Project 3: A Functional Language

CAS CS 320: Principles of Programming Languages

Due April 29, 2024 by 11:59PM

Introduction

Stack-oriented languages are powerful and fairly easy to implement, but they are less user-friendly than modern languages like Python and OCaml. In this project, you will be building a toy compiler for a functional language with OCaml-like syntax. You are given:

- ▷ a parser for a high-level OCaml-like syntax
- ▷ a base interpreter for a stack-oriented language with lexical scoping

and it is your task to implement the following collection of functions that convert a program in the high-level syntax to a program in the stack-oriented language which can be passed to the given interpreter:

- ▷ `desugar : top_prog -> lexpr`
- ▷ `translate : lexpr -> stack_prog`
- ▷ `serialize : stack_prog -> string`

The function `desugar` converts a program in the high-level syntax to a semantically equivalent expression in a low-level syntax that is easier to work with. The function `translate` converts an expression in the low-level syntax to a semantically-equivalent program in the stack-oriented language. The function `serialize` serializes a program in the stack-oriented language to a `string` which can be parsed by the interpreter. For example, a high-level program:

```
let k x y = x
let _ = trace (k 5 10)
```

can be desugared to a low-level expression (represented as an ADT):

```
(fun k ->
(fun _ -> ())
  (trace (k 5 10)))
  (fun x -> fun y -> x)
```

which can be translated into a program in the stack-oriented language:

```
fun C begin swap assign AX
  fun C begin swap assign AY
    lookup AX swap return
  end
  swap return
end
fun C begin swap assign AK
  push 10 push 5 lookup AK call call
  trace push unit
  fun C begin swap assign BK
    push unit swap return
```

```
  end
  call swap return
end
call
```

which can be executed by the base interpreter. This program should terminate with trace `["5"]`.

There are quite a few moving pieces here. We first present the syntax and semantics of the stack-oriented language. We then present the syntax of the high-level OCaml-like language. Finally we present the syntax and semantics of the low-level language, along with the rules for desugaring high-level programs to low-level expressions. The main challenge will be thinking about how to "run" low-level expressions as programs in the stack-oriented language.

The Base Stack-Oriented Language

Our based language is very simple, and similar to the other stack oriented-languages we have previously seen. One notable difference is that lexical scoping is implemented via **continuation passing**, and so variables are not mutable (not a problem if we're compiling from a functional language). See below for more details.

Syntax

A program in our base language is given by the following grammar. The given parser is whitespace agnostic except that there can be no whitespace between the letters of an identifier or the digits of a number.

```
<prog>   ::= {<com>}
<com>    ::= push <const> | swap | trace
           | add | sub | mul | div | lt
           | if <prog> else <prog> end
           | fun <ident> begin <prog> end | call | return
           | assign <ident> | lookup <ident>
<const>  ::= <bool> | <nat> | unit
<bool>   ::= true | false
<nat>    ::= <digit>{<digit>}
<ident>  ::= <letter>{<letter>}
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= 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
```

Semantics

A *configuration* is a 4-tuple $\langle\ S\ ,\ E\ ,\ T\ ,\ P\ \rangle$ consisting of

- S (`value list`): a stack of values ($\mathbb{V}$), which may be unit elements (unit), integers ($\mathbb{Z}$), Boolean values ($\mathbb{B}$), or closures ($\mathbb{C}$) (in set-notation, we write $\mathbb{V} = \{\mathsf{unit}\} \cup \mathbb{Z} \cup \mathbb{B} \cup \mathbb{C}$)
- E (`(ident * value) list`): a collection of variable bindings
- T (`string list`): a list of string printed while executing the program

- ▷ P (`stack_prog`): a program (i.e., a list of commands)

We write panic for the configuration

$$\langle\ \varnothing\ ,\ \varnothing\ ,\ \texttt{"panic"} :: T\ ,\ \epsilon\ \rangle$$

where T is understood from context to be the configuration on the left-hand side of a reduction.

A *closure* is a 3-tuple $[\ F\ ,\ C\ ,\ P\ ]$ consisting of

- ▷ F (`ident`): the name of the closure. We need this to implement recursive functions.
- ▷ C (`(ident * value) list`): a list of capture bindings, which we can think of as a snapshot of the entire environment when the closure was defined (this is possible because variables are immutable)
- ▷ P (`stack_prog`): the body of the function, represented as a list of commands

The operational semantics of this language are given as follows. We present the familiar rules without any exposition.

(The remainder of this pages has been intensionally left blank.)

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{push}\ c\ P\ \rangle \longrightarrow \langle\ c :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\text{push})$$

$$\frac{}{\langle\ a :: b :: S\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \langle\ b :: a :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\text{swap})$$

$$\frac{}{\langle\ a :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{swapErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{swap}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{swapErr}_2)$$

$$\frac{}{\langle\ a :: S\ ,\ E\ ,\ T\ ,\ \texttt{trace}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ \text{toString}(a) :: T\ ,\ P\ \rangle}\ (\text{trace})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{trace}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{traceErr})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{add}\ P\ \rangle \longrightarrow \langle\ (m + n) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\text{add})$$

$$\frac{m \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{add}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{addErr}_1) \qquad \frac{n \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{add}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{addErr}_2)$$

$$\frac{}{\langle\ m :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{add}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{addErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{add}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{addErr}_4)$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{sub}\ P\ \rangle \longrightarrow \langle\ (m - n) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\text{sub})$$

$$\frac{m \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{sub}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{subErr}_1) \qquad \frac{n \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{sub}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{subErr}_2)$$

$$\frac{}{\langle\ m :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{sub}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{subErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{sub}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{subErr}_4)$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{mul}\ P\ \rangle \longrightarrow \langle\ (m * n) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\text{mul})$$

$$\frac{m \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{mul}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{mulErr}_1) \qquad \frac{n \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{mul}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{mulErr}_2)$$

$$\frac{}{\langle\ m :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{mul}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{mulErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{mul}\ P\ \rangle \longrightarrow \text{panic}}\ (\text{mulErr}_4)$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z} \qquad n \neq 0}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \langle\ (m/n) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{div})$$

$$\frac{n = 0}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divZeroErr})$$

$$\frac{m \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_1) \qquad \frac{n \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_2)$$

$$\frac{}{\langle\ m :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{div}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_4)$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{lt}\ P\ \rangle \longrightarrow \langle\ (m < n) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{lt})$$

$$\frac{m \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{lt}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_1) \qquad \frac{n \notin \mathbb{Z}}{\langle\ m :: n :: S\ ,\ E\ ,\ T\ ,\ \texttt{lt}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_2)$$

$$\frac{}{\langle\ m :: \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{lt}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_3) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{lt}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_4)$$

$$\frac{}{\langle\ \mathsf{true} :: S\ ,\ E\ ,\ T\ ,\ \texttt{if}\ Q_1\ \texttt{else}\ Q_2\ \texttt{end}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ Q_1\ P\ \rangle}\ (\mathsf{ifTrue})$$

$$\frac{}{\langle\ \mathsf{false} :: S\ ,\ E\ ,\ T\ ,\ \texttt{if}\ Q_1\ \texttt{else}\ Q_2\ \texttt{end}\ P\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ Q_2\ P\ \rangle}\ (\mathsf{ifFalse})$$

$$\frac{x \notin \mathbb{B}}{\langle\ x :: S\ ,\ E\ ,\ T\ ,\ \texttt{if}\ Q_1\ \texttt{else}\ Q_2\ \texttt{end}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ifErr}_1)$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{if}\ Q_1\ \texttt{else}\ Q_2\ \texttt{end}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{ifErr}_2)$$

$$\frac{\mathsf{fetch}(E, X) \neq \bot}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{lookup}\ X\ P\ \rangle \longrightarrow \langle\ \mathsf{fetch}(E, X) :: S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{lookup})$$

$$\frac{\mathsf{fetch}(E, X) = \bot}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{lookup}\ X\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{fetchErr}_1)$$

$$\frac{v \notin \mathbb{C}}{\langle\ v :: S\ ,\ E\ ,\ T\ ,\ \texttt{assign}\ X\ P\ \rangle \longrightarrow \langle\ S\ ,\ \mathsf{update}(E, X, v)\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{assign})$$

$$\frac{}{\langle\ [\ F\ ,\ C\ ,\ Q\ ] :: S\ ,\ E\ ,\ T\ ,\ \texttt{assign}\ X\ P\ \rangle \longrightarrow \langle\ S\ ,\ \mathsf{update}(E, X, [\ X\ ,\ C\ ,\ Q\ ])\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{assignFun})$$

$$\frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{assign}\ X\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{assignErr})$$

Functions

We will look in more detail at the rules for functions. First, note that in assignFun above, we update the name in the closure whenever we bind a variable to a closure. This is so we can define anonymous functions and then give them names after they've been defined.

When a function is defined, a closure is put on the stack. The closure keeps track of the name of the function and the *entire* environment when the function is defined. This should never fail.

$$\frac{}{\langle\ S\ ,\ E\ ,\ T\ ,\ \texttt{fun}\ F\ \texttt{begin}\ Q\ \texttt{end}\ P\ \rangle \longrightarrow \langle\ [\ F\ ,\ E\ ,\ Q\ ]::S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{pushFun})$$

A function is called by pushing a closure to the stack and using the `call` command. The environment and the current program are then updated to the captured variables and the program of the closure, respectively. The closure at the top of the stack is then replaced with a *different* closure which holds the current environment and the current program, used for returning after executing the function (in essence, we are simulating the call stack within the stack itself). This is called *continuation-passing*, and CC below stands for "current continuation". Also note that the set of captured bindings of the called closure is updated to include the called closure with its given name. This is so we can make recursive calls in the body of the function. The `call` command should fail if the top of the stack is not a closure, or if the stack is empty.

$$\frac{}{\langle\ [\ F\ ,\ C\ ,\ Q\ ]::S\ ,\ E\ ,\ T\ ,\ \texttt{call}\ P\ \rangle \longrightarrow \langle\ [\ \mathsf{CC}\ ,\ E\ ,\ P\ ]\ ,\ \mathsf{update}(C,F,[\ F\ ,\ C\ ,\ Q\ ])\ ,\ T\ ,\ Q\ \rangle}\ (\mathsf{call})$$

$$\frac{v \notin \mathbb{C}}{\langle\ v::S\ ,\ E\ ,\ T\ ,\ \texttt{call}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{callErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{call}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{callErr}_2)$$

We return from a function call using the command `return` when there is a closure on top of the stack. This will only have the intended behavior if this closure is the "current continuation" that was pushed when the function was called. The current environment and program are again replaced with the captured variables and program of the closure, respectively (note the similarity with function calling). This command should fail if the top of the stack is not a closure or the stack is empty.

$$\frac{}{\langle\ [\ F\ ,\ E\ ,\ P\ ]::S\ ,\ C\ ,\ T\ ,\ \texttt{return}\ Q\ \rangle \longrightarrow \langle\ S\ ,\ E\ ,\ T\ ,\ P\ \rangle}\ (\mathsf{return})$$

$$\frac{v \notin \mathbb{C}}{\langle\ v::S\ ,\ E\ ,\ T\ ,\ \texttt{return}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{retErr}_1) \qquad \frac{}{\langle\ \varnothing\ ,\ E\ ,\ T\ ,\ \texttt{return}\ P\ \rangle \longrightarrow \mathsf{panic}}\ (\mathsf{retErr}_2)$$

The last detail, which we will leave somewhat opaque, is *how do we pass parameters to functions? And how do we return values?* In short, we put the parameters and return values *under* the closures in the stack. This means using the `swap` command liberally to manipulate the stack before calling a function or returning from a function.

The following is an example program which implements the squared distance function from the first project.

```
fun SQDIST
begin
  swap assign X swap assign Y
  lookup X lookup X mul
  lookup Y lookup Y mul
  add swap return
end
assign SQDIST

push 2 push 3 lookup SQDIST call
trace
```

The first line in the body of the function gets the two values under the closure that will be on the stack when the function is called. The last line puts the return value under that same closure before returning. After executing this program, the trace is `["13"]`.

High-Level Syntax

A program in the high-level language is given by the following grammar. It is very similar to the grammar of OCaml.

```
<prog>   ::= {let <ident> <args> = <expr>}
<expr>   ::= () | <num> | <bool> | <ident>
           | <uop> <expr> | <expr> <bop> <expr>
           | fun <args1> -> <expr> | <expr> <expr>
           | let <ident> <args> = <expr> in <expr>
           | if <expr> then <expr> else <expr>
           | trace <expr> | ( <expr> )
<args>   ::= {<ident>}
<args1>  ::=  <ident> {<ident>}
<uop>    ::= not | -
<bop>    ::= + | - | * | / | && | || | < | <= | > | >= | = | <>
<num>    ::= <digit>{<digit>}
<ident>  ::= (<letter> | _){<letter> | _}
<bool>   ::= true | false
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= 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
```

As presented, the grammar is ambiguous, but the ambiguity is handled in the usual way, by assigning precedence and associativity to the operators. We use the same rules as OCaml for the operators in this language.[1] The parser for this grammar is given. You should look through the definitions of `expr` and `top_prog` to make sure you understand how they represent this grammar.

[1] `https://v2.ocaml.org/api/Ocaml_operators.html`

Low-Level Language

Your primary focus will be the low-level language into which the above high-level syntax is desugared. We present both the syntax and the semantics.

Syntax

An expression in the low-level syntax is given by the following grammar.

```
<lexpr>  ::= <value> | <uop> <lexpr> | <lexpr> <bop> <lexpr> | <lexpr> <lexpr>
           | if <lexpr> then <lexpr> else <lexpr>
           | trace <lexpr> | ( <lexpr> )
<value>  ::= () | <num> | <bool> | <ident> | fun <ident> -> <lexpr>
<uop>    ::= not | -
<bop>    ::= + | - | * | / | && | || | < | <= | > | >= | = | <>
<num>    ::= <digit>{<digit>}
<ident>  ::= (<letter> | _){<letter> | _}
<bool>   ::= true | false
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= 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
```

You will not work with the syntax directly—instead you will work expressions of type `lexpr`. You should look through the definition of `lexpr` to make sure you understand how it represents this grammar.

Desugaring

The function `desugar` converts a `top_prog` to a `lexpr` which is semantically equivalent according to the following rules. Note that the low-level syntax is a strict subset of the high-level syntax, so we only need to describe how to desugar those parts of the high-level syntax which are not a part of the low-level syntax.

- ▷ The program

  ```
  let f1 args = expr
  let f2 args = expr
  ...
  let fn args = expr
  ```

  is equivalent to the high-level expression

  ```
  let f1 args = expr in
  let f2 args = expr in
  ...
  let fn args = expr in
  ()
  ```

- ▷ The high-level expression

  ```
  let f args = expr in expr
  ```

  is equivalent to the high-level expression

```
let f = fun args -> expr in expr
```

- ▷ The high-level expression

  ```
  let x = expr1 in expr2
  ```

  is equivalent to the low-level expression

  ```
  (fun x -> expr2) expr1
  ```

- ▷ The high-level expression

  ```
  fun arg1 arg2 ... argn -> expr
  ```

  is equivalent to the low-level expression

  ```
  fun arg1 -> fun arg2 -> ... fun argn -> expr
  ```

Semantics

A *configuration* is a pair $(\ S\ ,\ T\ )$ consisting of

- ▷ T (`string list`): a list of strings printed while evaluating the expression
- ▷ S: a state which is either a low-level expression (`lexpr`) or an error state error

A *value* ($\mathbb{V}$) is a unit (`()`), an integer ($\mathbb{Z}$), a Boolean value ($\mathbb{B}$) or a function ($\mathbb{F}$). This is exactly the collection of expressions given by the production rule for the nonterminal symbol <value> in grammar for the low-level syntax. Note that we are not required to reduce the body of a function for it to be considered a value, e.g., `fun x -> (fun x -> x) x` is a value. Like before, we use panic to refer to the configuration

$$(\ \texttt{"panic"} :: T\ ,\ \mathsf{error}\ )$$

where T is understood from context to be the trace on the left-hand side of a reduction.

The most important consideration in the rules below is the order in which expressions are evaluated. Some rules are designed to evaluate arguments from right to left, others from left to right. Please take care in reading the rules.

Trace

`trace` should add a string representation of the *value* of its argument to the trace. This means first reducing its argument.

$$\frac{(\ T\ ,\ e\ ) \longrightarrow (\ T'\ ,\ e'\ )}{(\ T\ ,\ \texttt{trace}\ e\ ) \longrightarrow (\ T'\ ,\ \texttt{trace}\ e'\ )}\ (\mathsf{traceRed}) \qquad \frac{v \in \mathbb{V}}{(\ T\ ,\ \texttt{trace}\ v\ ) \longrightarrow (\ \mathsf{toString(v)} :: T\ ,\ \texttt{()}\ )}\ (\mathsf{trace})$$

$$\frac{(\ T\ ,\ e\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ \texttt{trace}\ e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{traceErr})$$

Add

Adding two expressions requires first evaluating the *right-hand* argument, then evaluating the *left-hand* argument, and finally adding those two values together. If any part of this process fails, evaluation should end in the error state. In particular, the left-hand expression should not be evaluated if the right-hand argument fails to be evaluated **or evaluates to something which is not an integer**. Note that in each case, if both arguments evaluate to integers, we replace the *syntactic* operation (a part of the language

rendered purple) with the *semantic* opertion (not a part of the language, rendered as a standard latex symbol). The rules for subtraction and multiplication follow the same pattern.

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ + } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ + } e_2'\ )}\ (\mathsf{addRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ + } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ + } v\ )}\ (\mathsf{addLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ + } n\ ) \longrightarrow (\ T\ ,\ m + n\ )}\ (\mathsf{addNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ + } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ + } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ + } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ + } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_4)$$

Subtract

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ - } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ - } e_2'\ )}\ (\mathsf{subRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ - } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ - } v\ )}\ (\mathsf{subLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ - } n\ ) \longrightarrow (\ T\ ,\ m - n\ )}\ (\mathsf{subNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ - } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{subErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ - } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{subErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ - } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{subErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ - } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{subErr}_4)$$

Multiply

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ * } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ * } e_2'\ )}\ (\mathsf{mulRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ * } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ * } v\ )}\ (\mathsf{mulLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ * } n\ ) \longrightarrow (\ T\ ,\ m * n\ )}\ (\mathsf{mulNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ * } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ * } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ * } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ * } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{mulErr}_4)$$

Divide

The rules for division follows the same pattern as above, but with the added rule that dividing by zero should yield the error state. Note that you don't have to check that the right-hand argument is 0 before

evaluating the left-hand argument (but you still have to verify it is an integer before evaluating the left-hand argument).

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1\ /\ e_2\ ) \longrightarrow (\ T'\ ,\ e_1\ /\ e_2'\ )}\ (\mathsf{divRight}) \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1\ /\ v\ ) \longrightarrow (\ T'\ ,\ e_1'\ /\ v\ )}\ (\mathsf{divLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m\ /\ n\ ) \longrightarrow (\ T\ ,\ m/n\ )}\ (\mathsf{divNum}) \qquad \frac{a \in \mathbb{V}}{(\ T\ ,\ a\ /\ 0\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{divZeroErr})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e\ /\ b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_1) \qquad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a\ /\ e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ /\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_3) \qquad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ /\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{divErr}_4)$$

Negate

Negating an expression required first evaluating that expression and then negating it if it evaluates to an integer. As with the previous arithmetic expressions, evaluation should end in the error state if any part of this process fails.

$$\frac{(\ T\ ,\ e\ ) \longrightarrow (\ T'\ ,\ e'\ )}{(\ T\ ,\ -\ e\ ) \longrightarrow (\ T'\ ,\ -\ e'\ )}\ (\mathsf{negRed}) \qquad \frac{n \in \mathbb{Z}}{(\ T\ ,\ -\ n\ ) \longrightarrow (\ T\ ,\ -n\ )}\ (\mathsf{negNum})$$

$$\frac{v \in \mathbb{V} \qquad v \notin \mathbb{Z}}{(\ T\ ,\ -\ v\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{negErr}_1) \qquad \frac{(\ T\ ,\ e\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ -\ e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{negErr}_2)$$

Less Than

Evaluating binary predicates on integers follows the same pattern as evaluating binary arithmetic operations. We evaluate arguments from right to left, failing if evaluating either argument fails or if the right-hand argument does not evaluate to an integer. We will include the rules for all predicates without additional exposition. Note, however, that the only predicate in our stack-oriented language is less-than. You will have to think about how to compile the other predicates in terms of the primitives available in the language *being careful not to change the evaluation order*.

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1\ <\ e_2\ ) \longrightarrow (\ T'\ ,\ e_1\ <\ e_2'\ )}\ (\mathsf{ltRight}) \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1\ <\ v\ ) \longrightarrow (\ T'\ ,\ e_1'\ <\ v\ )}\ (\mathsf{ltLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m\ <\ n\ ) \longrightarrow (\ T\ ,\ m < n\ )}\ (\mathsf{ltNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e\ <\ b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_1) \qquad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a\ <\ e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ <\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_3) \qquad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ <\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{ltErr}_4)$$

Less Than or Equal

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ <= } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ <= } e_2'\ )}\ (\mathsf{lteRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ <= } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ <= } v\ )}\ (\mathsf{lteLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ <= } n\ ) \longrightarrow (\ T\ ,\ m \leq n\ )}\ (\mathsf{lteNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ <= } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{lteErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ <= } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{lteErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ <= } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{lteErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ <= } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{lteErr}_4)$$

Greater Than

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ > } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ > } e_2'\ )}\ (\mathsf{gtRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ > } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ > } v\ )}\ (\mathsf{gtLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ > } n\ ) \longrightarrow (\ T\ ,\ m > n\ )}\ (\mathsf{gtNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ > } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gtErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ > } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gtErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ > } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gtErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ > } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gtErr}_4)$$

Greater Than or Equal

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ >= } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ >= } e_2'\ )}\ (\mathsf{gteRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ >= } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ >= } v\ )}\ (\mathsf{gteLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ >= } n\ ) \longrightarrow (\ T\ ,\ m \geq n\ )}\ (\mathsf{gteNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ >= } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gteErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ >= } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gteErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ >= } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gteErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ >= } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{gteErr}_4)$$

Equal

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ = } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ = } e_2'\ )}\ (\mathsf{eqRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ = } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ = } v\ )}\ (\mathsf{eqLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ = } n\ ) \longrightarrow (\ T\ ,\ m = n\ )}\ (\mathsf{eqNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ = } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ = } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ = } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ = } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{eqErr}_4)$$

Not Equal

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1 \texttt{ <> } e_2\ ) \longrightarrow (\ T'\ ,\ e_1 \texttt{ <> } e_2'\ )}\ (\mathsf{neqRight}) \quad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{Z}}{(\ T\ ,\ e_1 \texttt{ <> } v\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ <> } v\ )}\ (\mathsf{neqLeft})$$

$$\frac{m \in \mathbb{Z} \qquad n \in \mathbb{Z}}{(\ T\ ,\ m \texttt{ <> } n\ ) \longrightarrow (\ T\ ,\ m \neq n\ )}\ (\mathsf{neqNum})$$

$$\frac{b \in \mathbb{V} \qquad b \notin \mathbb{Z}}{(\ T\ ,\ e \texttt{ <> } b\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{neqErr}_1) \quad \frac{a \in \mathbb{V} \qquad a \notin \mathbb{Z}}{(\ T\ ,\ a \texttt{ <> } e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{neqErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ <> } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{neqErr}_3) \quad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1 \texttt{ <> } e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{neqErr}_4)$$

And

Evaluating the conjunction of two expressions required first evaluating the *left-hand* argument, then evaluating the *right-hand* argument **if the first argument evaluates to** `true`, and finally emitting the value `true` if both arguments evaluate to `true` and emitting `false` otherwise. This short-circuiting is common in most programming languages (including OCaml). Note that these rules imply it is possible to evaluate `false && 15` since the right-hand argument will be ignored. The pattern is similar for disjunction.

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ )}{(\ T\ ,\ e_1 \texttt{ \&\& } e_2\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ \&\& } e_2\ )}\ (\text{andLeftRed}) \qquad \frac{}{(\ T\ ,\ \texttt{false \&\& } e\ ) \longrightarrow (\ T\ ,\ \texttt{false}\ )}\ (\text{andLeftFalse})$$

$$\frac{(\ T\ ,\ e\ ) \longrightarrow (\ T'\ ,\ e'\ )}{(\ T\ ,\ \texttt{true \&\& } e\ ) \longrightarrow (\ T'\ ,\ \texttt{true \&\& } e'\ )}\ (\text{andRightRed}) \qquad \frac{b \in \mathbb{B}}{(\ T\ ,\ \texttt{true \&\& } b\ ) \longrightarrow (\ T\ ,\ b\ )}\ (\text{andRight})$$

$$\frac{v \in \mathbb{V} \qquad v \notin \mathbb{B}}{(\ T\ ,\ v \texttt{ \&\& } e\ ) \longrightarrow \text{panic}}\ (\text{andErr}_1) \qquad \frac{v \in \mathbb{V} \qquad v \notin \mathbb{B}}{(\ T\ ,\ e \texttt{ \&\& } v\ ) \longrightarrow \text{panic}}\ (\text{andErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \text{panic}}{(\ T\ ,\ e_1 \texttt{ \&\& } e_2\ ) \longrightarrow \text{panic}}\ (\text{andErr}_3) \qquad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \text{panic}}{(\ T\ ,\ e_1 \texttt{ \&\& } e_2\ ) \longrightarrow \text{panic}}\ (\text{andErr}_4)$$

Or

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ )}{(\ T\ ,\ e_1 \texttt{ || } e_2\ ) \longrightarrow (\ T'\ ,\ e_1' \texttt{ || } e_2\ )}\ (\text{orLeftRed}) \qquad \frac{}{(\ T\ ,\ \texttt{true || } e\ ) \longrightarrow (\ T\ ,\ \texttt{true}\ )}\ (\text{orLeftTrue})$$

$$\frac{(\ T\ ,\ e\ ) \longrightarrow (\ T'\ ,\ e'\ )}{(\ T\ ,\ \texttt{false || } e\ ) \longrightarrow (\ T'\ ,\ \texttt{false || } e'\ )}\ (\text{orRightRed}) \qquad \frac{b \in \mathbb{B}}{(\ T\ ,\ \texttt{false || } b\ ) \longrightarrow (\ T\ ,\ b\ )}\ (\text{orRight})$$

$$\frac{v \in \mathbb{V} \qquad v \notin \mathbb{B}}{(\ T\ ,\ v \texttt{ || } e\ ) \longrightarrow \text{panic}}\ (\text{orErr}_1) \qquad \frac{v \in \mathbb{V} \qquad v \notin \mathbb{B}}{(\ T\ ,\ e \texttt{ || } v\ ) \longrightarrow \text{panic}}\ (\text{orErr}_2)$$

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow \text{panic}}{(\ T\ ,\ e_1 \texttt{ || } e_2\ ) \longrightarrow \text{panic}}\ (\text{orErr}_3) \qquad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \text{panic}}{(\ T\ ,\ e_1 \texttt{ || } e_2\ ) \longrightarrow \text{panic}}\ (\text{orErr}_4)$$

If-Then-Else

Evaluating an if-then-else expressions means evaluating the condition and, if it evaluates to a boolean value, emitting one of the two case expressions. In particular, neither case should be evaluated until the condition is completely evaluated.

$$\frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ )}{(\ T\ ,\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\ ) \longrightarrow (\ T'\ ,\ \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3\ )}\ (\text{ifRed})$$

$$\frac{}{(\ T\ ,\ \texttt{if true then } e_2 \texttt{ else } e_3\ ) \longrightarrow (\ T\ ,\ e_2\ )}\ (\text{ifTrue})$$

$$\frac{}{(\ T\ ,\ \texttt{if false then } e_2 \texttt{ else } e_3\ ) \longrightarrow (\ T\ ,\ e_3\ )}\ (\text{ifFalse})$$

$$\frac{v \in \mathbb{V} \qquad v \notin \mathbb{B}}{(\ T\ ,\ \texttt{if } v \texttt{ then } e_2 \texttt{ else } e_3\ ) \longrightarrow \text{panic}}\ (\text{ifErr}_1) \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow \text{panic}}{(\ T\ ,\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3\ ) \longrightarrow \text{panic}}\ (\text{ifErr}_2)$$

Function Application

We use call-by-value evaluation in our rules for function application. When evaluating a function applied to an argument, we first evaluate its argument, then the function itself, and finally—given that evaluating the function yields a function value—*substitute* the value of the argument into the body of the function. We will take substitution somewhat informally here to be the replacement of all free occurrences of a variable with a given value in a given expression. This is written $e[v/x]$ (which reads "e with v substituted for x"). Of course, when you compile function application, there will be no notion of substitution. You will have to think about how to *simulate* substitution in a stack-oriented program.

$$\frac{(\ T\ ,\ e_2\ ) \longrightarrow (\ T'\ ,\ e_2'\ )}{(\ T\ ,\ e_1\ e_2\ ) \longrightarrow (\ T'\ ,\ e_1\ e_2'\ )}\ (\mathsf{appRightRed}) \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow (\ T'\ ,\ e_1'\ ) \qquad v \in \mathbb{V}}{(\ T\ ,\ e_1\ v\ ) \longrightarrow (\ T'\ ,\ e_1'\ v\ )}\ (\mathsf{appLeftRed})$$

$$\frac{v \in \mathbb{V}}{(\ T\ ,\ (\texttt{fun}\ x\ \texttt{->}\ e)\ v\ ) \longrightarrow (\ T\ ,\ e[v/x]\ )}\ (\mathsf{appFun})$$

$$\frac{a \in \mathbb{V} \qquad a \notin \mathbb{F}}{(\ T\ ,\ a\ e\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{appErr}_1) \qquad \frac{(\ T\ ,\ e_1\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_2) \qquad \frac{(\ T\ ,\ e_2\ ) \longrightarrow \mathsf{panic}}{(\ T\ ,\ e_1\ e_2\ ) \longrightarrow \mathsf{panic}}\ (\mathsf{addErr}_3)$$

The Task

The task of this project is to compile a program in the high-level syntax to a program in our base stack-oriented language. This means:

- converting a high-level program (`top_prog`) into a low-level expression (`lexpr`) via the desugaring rules given above
- converting a low-level expression (`lexpr`) into an abstract representation of a stack-oriented program (`stack-prog`)
- serializing an abstract representation of a stack-oriented program into a string which can be interpreted.

A couple final notes.

- The bulk of this project will be the second step. How I would recommend thinking about this part: the stack-oriented program that you compile an expression down to should be one which, when executed, leaves only the value of that expression onto the stack. The expression `10` should compile to `push 10` and the expression `(10 + (20 - 2))` should compile to `push 2 push 20 sub push 10 add` (note the order of commands). The function `translate` should recursively *compose* these programs, maintaining the invariant that the compiled program always leaves the correct value on the top of the stack.
- There are a couple subtle details in the rules so please read them carefully. Make sure to maintain the evaluation order, and to check that the compiled program panics at the right time. One question to think about here: *how do you panic when a value on the stack is not an integer?* This is necessary for several of the arithmetic operations above.
- You can verify your evaluation order is correct by building programs in the high-level language with intermediate traces. See some of the examples given in the starter code.
- The stack-oriented language is incredibly simple, which means you'll have to think about how to represent more complex operations like equality. This will be a game in relating different operations (e.g., what is the relationship between `<` and `>=`).
- Remember that the parser for the stack-oriented language is white-space agnostic. So you *can* try to implement `serialize` to properly indent functions and conditionals, but you can also just put each command on its own line.

- ▷ Its a lot of starter code, but not a lot of code to write. Make sure you understand your task before jumping into it.

Running the Compiler

When you're done, you should be able to run:

```
utop compile.ml < file.myml
```

on the command line to print out a compiled program. You can run:

```
utop compile.ml < file.myml > file-compiled.my
```

to redirect the printed program to another file, which you can then run using the base interpreter:

```
utop base_interp.ml < file-compiled.my
```

Alternatively, you can pipe the output of compiling directly into the interpreter:

```
utop compile.ml < file.myml | utop base_interp.ml
```

If you give `compile.ml` an ill-formed high-level program you will see:

```
Exception: TopParseError
```

If you give `base_interp.ml` an ill-formed stack-oriented program you will see:

```
Exception: BaseParseError
```