## Syntax

We will adopt the following meta-variable conventions:

| | | | |
|---|---|---|---|
| $x$ | $\in$ | Var | variables |
| $b$ | $\in$ | $\{$`true, false`$\}$ | booleans |
| $n$ | $\in$ | $\mathbb{Z}$ | integers |
| $s$ | $\in$ | $\Sigma^*$ | ASCII strings |
| $\ell$ | $\in$ | Loc | memory locations |
| $p$ | $\in$ | Pat | patterns |
| $e$ | $\in$ | Exp | expressions |

The abstract syntax of **expressions** can be defined as follows, using auxiliary definitions for patterns $p$, unary operations $\odot$, binary operations $\oplus$, and types $\tau$ given below:

| | | | |
|---|---|---|---|
| $e \in \mathit{Exp}$ | $::=$ | $()$ | *Unit* |
| | $\mid$ | $b$ | *Booleans* |
| | $\mid$ | $n$ | *Integers* |
| | $\mid$ | $s$ | *Strings* |
| | $\mid$ | $x$ | *Variables* |
| | $\mid$ | $(e_1, e_2)$ | *Pairs* |
| | $\mid$ | $[\,]$ | *Empty list* |
| | $\mid$ | $e_1 :: e_2$ | *Non-empty lists* |
| | $\mid$ | $\odot\ e$ | *Unary operators* |
| | $\mid$ | $e_1 \oplus e_2$ | *Binary operators* |
| | $\mid$ | $e_1;\ e_2$ | *Sequential composition* |
| | $\mid$ | $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ | *Conditionals* |
| | $\mid$ | $\texttt{fun}\ (p : \tau) \to e$ | *Functions* |
| | $\mid$ | $\texttt{let}\ (p : \tau) = e_1\ \texttt{in}\ e_2$ | *Let expressions* |
| | $\mid$ | $\texttt{let rec}\ (f : \tau_1) = \texttt{fun}\ (p : \tau_2) \to e_1\ \texttt{in}\ e_2$ | *Recursive function expressions* |
| | $\mid$ | $e_1\ e_2$ | *Function application* |
| | $\mid$ | $\texttt{match}\ e_0\ \texttt{with}\ \mid p_1 \to e_1\ \ldots\ \mid p_n \to e_n\ \texttt{end}$ | *Pattern matching* |
| | $\mid$ | $\texttt{ref}\ e$ | *Reference creation* |
| | $\mid$ | $!e$ | *Dereference* |
| | $\mid$ | $e_1 := e_2$ | *Assignment* |

The syntax for patterns, unary operations, and binary operations is defined as follows:

| | | |
|---|---|---|
| $p \in \mathit{Pat}$ | $::=$ | $\_\_ \mid x \mid () \mid b \mid n \mid s \mid (p_1, p_2) \mid [\,] \mid p_1 :: p_2$ |
| $\odot \in \mathit{UOp}$ | $::=$ | `-` $\mid$ `not` |
| $\oplus \in \mathit{BOp}$ | $::=$ | `+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `&&` $\mid$ `\|\|` $\mid$ `<` $\mid$ `<=` $\mid$ `>` $\mid$ `>=` $\mid$ `=` $\mid$ `<>` $\mid$ `^` |

The set of **types** is defined as follows:

$$
\begin{array}{rcll}
\tau \in \mathit{Type} & ::= & \texttt{unit} & \textit{Unit Type} \\
 & | & \texttt{bool} & \textit{Boolean Type} \\
 & | & \texttt{int} & \textit{Integer Type} \\
 & | & \texttt{string} & \textit{String Type} \\
 & | & \tau_1 * \tau_2 & \textit{Pair Types} \\
 & | & \tau\ \texttt{list} & \textit{List Types} \\
 & | & \tau_1 \rightarrow \tau_2 & \textit{Function Types} \\
 & | & \tau\ \texttt{ref} & \textit{Reference Types} \\
 & | & \alpha & \textit{Type Variables}
\end{array}
$$

The set of **values** is defined as follows, using the auxiliary definition for environments $\mathcal{E}$, which is given below:

$$
\begin{array}{rcll}
v \in \mathit{Val} & ::= & () & \textit{Unit} \\
 & | & b & \textit{Boolean} \\
 & | & n & \textit{Integers} \\
 & | & s & \textit{Strings} \\
 & | & (v_1, v_2) & \textit{Pairs} \\
 & | & [\,] & \textit{Empty list} \\
 & | & v_1 :: v_2 & \textit{Non-empty lists} \\
 & | & (\!|\mathcal{E}, p, e|\!) & \textit{Closures} \\
 & | & \ell & \textit{Locations}
\end{array}
$$

**Environments** and **stores** are defined as partial functions from variables to values and from locations to values respectively:

$$
\begin{array}{lcl}
\mathcal{E} & \in & \mathrm{Var} \rightharpoonup \mathit{Val} \\
\sigma & \in & \mathrm{Loc} \rightharpoonup \mathit{Val}
\end{array}
$$

We use the following notation for describing environments:

- *dom* $\mathcal{E}$ denotes the domain of $\mathcal{E}$, that is the set of variables that it is defined on,
- $\{\}$ denotes the environment that is undefined on all variables,
- $\{x_1 \mapsto v_1, x_2 \mapsto v_2, ..., x_n \mapsto v_n\}$ denotes the environment that maps $x_1$ to $v_1$, $x_2$ to $v_2$, etc, through $x_n$ to $v_n$, and is otherwise undefined.
- $\mathcal{E}_2$ @ $\mathcal{E}_1$ denotes the environment that works as follows: if $x$ is in *dom* $\mathcal{E}_2$, then $(\mathcal{E}_2 \mathbin{@} \mathcal{E}_1)(x) = \mathcal{E}_2(x)$, if $x$ is in *dom* $\mathcal{E}_1$ but not *dom* $\mathcal{E}_2$, then $(\mathcal{E}_2 \mathbin{@} \mathcal{E}_1)(x) = \mathcal{E}_1(x)$, and otherwise, is undefined.

  Note: for those who are more math inclined, you may notice that the order of this is swapped compared to function composition, $\circ$. We choose the @ symbol and order to align more closely with OCaml.

We use the same notation for the analogous operations on stores $\sigma$.

## Semantics

We will use big-step semantics to define the behavior of RML. It will take in an environment $\mathcal{E}$ and expression $e$, and return a value $v$ and a new store $\sigma'$. Formally, we will define a relation,

$$\langle \mathcal{E}, \sigma, e \rangle \Downarrow \langle v, \sigma' \rangle$$

which can intuitively be read as follows: "under environment $\mathcal{E}$, and store $\sigma$, the expression $e$ evaluates to value $v$ and store $\sigma'$."

To define the big-step evaluation relation, we will use **inference rules**:

$$\textsc{E-Sequence}\ \frac{\langle \mathcal{E}, \sigma, e_1 \rangle \Downarrow \langle (), \sigma_1 \rangle \qquad \langle \mathcal{E}, \sigma_1, e_2 \rangle \Downarrow \langle v, \sigma_2 \rangle}{\langle \mathcal{E}, \sigma, e_1; e_2 \rangle \Downarrow \langle v, \sigma_2 \rangle}$$

Such rules are similar to the definitions we have seen in lecture, and we will read the bottom conclusion before the top premises. The **conclusion** below the line, such as $\langle \mathcal{E}, \sigma, e \rangle \Downarrow \langle v, \sigma_2 \rangle$, holds if all of the **premises** above the line, such as $\langle \mathcal{E}, \sigma, e_1 \rangle \Downarrow \langle (), \sigma_1 \rangle$ and $\langle \mathcal{E}, \sigma_1, e_2 \rangle \Downarrow \langle v, \sigma_2 \rangle$, also hold. For this rule, that means that if environment $\mathcal{E}$, store $\sigma$, and expression $e_1$ evaluate to the unit value and store $\sigma_1$; and environment $\mathcal{E}$, store $\sigma_1$, and expression $e_2$ evaluate to a value $v$ and store $\sigma_2$, then the entire expression $e_1; e_2$ evaluates to value $v$ and store $\sigma_2$ under environment $\mathcal{E}$ and store $\sigma$. In general, we have chosen to exclude the store output from most of the evaluation descriptions, since they are just dependent on passing through the evaluations in the premises.

### Simple Expressions

To warm up, let us consider the semantics of several simple expressions: values, pairs, lists, and variables.

$$\textsc{E-Value}\ \frac{}{\langle \mathcal{E}, \sigma, v \rangle \Downarrow \langle v, \sigma \rangle} \qquad \textsc{E-Var}\ \frac{\mathcal{E}(x) = v}{\langle \mathcal{E}, \sigma, x \rangle \Downarrow \langle v, \sigma \rangle}$$

$$\textsc{E-Pair}\ \frac{\begin{array}{c}\langle \mathcal{E}, \sigma, e_1 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \langle \mathcal{E}, \sigma_1, e_2 \rangle \Downarrow \langle v_2, \sigma_2 \rangle\end{array}}{\langle \mathcal{E}, \sigma, (e_1, e_2) \rangle \Downarrow \langle (v_1, v_2), \sigma_2 \rangle} \qquad \textsc{E-Cons}\ \frac{\begin{array}{c}\langle \mathcal{E}, \sigma, e_1 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \langle \mathcal{E}, \sigma_1, e_2 \rangle \Downarrow \langle v_2, \sigma_2 \rangle\end{array}}{\langle \mathcal{E}, \sigma, e_1 :: e_2 \rangle \Downarrow \langle v_1 :: v_2, \sigma_2 \rangle}$$

Intuitively, these inference rules can be understood as follows:

**E-Value:** An expression $v$ evaluates to a value $v$ if it is a unit, boolean, integer, string, or empty list.

**E-Var:** a variable $x$ evaluates to the value obtained by looking up $x$ in the environment $\mathcal{E}$.

E-Pair**:** a pair expression $(e_1, e_2)$ firsts evaluates $e_1$ to a value $v_1$, then $e_2$ to a value $v_2$, and then evaluates overall to a pair value $(v_1, v_2)$.

E-Cons**:** a cons expression $e_1 :: e_2$ evaluates to a non-empty list value $v_1 :: v_2$ by evaluating $e_1$ to a value $v_1$ and $e_2$ to a value $v_2$. Note that if $v_1$ has type $\alpha$, then $v_2$ must have type $\alpha$ list. This will follow from the typing rules given at the end of the document.

## Unary and Binary Operations

The next few rules model unary and binary operations.

$$\text{E-UOp}\ \frac{\langle \mathcal{E}, \sigma, e\rangle \Downarrow \langle v_1, \sigma'\rangle \qquad v = [\![\odot]\!]\ v_1}{\langle \mathcal{E}, \sigma, \odot\ e\rangle \Downarrow \langle v, \sigma'\rangle}$$

$$\text{E-Or-True}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{true}, \sigma'\rangle}{\langle \mathcal{E}, \sigma, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle \texttt{true}, \sigma'\rangle}$$

$$\text{E-Or-False}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{false}, \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, e_1\ \texttt{||}\ e_2\rangle \Downarrow \langle v, \sigma_2\rangle}$$

$$\text{E-And-False}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{false}, \sigma'\rangle}{\langle \mathcal{E}, \sigma, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle \texttt{false}, \sigma'\rangle}$$

$$\text{E-And-True}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{true}, \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, e_1\ \texttt{\&\&}\ e_2\rangle \Downarrow \langle v, \sigma_2\rangle}$$

$$\text{E-Other-BOp}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle v_1, \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v_2, \sigma_2\rangle \qquad v = [\![\oplus]\!]\ v_1\ v_2}{\langle \mathcal{E}, \sigma, e_1\ \oplus\ e_2\rangle \Downarrow \langle v, \sigma_2\rangle}$$

These inference rules can be understood as follows:

E-UOp**:** a unary operation $\odot\ e_1$ evaluates $e_1$ to a value $v_1$ and then uses the implementation of the operation, denoted $[\![\odot]\!]$, to produce the final value $v$. Note that implementations may require the value $v_1$ to have a specific type—e.g., unary negation is only defined on integers.

E-Or-True **and** E-Or-False : an or-expression $e_1$ `||` $e_2$ first evaluates $e_1$ to a boolean, and then either returns `true` right away or evaluates and returns $e_2$. Note that it if $e_1$ is `true`, it does *not* evaluate $e_2$. Also, $v_2$ must be a boolean.

E-And-False **and** E-And-True : an and-expression $e_1$ `&&` $e_2$ first evaluates $e_1$ to a boolean, and then either returns `false` right away or evaluates and returns $e_2$. Note that it if $e_1$ is `false`, it does *not* evaluate $e_2$. Also, $v_2$ must be a boolean.

E-Other-BOp**:** This applies to all other binary operations except `||` and `&&`. It is similar to the case for unary operations. We also implicitly require that the values $v_1$ and $v_2$ are of the expected type for the operation $\oplus$; for instance, if the addition is addition, then $v_1$ and $v_2$ must be integer values (and not, say, booleans or strings). The equality (`=`) and inequality (`<>`) operators are only defined when the two sides have the same, *base* type: `bool`, `int`, or `string`.

## Standard Control-Flow Expressions

The next few rules model standard control-flow expressions.

$$\textsc{E-Sequence}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle (), \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, e_1; e_2\rangle \Downarrow \langle v, \sigma_2\rangle}$$

$$\textsc{E-If-True}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{true}, \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \Downarrow \langle v, \sigma_2\rangle}$$

$$\textsc{E-If-False}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \texttt{false}, \sigma_1\rangle \qquad \langle \mathcal{E}, \sigma_1, e_3\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \Downarrow \langle v, \sigma_2\rangle}$$

These inference rules can be understood as follows:

E-Sequence**:** a sequential composition $e_1;\ e_2$ evaluates $e_1$ to unit () and then evaluates $e_2$ to a value $v$.

E-If-True **and** E-If-False : a conditional `if` $e_1$ `then` $e_2$ `else` $e_3$ first evaluates $e_1$ to a boolean, and then either evaluates $e_2$ or $e_3$ depending on the value of $e_1$. Note however that it does *not* evaluate both $e_2$ and $e_3$.

## Pattern Matching Expressions

To model pattern matching, we will use a three-place relation of the form $v : p \rightsquigarrow \mathcal{E}$, read as "value $v$ matches pattern $p$ and produces the bindings in $\mathcal{E}$."

$$\textsc{M-Wild}\ \frac{}{v : \_\_ \rightsquigarrow \{\}} \qquad \textsc{M-Var}\ \frac{}{v : x \rightsquigarrow \{x \mapsto v\}} \qquad \textsc{M-Unit}\ \frac{}{() : () \rightsquigarrow \{\}}$$

$$\textsc{M-Bool}\ \frac{}{b : b \rightsquigarrow \{\}} \qquad \textsc{M-Int}\ \frac{}{n : n \rightsquigarrow \{\}} \qquad \textsc{M-String}\ \frac{}{s : s \rightsquigarrow \{\}}$$

$$\textsc{M-EmptyList}\ \frac{}{[\,] : [\,] \rightsquigarrow \{\}} \qquad \textsc{M-Pair}\ \frac{v_1 : p_1 \rightsquigarrow \mathcal{E}_1 \qquad v_2 : p_2 \rightsquigarrow \mathcal{E}_2}{(v_1, v_2) : (p_1, p_2) \rightsquigarrow \mathcal{E}_2\ @\ \mathcal{E}_1}$$

$$\textsc{M-Cons}\ \frac{v_1 : p_1 \rightsquigarrow \mathcal{E}_1 \qquad v_2 : p_2 \rightsquigarrow \mathcal{E}_2}{v_1 :: v_2 : p_1 :: p_2 \rightsquigarrow \mathcal{E}_2\ @\ \mathcal{E}_1}$$

These pattern-matching rules can be understood as follows:

M-Wild: any value $v$ matches the wildcard pattern and produces an empty environment.

M-Var: any value $v$ matches the variable $x$ pattern and produces an environment that binds $x$ to $v$.

M-Unit: the unit value matches the unit pattern and produces an empty environment.

M-Bool: a boolean value $b$ matches against the pattern containing the same boolean $b$ to produce the empty environment.

M-Int: an integer value $n$ matches against the pattern containing the same integer $n$ to produce the empty environment.

M-String: a string value $s$ matches against the pattern containing the same string $s$ to produce the empty environment.

M-EmptyList: an empty list value matches against the empty list pattern to produce the empty environment.

M-Pair: a pair value $(v_1, v_2)$ matches with a pair pattern $(p_1, p_2)$ by matching value $v_1$ to a pattern $p_1$ to produce an environment $\mathcal{E}_1$, then matching value $v_2$ to a pattern $p_2$ to produce an environment $\mathcal{E}_2$, then appending $\mathcal{E}_2$ and $\mathcal{E}_1$.

M-Cons: a list-cons value $v_1 :: v_2$ matches with a list-cons pattern $p_1 :: p_2$ by matching value $v_1$ to a pattern $p_1$ to produce an environment $\mathcal{E}_1$, then matching value $v_2$ to a pattern $p_2$ to produce an environment $\mathcal{E}_2$, then appending $\mathcal{E}_2$ and $\mathcal{E}_1$.

The evaluation rule for match expressions is as follows.

$$\textsc{E-Match}\ \frac{\begin{array}{c}\langle \mathcal{E}, \sigma, e\rangle \Downarrow \langle v, \sigma_e\rangle \\ \text{for some } 1 \le j \le n,\ \ v : p_j \rightsquigarrow \mathcal{E}_j \text{ and } v : p_i \not\rightsquigarrow \mathcal{E}_i \text{ for } i < j \\ \langle \mathcal{E}_j \mathbin{@} \mathcal{E}, \sigma_e, e_j\rangle \Downarrow \langle v_j, \sigma'\rangle\end{array}}{\langle \mathcal{E}, \sigma, \texttt{match}\ e\ \texttt{with} \mid p_1 \to e_1\ \ldots\ \mid p_n \to e_n\ \texttt{end}\rangle \Downarrow \langle v_j, \sigma'\rangle}$$

This inference rule can be understood as follows:

E-Match: This inference rule evaluates $e$ to a value $v$, finds the first pattern $p_j$ that matches $v$, and then evaluates the corresponding expression $e_j$ in an environment extended with the bindings from $v$ obtained using $p_j$.

## Functions, Definitions, and Application Expressions

The next few inference rules handle functions, `let`-definitions, and application expressions.

$$\textsc{E-Fun}\ \frac{}{\langle \mathcal{E}, \sigma, \texttt{fun}\ (p : \tau) \rightarrow e\rangle \Downarrow \langle (\!|\mathcal{E}, p, e|\!), \sigma\rangle}$$

$$\textsc{E-App}\ \frac{\begin{array}{c}\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle (\!|\mathcal{E}_{cl}, p_{cl}, e_{cl}|\!), \sigma_1\rangle \\ \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v_2, \sigma_2\rangle \quad v_2 : p_{cl} \rightsquigarrow \mathcal{E}_2 \quad \langle \mathcal{E}_2\ @\ \mathcal{E}_{cl}, \sigma_2, e_{cl}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \mathcal{E}, \sigma, e_1\ e_2\rangle \Downarrow \langle v, \sigma'\rangle}$$

$$\textsc{E-Let}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle v_1, \sigma_1\rangle \quad v_1 : p \rightsquigarrow \mathcal{E}_1 \quad \langle \mathcal{E}_1\ @\ \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle}{\langle \mathcal{E}, \sigma, \texttt{let}\ (p : \tau) = e_1\ \texttt{in}\ e_2\rangle \Downarrow \langle v, \sigma_2\rangle}$$

$$\textsc{E-LetRec}\ \frac{\mathcal{E}_f = \mathcal{E}[f \mapsto (\!|\mathcal{E}_f, p, e_f|\!)] \quad \langle \mathcal{E}_f, \sigma, e_2\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathcal{E}, \sigma, \texttt{let rec}\ (f : \tau_1) = \texttt{fun}\ (p : \tau_2) \rightarrow e_f\ \texttt{in}\ e_2\rangle \Downarrow \langle v, \sigma'\rangle}$$

These inference rules can be understood as follows:

E-Fun**:** a function `fun` $p \rightarrow e$ evaluates to a closure.

E-App**:** an application $e_1\ e_2$ evaluates $e_1$ to a closure $(\!|\mathcal{E}, p, e|\!)$, evaluates $e_2$ to a value $v_2$, matches $v_2$ against the pattern $p$, and finally evaluates the body of the closure $e$.

E-LetRec**:** is similar to the case for `let`-definitions. First it evaluates $e_1$ to an anonymous function (using the rec keyword doesn't make sense for non-functions). Then instead of pattern matching to create new bindings, it builds a recursive environment $\mathcal{E}_f$ in which $f$ is bound to the closure for the function with parameter $p$ and body $e$. Then it sets this environment as the environment in the closure that $f$ is bound to, and evaluates $e_2$ with $\mathcal{E}_f$

## Imperative Expressions

The next few inference rules model OCaml-style references:

$$\textsc{E-Ref}\ \frac{\langle \mathcal{E}, \sigma, e\rangle \Downarrow \langle v, \sigma_e\rangle \quad \ell \notin dom\ \sigma_e \quad \sigma' = \{\ell \mapsto v\}\ @\ \sigma_e}{\langle \mathcal{E}, \sigma, \texttt{ref}\ e\rangle \Downarrow \langle \ell, \sigma'\rangle}$$

$$\textsc{E-Deref}\ \frac{\langle \mathcal{E}, \sigma, e\rangle \Downarrow \langle \ell, \sigma'\rangle \quad v = \sigma'(\ell)}{\langle \mathcal{E}, \sigma, !e\rangle \Downarrow \langle v, \sigma'\rangle}$$

$$\textsc{E-Assign}\ \frac{\langle \mathcal{E}, \sigma, e_1\rangle \Downarrow \langle \ell, \sigma_1\rangle \quad \langle \mathcal{E}, \sigma_1, e_2\rangle \Downarrow \langle v, \sigma_2\rangle \quad \sigma' = \{\ell \mapsto v\}\ @\ \sigma_2}{\langle \mathcal{E}, \sigma, e_1 := e_2\rangle \Downarrow \langle (), \sigma'\rangle}$$

These inference rules can be understood as follows:

E-Ref: a reference `ref` $e$ evaluates $e$ to a value $v$ and then adds it to the store $\sigma$ under a fresh location $\ell$. The location $\ell$ is returned along with the new store $\sigma'$, which is the old store with the new one added.

E-Deref: a dereference $!e_1$ evaluates $e_1$ to a location $\ell$ and looks it up in the store $\sigma$.

E-Assign: an assignment $e_1 := e_2$ evaluates $e_1$ to a location $\ell$ and $e_2$ to a value, updates the store $\sigma$ so that $\ell$ maps to $v_2$, and returns () along with the updated store.

## Definitions and Programs

A complete RML program is defined as a non-empty list of definitions:

$$\begin{array}{lcll} \mathit{prog} \in \mathit{Prog} & ::= & d & \textit{Single definition} \\ & \mid & d\ \mathit{prog} & \textit{Definition followed by more definitions} \end{array}$$

A definition is either a let-definition or a let-rec-definition:

$$\begin{array}{lcll} d \in \mathit{Defn} & ::= & \texttt{let}\ (p : \tau) = e & \textit{Let definition} \\ & \mid & \texttt{let rec}\ (f : \tau_1 \to \tau_2) = \texttt{fun}\ (p : \tau_1) \to e & \textit{Let rec definition} \end{array}$$

The relation $\langle \mathcal{E}, \sigma, d\rangle \Downarrow \langle \mathcal{E}', \sigma'\rangle$, which can be read "under environment $\mathcal{E}$, the definition $d$ evaluates to the updated environment $\mathcal{E}'$" and updated store $\sigma'$, is given as follows:

$$\text{E-DLet}\ \frac{\langle \mathcal{E}, \sigma, e\rangle \Downarrow \langle v, \sigma'\rangle \qquad v : p \rightsquigarrow \mathcal{E}'}{\langle \mathcal{E}, \sigma, \texttt{let}\ (p : \tau) = e\rangle \Downarrow \langle \mathcal{E}' \mathbin{@} \mathcal{E}, \sigma'\rangle}$$

$$\text{E-DLetRec}\ \frac{\mathcal{E}_f = \mathcal{E}[f \mapsto (|\mathcal{E}_f, p, e_f|)]}{\langle \mathcal{E}, \sigma, \texttt{let rec}\ (f : \tau) = \texttt{fun}\ (p : \tau_2) \to e_f\rangle \Downarrow \langle \mathcal{E}_f, \sigma\rangle}$$

Program evaluation threads the initial environment, denoted $\mathcal{E}_0$, and the initial store, denoted $\sigma_0$, through the definitions, accumulating more new mappings each time:

$$\text{E-Prog}\ \frac{\begin{array}{c}\langle \mathcal{E}_0, \sigma_0, d_1\rangle \Downarrow \langle \mathcal{E}_1, \sigma_1\rangle \\ \langle \mathcal{E}_1, \sigma_1, d_2\rangle \Downarrow \langle \mathcal{E}_2, \sigma_2\rangle \qquad \ldots \qquad \langle \mathcal{E}_{n-1}, \sigma_{n-1}, d_n\rangle \Downarrow \langle \mathcal{E}_n, \sigma_n\rangle\end{array}}{d_1\ \ldots\ d_n \Downarrow \langle \mathcal{E}_n, \sigma_n\rangle}$$

## Type System

As discussed in the assignment writeup, the behavior of the interpreter is undefined if the program in question does not pass the type checker. The type system of RML is very similar to that of OCaml, and should not be hard to reason about informally. However, we include the following typing rules for those who wish to reason more carefully about what programs are considered valid in RML. It is not required to read or understand the typing rules, and you may feel free to skip them if you wish.

For the typing judgement, we will need to define the notion of a *typing context*, a partial map from variable names to types. We will use the capital Greek letter $\Gamma$ for contexts:

$$\Gamma : Var \rightharpoonup \mathit{Type}$$

We adopt the following notational conventions for describing typing contexts:

- $\mathit{dom}\ \Gamma$ denotes the domain of $\Gamma$, that is the set of variable that it is defined on,
- $\emptyset$ denotes the typing context that is undefined on all variables,
- $\Gamma, x : \tau$ denotes the typing context that maps $x$ to $\tau$ and all other $y \in \mathit{dom}$ $\Gamma$ to $\Gamma(y)$.
- $\Gamma_1, \Gamma_2$ denotes the typing context that maps $x$ in $\mathit{dom}\ \Gamma_2$ to $\Gamma_2(x)$, $x$ in $\mathit{dom}\ \Gamma_1$ but not in $\mathit{dom}\ \Gamma_2$ to $\Gamma_1(x)$, and is otherwise undefined. (Yes, this is the opposite order of the @ symbol from environments and stores earlier in the document.)

First we will give the auxiliary definition for typing patterns, denoted $p : \tau \rightsquigarrow \Gamma$. This can be read "pattern $p$ is of type $\tau$ and produces typing context $\Gamma$". In describing these the typing rules for RML, we will use notation similar to what is used to describe the big-step semantics.

$$\frac{}{\_ : \tau \rightsquigarrow \emptyset} \qquad \frac{}{() : \texttt{unit} \rightsquigarrow \emptyset} \qquad \frac{}{b : \texttt{bool} \rightsquigarrow \emptyset} \qquad \frac{}{n : \texttt{int} \rightsquigarrow \emptyset}$$

$$\frac{}{s : \texttt{string} \rightsquigarrow \emptyset} \qquad \frac{}{x : \tau \rightsquigarrow \{x : \tau\}} \qquad \frac{}{[\,] : \tau\ \texttt{list} \rightsquigarrow \emptyset}$$

$$\frac{p_1 : \tau_1 \rightsquigarrow \Gamma_1 \qquad p_2 : \tau_2 \rightsquigarrow \Gamma_2}{(p_1, p_2) : \tau_1 * \tau_2 \rightsquigarrow \Gamma_1, \Gamma_2} \qquad \frac{p_1 : \tau \rightsquigarrow \Gamma_1 \qquad p_2 : \tau\ \texttt{list} \rightsquigarrow \Gamma_2}{p_1 :: p_2 : \tau\ \texttt{list} \rightsquigarrow \Gamma_1, \Gamma_2}$$

The typing judgement for expressions will be written $\Gamma \vdash e : \tau$, and can be read "$\Gamma$ shows that $e$ is of type $\tau$". Some rules involve type substitutions $S$, which are partial maps from type variables to types. We write $S(\tau)$ for the result of applying substitution $S$ to $\tau$.

$$\frac{}{\Gamma \vdash () : \texttt{unit}} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash s : \texttt{string}}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma(x) = \forall(\tau_1 \to \tau_2)}{\Gamma \vdash x : S(\tau_1) \to S(\tau_2)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{}{\Gamma \vdash [\,] : \tau\ \texttt{list}} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \texttt{list}}$$

$$\frac{p : \tau \rightsquigarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e : \tau'}{\Gamma \vdash \texttt{fun}\ (p : \tau) \to e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad p : \tau \rightsquigarrow \Gamma' \qquad \Gamma, \Gamma' \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let}\ (p : \tau) = e_1\ \texttt{in}\ e_2 : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \forall(\tau_1 \to \tau_2) \vdash e_2 : \tau' \qquad e_1 \text{ is value}}{\Gamma \vdash \texttt{let}\ (x : \tau_1 \to \tau_2) = e_1\ \texttt{in}\ e_2 : \tau'}$$

$$\frac{p : \tau_1 \rightsquigarrow \Gamma' \qquad \Gamma, f : \tau_1 \to \tau_2, \Gamma' \vdash e_1 : \tau_2 \qquad \Gamma, f : \forall(\tau_1 \to \tau_2) \vdash e_2 : \tau_3}{\Gamma \vdash \texttt{let rec}\ (f : \tau_1 \to \tau_2) = \texttt{fun}\ (p : \tau_1) \to e_1\ \texttt{in}\ e_2 : \tau_3}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}\oplus \in \{\texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \oplus \in \{\texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \oplus \in \{\texttt{<>}, \texttt{=}\}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{string} \qquad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\texttt{\^{}}e_2 : \texttt{string}} \qquad \frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 : \tau_1 \\ p_1 : \tau_1 \rightsquigarrow \Gamma_1 \qquad \Gamma, \Gamma_1 \vdash e_1 : \tau_2 \qquad \ldots \qquad p_n : \tau_1 \rightsquigarrow \Gamma_n \qquad \Gamma, \Gamma_n \vdash e_n : \tau_2 \end{array}}{\Gamma \vdash \texttt{match}\ e_0\ \texttt{with | }p_1 \to e_1\ \ldots\ \texttt{| }p_n \to e_n\ \texttt{end} : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}\ e : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash \texttt{!}e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ \texttt{:=}\ e_2 : \texttt{unit}}$$