CS320 Fall2023 Project Part 3

December 2023

1 Overview

The stack-oriented programming language of part1 and part2 is fairly low level and difficult to write non-trivial programs for. In this part of the project, your task is to implement a compiler that compiles a OCaml-like high-level programming language into the stack language of part2. More specifically, you must implement a `compile` function with the following signature.

```
compile : string -> string
```

Given a source string written in the high-level language, `compile` produces a string encoding a *semantically equivalent* program written in the syntax of the stack language of part2. In other words, given an arbitrary high-level program P, compiling P then interpreting its resulting stack program yields the same trace as interpreting P directly using the operational semantics of the high-level language (Section 3).

The concrete syntax of the high-level language is substantially more complex than the stack language. We provide the parser for the high-level language with the following AST and parser. The `parse_prog` function parses a string and produces its corresponding high-level AST. Note that the context free grammar presented in Section 2 **is not representative** of the strings accepted by `parse_prog`. The input to `parse_prog` is transformed by the parser (syntax desugaring + variable scoping) to produce a valid `expr` value. It is these `expr` values that correspond to the grammar of Section 2.

```
type uopr =
  | Neg | Not

type bopr =
  | Add | Sub | Mul | Div | Mod
  | And | Or
  | Lt  | Gt  | Lte | Gte | Eq

type expr =
  | Int of int | Bool of bool | Unit
  | UOpr of uopr * expr
  | BOpr of bopr * expr * expr
  | Var of string
  | Fun of string * string * expr
  | App of expr * expr
  | Let of string * expr * expr
  | Seq of expr * expr
  | Ifte of expr * expr * expr
  | Trace of expr

let parse_prog (s : string) : expr = ...
```

2 Grammar

⟨*digit*⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*nat*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩⟨*nat*⟩

⟨*int*⟩ ::= ⟨*nat*⟩ | `-` ⟨*nat*⟩

⟨*bool*⟩ ::= `true` | `false`

⟨*unit*⟩ ::= `()`

⟨*unop*⟩ ::= `-` | `not`

⟨*binop*⟩ ::= `+` | `-` | `*` | `/` | `mod` | `&&` | `||` | `<` | `>` | `<=` | `>=` | `=`

⟨*char*⟩ ::= `a` | `b` | .. | `z`

⟨*var*⟩ ::= ⟨*char*⟩ | ⟨*var*⟩⟨*char*⟩ | ⟨*var*⟩⟨*digit*⟩

⟨*expr*⟩ ::= ⟨*int*⟩ | ⟨*bool*⟩ | ⟨*unit*⟩
| ⟨*unop*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*var*⟩
| `fun` ⟨*var*⟩ ⟨*var*⟩ `->` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*expr*⟩
| `let` ⟨*var*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
| ⟨*expr*⟩ `;` ⟨*expr*⟩
| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
| `trace` ⟨*expr*⟩

The following table shows how symbols in the grammar correspond to constructors of the `expr` datatype.

Grammar	OCaml Constructor	Description
⟨*int*⟩	`Int of int`	integer constant
⟨*bool*⟩	`Bool of bool`	boolean constant
⟨*unit*⟩	`Unit`	unit constant
⟨*unop*⟩ ⟨*expr*⟩	`UOpr of uopr * expr`	unary operation
⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩	`BOpr of bopr * expr * expr`	binary operation
⟨*var*⟩	`Var of string`	variable
`fun` ⟨*var*⟩ ⟨*expr*⟩ `->` ⟨*expr*⟩	`Fun of string * string * expr`	function
⟨*expr*⟩ ⟨*expr*⟩	`App of expr * expr`	function application
`let` ⟨*var*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩	`Let of string * expr * expr`	let-in expression
⟨*expr*⟩`;` ⟨*expr*⟩	`Seq of expr * expr`	sequence expression
`if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩	`Ifte of expr * expr * expr`	if-then-else expression
`trace` ⟨*expr*⟩	`Trace of expr`	trace expression

3 Operational Semantics

3.1 Configuration of Programs

A program configuration is of the following form.

$$[\,T\,]\ s$$

- T: (**T**race): List of strings logging the program trace.
- s: (**S**tate): Current state of the program. This can be an $\langle expr \rangle$ or a special **Error** state.

Examples:

1. $[\,$`"True"` $:: \epsilon\,]$ `let` x `=` 1 `in let` y `=` 2 `in trace` $(x + y)$
2. $[\,$`"Unit"` $:: \epsilon\,]$ `let` foo `=` `fun` $f\ x$ `->` x `in let` y `=` 2 `in trace` $(foo\ y)$
3. $[\,$`"Panic"` $::$ `"1"` $:: \epsilon\,]$ **Error**

3.2 Single-Step Reduction

The operational semantics of the high-level language is similar to OCaml with a reduced feature set. The semantics are given through the following primitive single-step relation.

$$[\,T_1\,]\ s_1 \rightsquigarrow [\,T_2\,]\ s_2$$

Here, s_1 is the starting program state and T_1 is its associated trace. The reduction relation transforms program s_1 into program s_2 and transforms trace T_1 into trace T_2 in *a single step*. For the rules deriving the single-step relation, s_1 will always be an $\langle expr \rangle$ while s_2 may be an $\langle expr \rangle$ or the special **Error** state. This means that if the **Error** state is ever reached, reduction will not be able to continue further and the program terminates.

Note: Most of the operational semantics rules are simple but repetitive. Readers familiar with OCaml should already have a good understanding of the high-level language.

3.3 Multi-Step Reduction

A separate multi-step reduction relation $[\,T_1\,]\ s_1 \rightsquigarrow [\,T_2\,]\ s_2$ is defined to compose together 0 or more single-step reductions. The rules for deriving the multi-step relation are given as follows.

$$\textsc{StarRefl}\ \frac{}{[\,T\,]\ s \rightsquigarrow^* [\,T\,]\ s} \qquad \textsc{StarStep}\ \frac{[\,T_1\,]\ s_1 \rightsquigarrow^* [\,T_2\,]\ s_2 \qquad [\,T_2\,]\ s_2 \rightsquigarrow [\,T_3\,]\ s_3}{[\,T_1\,]\ s_1 \rightsquigarrow^* [\,T_3\,]\ s_3}$$

- StarRefl: A program configuration "reduces" to itself in 0 steps.
- StarStep: Multi-step can be extended with a single-step if their end state and starting state match.

3.4 Values

We refer to a special subset of expressions as *values*. In particular, values can be characterized by the following grammar. We can see here that integer constants, boolean constants, unit constants and functions are considered values. Basically, values are expressions which can not be reduced further.

$$\begin{aligned}\langle value \rangle ::=\ & \langle int \rangle \mid \langle bool \rangle \mid \langle unit \rangle \\ \mid\ & \texttt{fun}\ \langle var \rangle\ \langle var \rangle\ \texttt{->}\ \langle expr \rangle\end{aligned}$$

Note: Most of the operational semantics rules are simple but repetitive. Readers familiar with OCaml should already have an intuitive understanding of the high-level language.

3.5 Unary Operations

Integer Negation

Integer negation is a unary operation which negates the value of an integer argument.

$$\textsc{NegInt}\ \frac{v \text{ is an integer value}}{[\,T\,]\ \texttt{-}\ v \rightsquigarrow [\,T\,]\ -v} \qquad \textsc{NegExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ \texttt{-}\ m_1 \rightsquigarrow [\,T_2\,]\ \texttt{-}\ m_2}$$

$$\textsc{NegError1}\ \frac{v \text{ is not an integer value}}{[\,T\,]\ \texttt{-}\ m \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}} \qquad \textsc{NegError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ \texttt{-}\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- NegInt: Interpret *syntactic* `-` v as *mathematical negation* $-v$ if applied to an integer value v.
- NegExpr: Structurally reduce the argument of negation if it is not a value.
- NegError1: Applying negation to a value of an incorrect type results in **Error** and panic trace.
- NegError2: A negation expression reduces to **Error** if its argument reduces to **Error**.

Boolean Not

Boolean not is a unary operation which negates the value of a boolean argument.

$$\textsc{NotBool}\ \frac{v \text{ is a bool value}}{[\,T\,]\ \texttt{not}\ v \rightsquigarrow [\,T\,]\ \neg v} \qquad \textsc{NotExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ \texttt{not}\ m_1 \rightsquigarrow [\,T_2\,]\ \texttt{not}\ m_2}$$

$$\textsc{NotError1}\ \frac{v \text{ is not a boolean value}}{[\,T\,]\ \texttt{not}\ v \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}} \qquad \textsc{NotError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ \texttt{not}\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- NotBool: Interpret *syntactic* `not` v as *boolean negation* $\neg v$ if applied to a boolean value v.
- NotExpr: Structurally reduce the argument of `not` if it is not a value.
- NotError1: Applying `not` to a value of an incorrect type results in **Error** and panic trace.
- NotError2: A `not` expression reduces to **Error** if its argument reduces to **Error**.

3.6 Binary Operations

Integer Addition

Integer addition is a binary operation that sums the values of 2 integer arguments.

$$\textsc{AddInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1\ \texttt{+}\ v_2 \rightsquigarrow [T]\ v_1 + v_2} \qquad \textsc{AddExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{+}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{+}\ n} \qquad \textsc{AddExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{+}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{+}\ m_2}$$

$$\textsc{AddError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1\ \texttt{+}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{AddError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{+}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{AddError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{+}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- AddInt: Interpret *syntactic* v_1 `+` v_2 as *mathematical addition* $v_1 + v_2$ if applied to integers v_1 and v_2.
- AddExpr1: Structurally reduce the left-hand side of addition if it is not a value.
- AddExpr2: Structurally reduce the right-hand side of addition if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of addition must be evaluated from left to right in order for AddExpr2 to be applicable.
- AddError1: Applying addition to values of incorrect types results in **Error** and panic trace.
- AddError2: An addition expression reduces **Error** if its left-hand reduces to **Error**.
- AddError3: An addition expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to AddExpr2, the AddError3 rule requires the left-hand side to already be a value.

Integer Subtraction

Integer subtraction is a binary operation that finds the difference of 2 integer arguments.

$$\textsc{SubInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1\ \texttt{-}\ v_2 \rightsquigarrow [T]\ v_1 - v_2} \qquad \textsc{SubExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{-}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{-}\ n} \qquad \textsc{SubExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{-}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{-}\ m_2}$$

$$\textsc{SubError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1\ \texttt{-}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{SubError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{-}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{SubError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{-}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- SubInt: Interpret *syntactic* v_1 `-` v_2 as *mathematical subtraction* $v_1 - v_2$ if applied to integers v_1 and v_2.
- SubExpr1: Structurally reduce the left-hand side of subtraction if it is not a value.
- SubExpr2: Structurally reduce the right-hand side of subtraction if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of subtraction must be evaluated from left to right in order for SubExpr2 to be applicable.
- SubError1: Applying subtraction to values of incorrect types results in **Error** and panic trace.
- SubError2: A subtraction expression reduces to **Error** if its left-hand side reduces to **Error**.
- SubError3: A subtraction expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to SubExpr2, the SubError3 rule requires the left-hand side to already be a value.

Integer Multiplication

Integer multiplication is a binary operation that finds the product of 2 integer arguments.

$$\textsc{MulInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1 \texttt{ * } v_2 \rightsquigarrow [T]\ v_1 \times v_2} \qquad \textsc{MulExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1 \texttt{ * } n \rightsquigarrow [T_2]\ m_2 \texttt{ * } n} \qquad \textsc{MulExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v \texttt{ * } m_1 \rightsquigarrow [T_2]\ v \texttt{ * } m_2}$$

$$\textsc{MulError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1 \texttt{ * } v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{MulError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m \texttt{ * } n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{MulError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v \texttt{ * } m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- MulInt: Interpret *syntactic* v_1 * v_2 as *mathematical multiply* $v_1 \times v_2$ if applied to integers v_1 and v_2.
- MulExpr1: Structurally reduce the left-hand side of multiplication if it is not a value.
- MulExpr2: Structurally reduce the right-hand side of multiplication if it is not a value. This rule additionally requires the left-hand side to already be value. Basically, the arguments of multiplication must be evaluated from left to right in order for MulExpr2 to be applicable.
- MulError1: Applying multiplication to values of incorrect types results in **Error** and panic trace.
- MulError2: A multiplication expression reduces to **Error** if its left-hand side reduces to **Error**.
- MulError3: A multiplication expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to MulExpr2, the MulError3 rule requires the left-hand side to already be a value.

Integer Division

Integer division is a binary operation that finds the *truncated quotient* of 2 integer arguments. Truncated quotient (written as $\lfloor \frac{v_1}{v_2} \rfloor$) basically tells us how many times v_2 cleanly divides v_1.

$$\textsc{DivInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_2 \neq 0}{[T]\ v_1 \texttt{ / } v_2 \rightsquigarrow [T]\ \lfloor \frac{v_1}{v_2} \rfloor} \qquad \textsc{DivExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1 \texttt{ / } n \rightsquigarrow [T_2]\ m_2 \texttt{ / } n} \qquad \textsc{DivExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v \texttt{ / } m_1 \rightsquigarrow [T_2]\ v \texttt{ / } m_2}$$

$$\textsc{DivError0}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_2 = 0}{[T]\ v_1 \texttt{ / } v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{DivError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1 \texttt{ / } v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}}$$

$$\textsc{DivError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m \texttt{ / } n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{DivError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v \texttt{ / } m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- DivInt: Interpret *syntactic* v_1 / v_2 as *truncated quotient* $\lfloor \frac{v_1}{v_2} \rfloor$ if applied to integers v_1 and v_2 where $v_2 \neq 0$.

  Note: The `Div` command from the stack language performs truncated quotient.
- DivExpr1: Structurally reduce the left-hand side of division if it is not a value.
- DivExpr2: Structurally reduce the right-hand side of division if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of division must be evaluated from left to right in order for DivExpr2 to be applicable.
- DivError0: Applying division to integers v_1 and v_2 where $v_2 = 0$ results in **Error** and panic trace
- DivError1: Applying division to values of incorrect types results in **Error** and panic trace.
- DivError2: A division expression reduces to **Error** if its left-hand side reduces to **Error**.
- DivError3: A division expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to DivExpr2, the DivError3 rule requires the left-hand side to already be a value.

Integer Modulo

Integer modulo is a binary operation that finds the division remainder of 2 integer arguments. Due to the fact that there is no modulo command in the stack language, your compiler must implement integer modulo as a series of primitive commands supported by the stack language that produce the correct result and trace when evaluated.

$$\textsc{ModInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_2 \neq 0}{[\,T\,]\ v_1\ \texttt{mod}\ v_2 \rightsquigarrow [\,T\,]\ v_1 - v_2\lfloor\frac{v_1}{v_2}\rfloor} \qquad \textsc{ModExpr1}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ m_1\ \texttt{mod}\ n \rightsquigarrow [\,T_2\,]\ m_2\ \texttt{mod}\ n} \qquad \textsc{ModExpr2}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ v\ \texttt{mod}\ m_1 \rightsquigarrow [\,T_2\,]\ v\ \texttt{mod}\ m_2}$$

$$\textsc{ModError0}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_2 = 0}{[\,T\,]\ v_1\ \texttt{mod}\ v_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}} \qquad \textsc{ModError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[\,T\,]\ v_1\ \texttt{mod}\ v_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}}$$

$$\textsc{ModError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ m\ \texttt{mod}\ n \rightsquigarrow [\,T_2\,]\ \textbf{Error}} \qquad \textsc{ModError3}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ v\ \texttt{mod}\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- ModInt: Interpret *syntactic* v_1 `mod` v_2 as integer modulo $v_1 - v_2\lfloor\frac{v_1}{v_2}\rfloor$ if applied to integers v_1 and v_2 where $v_2 \neq 0$.
  Hint: Given a modulo expression m, transform it into an equivalent expression n comprised of only operators supported by the stack language. Now compile expression n stead instead of m.
- ModExpr1: Structurally reduce the left-hand side of modulo if it is not a value.
- ModExpr2: Structurally reduce the right-hand side of modulo if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of modulo must be evaluated from left to right in order for ModExpr2 to be applicable.
- ModError0: Applying modulo to integers v_1 and v_2 where $v_2 = 0$ results in **Error** and panic trace
- ModError1: Applying modulo to values of incorrect types results in **Error** and panic trace.
- ModError2: A modulo expression reduces to **Error** if its left-hand side reduces to **Error**.
- ModError3: A modulo expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to ModExpr2, the ModError3 rule requires the left-hand side to already be a value.

Boolean And

Boolean and is a binary operation that finds the conjunction of 2 boolean arguments.

$$\textsc{AndBool}\ \frac{v_1 \text{ and } v_2 \text{ are bools}}{[\,T\,]\ v_1\ \texttt{\&\&}\ v_2 \rightsquigarrow [\,T\,]\ v_1 \wedge v_2} \qquad \textsc{AndExpr1}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ m_1\ \texttt{\&\&}\ n \rightsquigarrow [\,T_2\,]\ m_2\ \texttt{\&\&}\ n} \qquad \textsc{AndExpr2}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ v\ \texttt{\&\&}\ m_1 \rightsquigarrow [\,T_2\,]\ v\ \texttt{\&\&}\ m_2}$$

$$\textsc{AndError1}\ \frac{v_1 \text{ or } v_2 \text{ are not bools}}{[\,T\,]\ v_1\ \texttt{\&\&}\ v_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}} \qquad \textsc{AndError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ m\ \texttt{\&\&}\ n \rightsquigarrow [\,T_2\,]\ \textbf{Error}} \qquad \textsc{AndError3}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ v\ \texttt{\&\&}\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- AndInt: Interpret *syntactic* v_1 `&&` v_2 as *conjunction* $v_1 \wedge v_2$ if applied to booleans v_1 and v_2.
- AndExpr1: Structurally reduce the left-hand side of and if it is not a value.
- AndExpr2: Structurally reduce the right-hand side of and if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of and must be evaluated from left to right in order for AndExpr2 to be applicable.
- AndError1: Applying and to values of incorrect types results in **Error** and panic trace.
- AndError2: An and expression reduces to **Error** if its left-hand side reduces to **Error**.
- AndError3: An and expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to AndExpr2, the AndError3 rule requires the left-hand side to already be a value.

Boolean Or

Boolean or is a binary operation that finds the disjunction of 2 boolean arguments.

$$\textsc{OrBool}\ \frac{v_1 \text{ and } v_2 \text{ are bools}}{[T]\ v_1\ \texttt{||}\ v_2 \rightsquigarrow [T]\ v_1 \vee v_2} \qquad \textsc{OrExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{||}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{||}\ n} \qquad \textsc{OrExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{||}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{||}\ m_2}$$

$$\textsc{OrError1}\ \frac{v_1 \text{ or } v_2 \text{ are not bools}}{[T]\ v_1\ \texttt{||}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{OrError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{||}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{OrError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{||}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- OrInt: Interpret *syntactic* v_1 `||` v_2 as *disjunction* $v_1 \vee v_2$ if applied to booleans v_1 and v_2.
- OrExpr1: Structurally reduce the left-hand side of or if it is not a value.
- OrExpr2: Structurally reduce the right-hand side of or if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of or must be evaluated from left to right in order for OrExpr2 to be applicable.
- OrError1: Applying or to values of incorrect types results in **Error** and panic trace.
- OrError2: An or expression reduces to **Error** if its left-hand side reduces to **Error**.
- OrError3: An or expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to OrExpr2, the OrError3 rule requires the left-hand side to already be a value.

Integer Less-Than

Integer less-than is a binary operation that compares the values of 2 integer arguments.

$$\textsc{LtInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1\ \texttt{<}\ v_2 \rightsquigarrow [T]\ v_1 < v_2} \qquad \textsc{LtExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{<}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{<}\ n} \qquad \textsc{LtExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{<}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{<}\ m_2}$$

$$\textsc{LtError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1\ \texttt{<}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{LtError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{<}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{LtError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{<}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- LtInt: Interpret *syntactic* v_1 `<` v_2 as the *less-than comparison* $v_1 < v_2$ if applied to integers v_1 and v_2.
- LtExpr1: Structurally reduce the left-hand side of less-than if it is not a value.
- LtExpr2: Structurally reduce the right-hand side of less-than if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of less-than must be evaluated from left to right in order for LtExpr2 to be applicable.
- LtError1: Applying less-than to values of incorrect types results in **Error** and panic trace.
- LtError2: A less-than expression reduces to **Error** if its left-hand side reduces to **Error**.
- LtError3: A less-than expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to LtExpr2, the LtError3 rule requires the left-hand side to already be a value.

Integer Greater-Than

Integer greater-than is a binary operation that compares the values of 2 integer arguments.

$$\textsc{GtInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1 \texttt{ > } v_2 \rightsquigarrow [T]\ v_1 > v_2} \qquad \textsc{GtExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1 \texttt{ > } n \rightsquigarrow [T_2]\ m_2 \texttt{ > } n} \qquad \textsc{GtExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v \texttt{ > } m_1 \rightsquigarrow [T_2]\ v \texttt{ > } m_2}$$

$$\textsc{GtError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1 \texttt{ > } v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{GtError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m \texttt{ > } n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{GtError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v \texttt{ > } m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- GtInt: Interpret *syntactic* v_1 `>` v_2 as the *greater-than comparison* $v_1 > v_2$ if applied to integers v_1 and v_2.
- GtExpr1: Structurally reduce the left-hand side of greater-than if it is not a value.
- GtExpr2: Structurally reduce the right-hand side of greater-than if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of greater-than must be evaluated from left to right in order for GtExpr2 to be applicable.
- GtError1: Applying greater-than to values of incorrect types results in **Error** and panic trace.
- GtError2: A greater-than expression reduces to **Error** if its left-hand side reduces to **Error**.
- GtError3: A greater-than expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to GtExpr2, the GtError3 rule requires the left-hand side to already be a value.

Integer Less-Than-Equal

Integer less-than-equal is a binary operation that compares the values of 2 integer arguments. Due to the fact that there is no less-than-equal command in the stack language, your compiler must implement less-than-equal as a series of primitive commands supported by the stack language that produce the correct result and trace when evaluated.

$$\textsc{LteInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1 \texttt{ <= } v_2 \rightsquigarrow [T]\ v_1 \leq v_2} \qquad \textsc{LteExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1 \texttt{ <= } n \rightsquigarrow [T_2]\ m_2 \texttt{ <= } n} \qquad \textsc{LteExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v \texttt{ <= } m_1 \rightsquigarrow [T_2]\ v \texttt{ <= } m_2}$$

$$\textsc{LteError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1 \texttt{ <= } v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{LteError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m \texttt{ <= } n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{LteError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v \texttt{ <= } m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- LteInt: Interpret *syntactic* v_1 `<=` v_2 as the *less-than-equal comparison* $v_1 \leq v_2$ if applied to integers v_1 and v_2.
- LteExpr1: Structurally reduce the left-hand side of less-than-equal if it is not a value.
- LteExpr2: Structurally reduce the right-hand side of less-than-equal if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of less-than-equal must be evaluated from left to right in order for LteExpr2 to be applicable.
- LteError1: Applying less-than-equal to values of incorrect types results in **Error** and panic trace.
- LteError2: A less-than-equal expression reduces to **Error** if its left-hand side reduces to **Error**.
- LteError3: A less-than-equal expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to LteExpr2, the LteError3 rule requires the left-hand side to already be a value.

Integer Greater-Than-Equal

Integer greater-than-equal is a binary operation that compares the values of 2 integer arguments. Due to the fact that there is no greater-than-equal command in the stack language, your compiler must implement greater-than-equal as a series of primitive commands supported by the stack language that produce the correct result and trace when evaluated.

$$\textsc{GteInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers}}{[T]\ v_1\ \texttt{>=}\ v_2 \rightsquigarrow [T]\ v_1 \geq v_2} \qquad \textsc{GteExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{>=}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{>=}\ n} \qquad \textsc{GteExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{>=}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{>=}\ m_2}$$

$$\textsc{GteError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1\ \texttt{>=}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}} \qquad \textsc{GteError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{>=}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{GteError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{>=}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- GteInt: Interpret *syntactic* v_1 `>=` v_2 as the *greater-than-equal comparison* $v_1 \geq v_2$ if applied to integers v_1 and v_2.
- GteExpr1: Structurally reduce the left-hand side of greater-than-equal if it is not a value.
- GteExpr2: Structurally reduce the right-hand side of greater-than-equal if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of greater-than-equal must be evaluated from left to right in order for GteExpr2 to be applicable.
- GteError1: Applying greater-than-equal to values of incorrect types results in **Error** and panic trace.
- GteError2: A greater-than-equal expression reduces to **Error** if its left-hand side reduces to **Error**.
- GteError3: A greater-than-equal expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to GteExpr2, the GteError3 rule requires the left-hand side to already be a value.

Integer Equality

Integer equality is a binary operation that compares the values of 2 integer arguments. Due to the fact that there is no equality command in the stack language, your compiler must implement equality as a series of primitive commands supported by the stack language. **Hint**: What relationship does equality have with `Lt` and `Gt`?

$$\textsc{EqInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_1 = v_2}{[T]\ v_1\ \texttt{=}\ v_2 \rightsquigarrow [T]\ \texttt{true}} \qquad \textsc{NeqInt}\ \frac{v_1 \text{ and } v_2 \text{ are integers} \qquad v_1 \neq v_2}{[T]\ v_1\ \texttt{=}\ v_2 \rightsquigarrow [T]\ \texttt{false}} \qquad \textsc{EqExpr1}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ m_1\ \texttt{=}\ n \rightsquigarrow [T_2]\ m_2\ \texttt{=}\ n}$$

$$\textsc{EqExpr2}\ \frac{[T_1]\ m_1 \rightsquigarrow [T_2]\ m_2}{[T_1]\ v\ \texttt{=}\ m_1 \rightsquigarrow [T_2]\ v\ \texttt{=}\ m_2} \qquad \textsc{EqError1}\ \frac{v_1 \text{ or } v_2 \text{ are not integers}}{[T]\ v_1\ \texttt{=}\ v_2 \rightsquigarrow [\texttt{"Panic"} :: T]\ \textbf{Error}}$$

$$\textsc{EqError2}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ m\ \texttt{=}\ n \rightsquigarrow [T_2]\ \textbf{Error}} \qquad \textsc{EqError3}\ \frac{[T_1]\ m \rightsquigarrow [T_2]\ \textbf{Error}}{[T_1]\ v\ \texttt{=}\ m \rightsquigarrow [T_2]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- EqInt: Reduce *syntactic* v_1 `=` v_2 to `true` if they are *mathematically equal* $v_1 = v_2$.
- NeqInt: Reduce *syntactic* v_1 `=` v_2 to `false` if they are *mathematically unequal* $v_1 \neq v_2$.
- EqExpr1: Structurally reduce the left-hand side of equality if it is not a value.
- EqExpr2: Structurally reduce the right-hand side of equality if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the arguments of equality must be evaluated from left to right in order for EqExpr2 to be applicable.
- EqError1: Applying equality to values of incorrect types results in **Error** and panic trace.
- EqError2: An equality expression reduces to **Error** if its left-hand side reduces to **Error**.
- EqError3: An equality expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to EqExpr2, the EqError3 rule requires the left-hand side to already be a value.

3.7 Expressions

Let-In Expression

Let-in expressions allow one to bind values to variables.

$$\textsc{LetVal}\ \frac{}{[\,T\,]\ \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ m \rightsquigarrow [\,T\,]\ m[v/x]} \qquad \textsc{LetExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ \texttt{let}\ x\ \texttt{=}\ m_1\ \texttt{in}\ n \rightsquigarrow [\,T_2\,]\ \texttt{let}\ x\ \texttt{=}\ m_2\ \texttt{in}\ n}$$

$$\textsc{LetError}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ \texttt{let}\ x\ \texttt{=}\ m\ \texttt{in}\ n \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- LetVal: A let-in expression reduces by substituting value v for variable x in expression m. The notation $m[v/x]$ denotes an expression obtained from replacing all instances of variable x in expression m with value v. You may assume that the let-in expressions produced by the `parse_prog` function each have uniquely named variable x.

  Hint: In the stack language, we can use commands `Bind` and `Lookup` to achieve the same effect as substitution.

- LetExpr: Structurally reduce the bound expression m_1 if it is not a value.

- LetError: A let-in expression reduces to **Error** if its bound expression m reduces to **Error**.

Function Application

Function application applies (possibly recursive) functions to value arguments.

$$\textsc{AppFun}\ \frac{}{[\,T\,]\ (\texttt{fun}\ f\ x\ \texttt{->}\ m)\ v \rightsquigarrow [\,T\,]\ m[(\texttt{fun}\ f\ x\ \texttt{->}\ m)/f, v/x]} \qquad \textsc{AppExpr1}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ m_1\ n \rightsquigarrow [\,T_2\,]\ m_2\ n} \qquad \textsc{AppExpr2}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ v\ m_1 \rightsquigarrow [\,T_2\,]\ v\ m_2}$$

$$\textsc{AppError1}\ \frac{v_1 \text{ is not of the form } (\texttt{fun}\ f\ x\ \texttt{->}\ m)}{[\,T\,]\ v_1\ v_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}} \qquad \textsc{AppError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ m\ n \rightsquigarrow [\,T_2\,]\ \textbf{Error}} \qquad \textsc{AppError3}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ v\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- AppFun: An applied function is reduced by substituting argument v for variable x in function body m. Furthermore, the function itself is substituted for variable f in m. The notation $m[(\texttt{fun}\ f\ x\ \texttt{->}\ m)/f, v/x]$ denotes an expression obtained from m by replacing all instances of variable x with value v and all instances of variable f with function $(\texttt{fun}\ f\ x\ \texttt{->}\ m)$. The substitution of f for $(\texttt{fun}\ f\ x\ \texttt{->}\ m)$ facilitates recursion by allowing the function to refer to itself through variable f.

  Hint: Given an arbitrary closure $\langle f, V_f, C\rangle$ in the stack language, the `Call` command will evaluate its local commands C using the extended local environment $f \rightarrowtail \langle f, V_f, C\rangle :: V_f$. Can the extension $f \rightarrowtail \langle f, V_f, C\rangle$ to the local environment be utilized to achieve the same effect of substituting f for $(\texttt{fun}\ f\ x\ \texttt{->}\ m)$?

- AppExpr1: Structurally reduce the left-hand side of an application expression if it is not a value.

- AppExpr2: Structurally reduce the right-hand side of an application expression if it is not a value. This rule additionally requires the left-hand side to already be a value. Basically, the sub-expressions of application expressions must be evaluated from left to right in order for AppExpr2 to be applicable.

- AppError1: Applying a non-function value to a value argument results in **Error** and panic trace.

- AppError2: An application expression reduces to **Error** if its left-hand side reduces to **Error**.

- AppError3: An application expression reduces to **Error** if its right-hand side reduces to **Error**. Similarly to AppExpr2, the AppError3 rule requires the left-hand side to already be a value.

Sequence Expression

The sequence expression allows one to compose 2 expressions together for sequential evaluation.

$$\textsc{SeqVal}\ \frac{}{[\,T\,]\ v;\ m \rightsquigarrow [\,T\,]\ m} \qquad \textsc{SeqExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ m_1;\ n \rightsquigarrow [\,T_2\,]\ m_2;\ n} \qquad \textsc{SeqError}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ m;\ n \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- SeqVal: Once the left-hand side of a sequence expression has been fully evaluated, evaluate the right-hand side. Notice that the value of the overall sequence expression is determined by the right-hand side expression m because v on the left-hand side is discarded.
- SeqExpr: Structurally evaluate the left-hand side of a sequence expression. While the left-hand side of a sequence expression will not impact the value of the overall expression, traces made by the left-hand side will persist even after the left-hand side is discarded by SeqVal.
- SeqError: A sequential expression reduces to **Error** if its left-hand side reduces to **Error**.

If-Then-Else Expression

If-then-else expression facilitates program branching on boolean conditions.

$$\textsc{IfteTrue}\ \frac{}{[\,T\,]\ \texttt{if true then}\ n_1\ \texttt{else}\ n_2 \rightsquigarrow [\,T\,]\ n_1} \qquad \textsc{IfteFalse}\ \frac{}{[\,T\,]\ \texttt{if false then}\ n_1\ \texttt{else}\ n_2 \rightsquigarrow [\,T\,]\ n_2}$$

$$\textsc{IfteExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ \texttt{if}\ m_1\ \texttt{then}\ n_1\ \texttt{else}\ n_2 \rightsquigarrow [\,T_2\,]\ \texttt{if}\ m_2\ \texttt{then}\ n_1\ \texttt{else}\ n_2} \qquad \textsc{IfteError1}\ \frac{v \text{ is not a bool}}{[\,T\,]\ \texttt{if}\ v\ \texttt{then}\ n_1\ \texttt{else}\ n_2 \rightsquigarrow [\,\texttt{"Panic"} :: T\,]\ \textbf{Error}}$$

$$\textsc{IfteError2}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ \texttt{if}\ m\ \texttt{then}\ n_1\ \texttt{else}\ n_2 \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- IfteTrue: An if-then-else expression reduces to its then-branch if its condition is `true`.
- IfteFalse: An if-then-else expression reduces to its else-branch if its condition is `false`.
- IfteExpr: Structurally evaluate the condition of an if-then-else expression.
- IfteError1: Attempting to branch on a non-boolean value results in **Error** and panic trace.
- IfteError2: An if-then-else expression reduces to **Error** if its condition reduces to **Error**.

Trace Expression

A trace expression logs the value of its argument to the program trace T.

$$\textsc{TraceVal}\ \frac{}{[\,T\,]\ \texttt{trace}\ v \rightsquigarrow [\,\mathit{toString}(v) :: T\,]\ \texttt{()}} \qquad \textsc{TraceExpr}\ \frac{[\,T_1\,]\ m_1 \rightsquigarrow [\,T_2\,]\ m_2}{[\,T_1\,]\ \texttt{trace}\ m_1 \rightsquigarrow [\,T_2\,]\ \texttt{trace}\ m_2} \qquad \textsc{TraceError}\ \frac{[\,T_1\,]\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}{[\,T_1\,]\ \texttt{trace}\ m \rightsquigarrow [\,T_2\,]\ \textbf{Error}}$$

The meaning behind these rules can be summarized as follows.

- TraceVal: If `trace` is applied to value v, prepend the string representation of v to T and reduce to unit value `()`.
- TraceExpr: Structurally evaluate the argument of `trace`.
- TraceError: A trace expression reduces to **Error** if its argument reduces to **Error**.

Given an arbitrary value v in the high-level language, the string obtained from $\mathit{toString}(v)$ is the same string representation of v's corresponding stack language value. The following equations illustrate the strings expected for typical inputs.

$$\mathit{toString}(123) = \texttt{"123"} \tag{1}$$
$$\mathit{toString}(\texttt{true}) = \texttt{"True"} \tag{2}$$
$$\mathit{toString}(\texttt{false}) = \texttt{"False"} \tag{3}$$
$$\mathit{toString}(\texttt{()}) = \texttt{"Unit"} \tag{4}$$
$$\mathit{toString}(\texttt{fun}\ \mathit{abc}\ x\ \texttt{->}\ m) = \texttt{"Fun<abc>"} \tag{5}$$

Notice that these equations are missing the symbol case of the stack language. This is because variables in the high-level language are always bound by let-expressions or function parameters and never appear in isolation. In fact, the `parse_prog` function is raise an `UnboundVariable` exception and fail to parse if this criteria is not met. Consider the high-level program `let x = 1 in trace x` where `trace` is applied to a let-bound variable. The expression reduces to `trace 1` through the LetVal rule which means that `trace` is no longer applied to a variable. The string `"1"` can now be successfully logged by the TraceVal rule.

4 Compilation

In this section we demonstrate how to compile a simple high-level program into the stack language. Consider the program (`trace` 1; 2) `-` (`trace true`; 3) in the high-level language. Informally, the following evaluation steps happen in order.

1. Begin evaluation of the left-hand side of subtraction expression (`trace` 1; 2) `-` (`trace true`; 3).
2. Begin evaluation of the left-hand side of sequence expression `trace` 1; 2.
3. Evaluation of `trace` 1 logs `"1"` to the trace and the sequence expression becomes `()`; 2.
4. Now that `()` is a value, `()`; 2 reduces to 2.
5. The left-hand side of 2 `-` (`trace true`; 3) is now an integer value, so evaluation of the right-hand side begins.
6. Begin evaluation of the left-hand side of sequence expression `trace true`; 3.
7. Evaluation of `trace true` logs `"True"` to the trace and the sequence expression becomes `()`; 3.
8. Now that `()` is a value, `()`; 3 reduces to 3.
9. Our subtraction expression is now 2 `-` 3 which can be reduced one last time to -1.

To capture the evaluation process of the high-level program, we maintain an *compilation invariant*: for a high-level expression, the value expected of its direct evaluation is always on top of the stack after interpreting its compiled version. We can see how this invariant comes into play during the compilation of the program given above. Compilation begins by recursively compiling the left and right and sides of the subtraction expression. So we now have 3 subtasks: first compile (`trace` 1; 2), next compile (`trace true`; 3), finally add their results together.

To compile (`trace` 1; 2), it is easy to see that this corresponds to the stack commands `Push` 1; `Trace`; `Pop`; `Push` 2;. Notice that `trace` 1 is compiled to `Push` 1; `Trace`; which logs `"1"` and puts `Unit` on the stack. Our compilation invariant holds as the value expected of directly evaluating `trace` 1 is on top of the stack. Next, the sequence operator (_; _), which is compiled to `Pop`, removes `Unit` from the stack. This implements the value discarding behavior of sequence expressions. Finally the constant 2 is compiled to `Push` 2. The resulting stack after executing `Push` 1; `Trace`; `Pop`; `Push` 2 is exactly 2 so our compilation invariant is maintained for the entire (`trace` 1; 2) expression.

Next, (`trace true`; 3) is compiled to `Push True`; `Trace`; `Pop`; `Push` 3;. Following the same reasoning as before, the invariant is maintained as the expected value 3 is on top of the stack after interpreting these commands.

Now that we have recursively compiled the sub-expressions of our original subtraction expression, we are left with the task of composing together the compiled commands and performing subtraction. Suppose we append the compiled commands together in the following way where the commands of (`trace` 1; 2) occur before the commands of (`trace true`; 3).

`Push` 1; `Trace`; `Pop`; `Push` 2; `Push True`; `Trace`; `Pop`; `Push` 3; (1)

Interpreting these stack commands results in stack $[3 :: 2 :: \epsilon]$ and trace $[$`"True"` $::$ `"1"` $:: \epsilon]$. In order for our compilation invariant to hold for all of (`trace` 1; 2) `-` (`trace true`; 3), the integer value -1 must be on top of the stack after evaluating its compiled commands. If we haphazardly append `Sub` to the end of (1), the resulting command list (2) will produce $[1 :: \epsilon]$ which violates the compilation invariant since -1 is not on top of the stack.

✗ `Push` 1; `Trace`; `Pop`; `Push` 2; `Push True`; `Trace`; `Pop`; `Push` 3; `Sub`; (2)

Suppose we append the compiled commands together in the following way where the commands of (`trace true`; 3) occur before the commands of (`trace` 1; 2). Interpreting these stack commands results in stack $[2 :: 3 :: \epsilon]$ and trace $[$`"1"` $::$ `"True"` $:: \epsilon]$. Although the stack is in the correct state for performing `Sub`, the order of strings in the trace is incorrect as it logs `"True"` before logging `"1"`.

✗ `Push True`; `Trace`; `Pop`; `Push` 3; `Push` 1; `Trace`; `Pop`; `Push` 2; (3)

The correct way to perform subtraction here is to insert a `Swap` command before the `Sub` command in (2). After the `Swap` command, the stack is $[2 :: 3 :: \epsilon]$ and the trace is $[$`"True"` $::$ `"1"` $:: \epsilon]$. At this point, performing `Sub` results in -1 on top of the stack so our invariant is preserved. Furthermore, the order of strings in the trace is correct.

✓ `Push` 1; `Trace`; `Pop`; `Push` 2; `Push True`; `Trace`; `Pop`; `Push` 3; `Swap`; `Sub`; (4)

5 Example Programs

In this section, we present some programs written in the concrete syntax of the high-level language along with their expected traces. Interpreting their compiled commands using the stack interpreter of part2 should yield the same trace.

Sequence of Traces

```
trace 1; trace 2
```

Expected Trace: ["2" :: "1" :: ϵ]

Factorial

```
let rec fact x =
  if x <= 0 then 1
  else x * fact (x - 1)
in trace (fact 10)
```

Expected Trace: ["3628800" :: ϵ]

Fibonacci

```
let fibo x =
  let rec loop i a b =
    trace a;
    if i < x then
      loop (i + 1) b (a + b)
    else a
  in loop 0 0 1
in trace (fibo 10)
```

Expected Trace: ["55" :: "55" :: "34" :: "21" :: "13" :: "8" :: "5" :: "3" :: "2" :: "1" :: "1" :: "0" :: ϵ]

Application of an Effectful Function

```
let eff x = trace x in
let foo x y z = () in
foo (eff 1) (eff 2) (eff 3)
```

Expected Trace: ["3" :: "2" :: "1" :: ϵ]

McCarthy's 91 Function

```
let rec mccarthy n =
  if n > 100 then n - 10
  else mccarthy (mccarthy (n + 11))
in
trace (mccarthy 22)
```

Expected Trace: ["91" :: ϵ]

Iterated Power Function

```
let rec iter n f g =
  if n <= 0 then g 0
  else f (iter (n - 1) f g)
in
let rec pow x = trace x; x * x in
iter 4 pow (fun _ -> 2)
```

Expected Trace: ["256" :: "16" :: "4" :: "2" :: ϵ]

Greatest Common Denominator

```
let rec gcd a b =
  if a = 0 then b
  else gcd (b mod a) a
in
trace (gcd 77 11);
trace (gcd 77 121);
trace (gcd 39 91)
```

Expected Trace: [”13” :: ”11” :: ”11” :: ϵ]

Square Root via Binary Search

```
let rec bsearch n i j =
  let k = (i + j) / 2  in
  if i > j then k
  else
    let sq = k * k in
    if sq = n then k
    else
      if n > sq
      then bsearch n (k + 1) j
      else bsearch n i (k - 1)
in
let rec sqrt n = bsearch n 0 n in
let x = 1234 * 1234 in
trace x;
trace (sqrt x)
```

Expected Trace: [”1234” :: ”1522756” :: ϵ]

Digits of π

```
let rec pi n =
  let q = 1 in
  let r = 180 in
  let t = 60 in
  let j = 2 in
  let rec loop n q r t j =
    if n > 0 then
      let u = 3 * (3 * j + 1) * (3 * j + 2) in
      let y = (q * (27 * j - 12) + 5 * r) / (5 * t) in
      trace y;
      let q' = 10 * q * j * (2 * j - 1) in
      let r' = 10 * u * (q * (5 * j - 2) + r - y * t) in
      let t' = t * u in
      let j' = j + 1 in
      loop (n - 1) q' r' t' j'
    else ()
  in
  loop n q r t j
in
pi 6
```

Expected Trace: [”9” :: ”5” :: ”1” :: ”4” :: ”1” :: ”3” :: ϵ]