf`.

$$\boxed{type(v) = \tau} \qquad \begin{array}{rcl} type(n) &=& \mathsf{Int} \\ type((v_1, \cdots, v_i)) &=& \mathsf{Tuple} \\ type(v_1 :: v_2) &=& \mathsf{List} \end{array} \qquad \begin{array}{rcl} type(b) &=& \mathsf{Boolean} \\ type(\mathsf{Nil}) &=& \mathsf{List} \\ type(\langle \lambda x_1 \cdots x_i.e, \sigma\rangle) &=& \mathsf{Function} \end{array}$$

Fig. 8. Types of Values

The `Project01.scala` file defines the abstract syntax of Fiber. In addition, it implements the parser and the desugarer of Fiber. **DO NOT** modify the `Project01.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 Fiber, you can find the correct results of your new test cases.

```
cs320
├─build.sbt
├─project
└─src/main/scala/cs320
  ├─Main.scala
  ├─Homework.scala
  ├─proj01
  │ ├─Project01.scala
  │ ├─package.scala
  │ └─proj01.pdf
  ├─ex01
  └─...
```

### 6.2 Rules for Implementation

- You can define helper functions.
- If $\sigma \vdash e \Rightarrow v$, then `interp(`$e$`, `$\sigma$`)` must eqaul to $v$.
- If $\forall v.\neg(\sigma \vdash e \Rightarrow v)$, then `interp(`$e$`, `$\sigma$`)` must throw an exception with the `error` function or does not terminate.
- Do not import any other libraries.
- 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 `Project01.scala`.

### 6.3 A Recommended Schedule

You do not need to follow below. You can decide your own schedule by yourself.

(1) After the April 8 (Functions) lecture:
  (a) Implement the `IntE`, `Add`, `Mul`, `Div`, and `Mod` cases. You should pass the `int`, `add`, `sub`, `mul`, `div`, `mod`, and `neg` tests.
  (b) Implement the `BooleanE`, `Eq`, and `Lt` cases. You should pass the `boolean`, `eq`, and `lt` tests.
  (c) Implement the `TupleE` and `Proj` cases. You should pass the `tuple1`, `tuple2`, `proj1`, and `proj2` tests.
  (d) Implement the `NilE`, `ConsE`, `Empty`, `Head`, and `Tail` cases. You should pass the `nil`, `cons`, `isemtpy1`, `isempty2`, `head`, `tail`, and `tail-head` tests.
  (e) Implement the `Id` and `Val` cases. You should pass the `val1` and `val2` tests.
  (f) Implement the `Fun` and `App` cases. You should pass the `fun`, `app1`, `app2`, and `app3` tests.
  (g) Implement the `Test` case. You should pass the `type1`, `type2`, `type3`, and `type4` tests.

(2) After the April 20 (Implementing Recursion) lecture:
  (a) Implement the `If` case. You should pass the `if`, `not`, `and`, `or`, `neq`, `lte`, `gt`, `gte`, `nonempty` tests.
  (b) Implement the `RecFuns` case. You should pass the `rec1` and `rec2` tests.