Topic 1. Abstract syntax of D. The abstract syntax of D is obtained by extending the expr datatype with forms for conditionals, variables, functions, and Let:

```
type ident = Ident of string

type expr =
  Bool of bool
| ...
| If of expr * expr * expr
| Var of ident
| Let of ident * expr * expr
| Function of ident * expr
| Appl of expr * expr
| Fix of ident * ident * expr (* to be discussed *)
```

The concrete-to-abstract syntax encoding is appropriately extended:

```
[True] = Bool(true)
[False] = Bool(false)

[x] = Var(Ident("x"))
[If e Then e1 Else e2] = If([e], [e1], [e2])
[e1,e2] = Appl([e1], [e2])
[(Function x → e)] = Function(Ident("x"), [e])
[Let x = e1 In e2] = Let(Ident("x"), [e1], [e2])
```

Example:

```
[(Function x → If x = 0 Then x Else Minus(x,1))3] =
Appl(Function(Ident("x"), If(Equal(Var(Ident("x"))),
Int(0)),Var(Ident("x")),Minus(Var(Ident("x")),Int(1)))), Int(3))
```

Topic 2. Recursion. The construct Fix is introduced to encode recursion. In the specification (note that these are functions, hence values):

```
v ::= ··· | (Fix z.x → e) values (extended)
```

We require in any expression (Fix z.x → e) that z \neq x.

Definition 1.1 (Scope (extended)) The scope of z and x in (Fix z.x → e) is e.

Substitution is thus implicitly extended as follows:

```
((Fix z.y → e))[v/x] = (Fix z.x → e)          x = z or x = y
((Fix z.y → e))[v/x] = (Fix z.y → e[v/x])    x \neq z and x \neq y
```
Since Fix expressions are values (as recursive functions), they evaluate to themselves. The necessary
modification to the D semantics comprises an additional rule form for application, to accomodate the case
when a recursive function is applied.

\[
\begin{align*}
\text{APPL} & \quad \frac{e_1 \Rightarrow (\text{Fix } z.x \rightarrow e) \quad e_2 \Rightarrow v'}{e_1 e_2 \Rightarrow (\text{Fix } z.x \rightarrow e)[v'/x] \Rightarrow v}
\end{align*}
\]

For example, omitting all the VAL rule instances for brevity, and letting:

\[
f \triangleq ((\text{Fix } \text{sumt}.x \rightarrow \text{If } x = 0 \text{ Then } 0 \text{ Else } x + \text{sumt}(x - 1)))
\]

\[
\begin{align*}
0 = 0 & \Rightarrow \text{True} \\
1 - 1 & \Rightarrow 0 \quad \text{If } 0 = 0 \text{ Then } 0 \text{ Else } 0 + f(0 - 1) \Rightarrow 0 \\
f(1 - 1) & \\
\vdots & \\
5 - 1 & \Rightarrow 4 \quad \text{If } 4 = 0 \text{ Then } 0 \text{ Else } 4 + f(4 - 1) \Rightarrow 10 \\
5 = 0 & \Rightarrow \text{False} \quad \text{If } 5 = 0 \text{ Then } 0 \text{ Else } 5 + f(5 - 1) \Rightarrow 15 \\
f5 & \Rightarrow 15
\end{align*}
\]

The concrete-to-abstract syntax encoding is appropriately extended to accommodate recursive functions:

\[
[(\text{Fix } z.x \rightarrow e)] = \text{Fix}(\text{Ident}("z"), \text{Ident}("x"), [e])
\]

**Topic 3. Compound types: the language DP.** As demonstrated in examples for hw102, it is possible
to encode pairs in D with functional trickery, but it is more efficient and appealing from a programming
perspective to include them directly in the language. We therefore extend our language with operations for
constructing and deconstructing pairs. In general, pairs are very useful datatypes, since they can be used to
define other datatypes such as n-tuples and lists.

\[
v ::= \cdots | (v,v) \quad \text{values}
\]

\[
e ::= \cdots | (e,e) \mid \text{left}(e) \mid \text{right}(e) \quad \text{expressions}
\]

The operational rules are straightforward; note that our definition of values and the PAIR rule specifies an
eager evaluation of pairs (this will show up in an example, below):

\[
\begin{align*}
\text{PAIR} & \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{(e_1,e_2) \Rightarrow (v_1,v_2)} \\
\text{LEFT} & \quad \frac{e \Rightarrow (v,v')}{\text{left}(e) \Rightarrow v} \\
\text{RIGHT} & \quad \frac{e \Rightarrow (v',v)}{\text{right}(e) \Rightarrow v}
\end{align*}
\]

example:

\[
\begin{align*}
\text{add} & \triangleq (\text{Function } x \rightarrow \text{left}(x) + \text{right}(x)) \\
1 \Rightarrow 1 & \\
2 \Rightarrow 2 \quad 3 \Rightarrow 3 \quad 2 + 3 = 5 & \\
(1, 2 + 3) \Rightarrow (1, 5) & \\
(1, 5) \Rightarrow (1, 5) & (1, 5) \Rightarrow (1, 5) \\
\text{left}(1, 5) \Rightarrow 1 & \text{right}(1, 5) \Rightarrow 5 \\
\text{add} \Rightarrow \text{add} & \\
(1, 2 + 3) \Rightarrow (1, 5) & \text{left}(1, 5) + \text{right}(1, 5) \Rightarrow 6 \\
\text{add}(1, 2 + 3) \Rightarrow 6
\end{align*}
\]
To implement pairs, we follow the familiar pattern; we extend our abstract syntax definition to include pairs and projection:

```ml
type expr =
  ...
  | Pr of expr * expr
  | Left of expr
  | Right of expr
```

We extend our definition of concrete-to-abstract expression transformations:

\[
\begin{align*}
[(e_1, e_2)] &= Pr([e_1], [e_2]) \\
[left(e)] &= Left([e]) \\
[right(e)] &= Right([e])
\end{align*}
\]

We then extend `eval` to implement the operational semantics for these language forms.

```ml
let rec eval e =
  match e with
  ...
  | Pr(e1, e2) -> Pr(eval e1, eval e2)
  | Left(e) -> (match eval e with
    Pr(e1, e2) -> e1
    | _ -> raise TypeMismatch)
  | Right(e) -> (match eval e with
    Pr(e1, e2) -> e2
    | _ -> raise TypeMismatch)
```