Assignment 4

CAS CS 320: Concepts of Programming Languages

This assignment is due on **Thursday 2/26 by 8:00PM**. You should put all of your programming solutions in the file `assign4/lib/assign4.ml`. See the file `test/test_assign4.ml` for example behavior of each function.

1 Programming (100%)

This week we're be building calculator (of sorts). Instead of (just) evaluating arithmetic expressions, we'll be *checking* typing derivations of typing judgments over arithmetic expressions. The expression language we'll be using is a very mild extension of the toy calculator language from Section 2.3 in the course notes. Let's start by stating the syntax, typing, and semantic rules for the language.

Language Specification

Syntax

$$\begin{aligned}
\texttt{<ty>} &::= \texttt{int} \mid \texttt{bool} \\
\texttt{<op>} &::= \texttt{+} \mid \texttt{*} \mid \texttt{=} \\
\texttt{<expr>} &::= \texttt{<int>} \mid \texttt{(<op> <expr> <expr>)} \mid \texttt{(if <expr> <expr> <expr>)}
\end{aligned}$$

Typing

$$\frac{\text{n is an integer literal}}{\texttt{n} : \texttt{int}}\ \textsc{IntLit} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(+ } e_1\ e_2\ \texttt{)} : \texttt{int}}\ \textsc{AddInt} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\texttt{(* } e_1\ e_2\ \texttt{)} : \texttt{int}}\ \textsc{MulInt}$$

$$\frac{e_1 : \tau \qquad e_2 : \tau}{\texttt{(= } e_1\ e_2\ \texttt{)} : \texttt{bool}}\ \textsc{Eq} \qquad \frac{e_1 : \texttt{bool} \qquad e_2 : \tau \qquad e_3 : \tau}{\texttt{(if } e_1\ e_2\ e_2\ \texttt{)} : \texttt{bool}}\ \textsc{Eq}$$

Semantics

$$\frac{\text{n is an integer literal for } n}{\texttt{n} \Downarrow n}\ \textsc{IntLitE} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 + v_2}{\texttt{(+ } e_1\ e_2\ \texttt{)} \Downarrow v}\ \textsc{AddIntE}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 \times v_2}{\texttt{(* } e_1\ e_2\ \texttt{)} \Downarrow v}\ \textsc{MulIntE} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 = v_2}{\texttt{(= } e_1\ e_2\ \texttt{)} \Downarrow \top}\ \textsc{EqTrue}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\texttt{(= } e_1\ e_2\ \texttt{)} \Downarrow \bot}\ \textsc{EqFalse} \qquad \frac{e_1 \Downarrow \top \qquad e_2 \Downarrow v_2}{\texttt{(if } e_1\ e_2\ e_3\ \texttt{)} \Downarrow v_2}\ \textsc{IfTrue} \qquad \frac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v_3}{\texttt{(if } e_1\ e_2\ e_3\ \texttt{)} \Downarrow v_3}\ \textsc{IfFalse}$$

1.1 Parsing Arithmetic Expressions

The syntax above indicates that the expressions we'll be considering are written using S-expressions. With this syntax, operators appear *before* their arguments, and must be parenthesized. This is typical of Lisp dialects. S-expressions are represented abstractly by the following algebraic data type (ADT).[1]

```
type sexpr =
  | Atom of string
  | List of sexpr list
```

So, for example, the S-expression `(a (bc d) ((efg)) h)` is represented by the following value.

```
List
  [
    Atom "a";
    List [Atom "bc"; Atom "d"];
    List [List [Atom "efg"]];
    Atom "h";
  ]
```

Arithmetic expressions are represented abstractly by the following ADTs.

```
type op = Add | Mul | Eq

type expr =
  | Int of int
  | Bop of op * expr * expr
  | If of expr * expr * expr
```

So the arithmetic expression $(1 + 2) * 3 + 4$ is represented by the following value.

```
Bop
( Add
, Bop
  ( Mul
  , Bop (Add, Int 1, Int 2)
  , Int 3
  )
, Int 4
)
```

[1] S-expressions will be the topic of this weeks lab, which is particularly important to attend if you want a more in-depth introduction to S-expressions.

This same expression would be written as an S-expression as `(+ (* (+ 1 2) 3) 4)`, and would be represented as an `sexpr` as follows.

```
List
  [
    Atom "+";
    List
      [
        Atom "*";
        List [Atom "+"; Atom "1"; Atom "2"];
        Atom "3";
      ];
    Atom "4"
  ]
```

Task

We've provided a general S-expressions parser for you. Our first task is to convert between the two representations, and bootstrap this to convert between arithmetic expressions and strings. This means implementing the following functions.

```
val expr_of_sexpr_opt : sexpr -> expr option
val expr_of_string_opt : string -> expr option
val sexpr_of_expr : expr -> sexpr
val string_of_expr : expr -> string
```

Note that not all S-expression represent valid arithmetic expressions, but all arithmetic expressions can be converted back into S-expressions. The same asymmetry holds for string and arithmetic expressions.

1.2 Parsing Derivations

Derivations are trees of judgments. Therefore, like arithmetic expressions (which are trees of operators), typing derivations are naturally representing using ADTs (and records).

```
type ty = BoolT | IntT

type ty_jmt = {expr : expr; ty : ty}

type ty_rule = Int_lit | Add_int | Mul_int | Eq_rule | If_rule

type ty_deriv =
  | Rule_app of {
      prem_derivs : ty_deriv list;
      concl : ty_jmt;
      rname : ty_rule;
    }
  | Hole
```

We'll say more about the `Hole` variant in a moment, but note that `Rule_app` carries exactly the data we need to represent one step of a derivation: the conclusion of the applied rule (`concl`), a list of derivations of the premises of the applied rule (`prem_derivs`), and the name of the rule applied (`rname`).

Also like arithmetic expressions,[2] we can represent derivations via S-expressions. Here's the syntax that we'll use.

```
<rname>    ::= INTLIT | ADDINT | MULINT | EQ | IF
<ty_deriv> ::= ( <expr> <ty> <rname> <deriv> ... <deriv> ) | ???
```

So, for example, the following is a derivation of the judgment `(+ 1 2) : int`.

```
((+ 1 2) int ADDINT
    (1 int INTLIT)
    (2 int INTLIT))
```

We would typically draw this as follows.

$$\dfrac{\dfrac{}{\texttt{1} : \texttt{int}}\ \textsc{IntLit} \qquad \dfrac{}{\texttt{2} : \texttt{int}}\ \textsc{IntLit}}{\texttt{(+ 1 2)} : \texttt{int}}\ \textsc{AddInt}$$

Note that our S-expression-representation of derivations is "upside down" compared to how we typically draw them.

Holes

For longer expressions, it's difficult to build an entire derivation in one go. It's easier if we can build the derivation incrementally. This is the purpose of the `Hole` variant. If we want to write a *partial* derivation, we add a `Hole` to our derivation (written as `???` in our derivation syntax) to be filled in later. For example, here is a partial derivation of the judgment `(* (+ 1 2) (+ 2 3)) : int`.

```
((* (+ 1 2) (+ 2 3)) int
    ((+ 1 2) int ADDINT
        ???
        (2 int INTLIT))
    ???)
```

Note that we can have multiple holes in one derivation.

Task

Implement the function that converts an S-expression into a derivation, which used to then convert strings to derivations.

```
val ty_deriv_of_sexpr_opt : sexpr -> ty_deriv option
val ty_deriv_of_string_opt : string -> ty_deriv option
```

Make sure to take a look at the helper functions we provide in the starter code.

[2] And really any tree-like hierarchical data.

1.3 Checking Derivations

Our abstract representation of derivations does not guarantee that a `ty_deriv` is valid, e.g., there's nothing stopping us from building the following.

```
let bad_deriv =
  Rule_app
    {
      prem_derivs = [];
      concl = { expr = Int 1; ty = Bool };
      rname = Int_lit;
    }
```

We need to write functions that check whether or not a given `ty_deriv` is valid, i.e., that all rules have been applied correctly. This is made slightly more complicated by the fact that *there may be holes* in the derivation. For example, we should except:

```
((if (= 0 1) 1 2) int IF
    ???
    (1 int INTLIT)
    ???)
```

but not:

```
((if (= 0 1) 1 2) int IF
    (1 int INTLIT)
    ???
    ???)
```

or:

```
((if (= 0 1) 1 2) int IF
    (1 int INTLIT)
    ???)
```

Furthermore, we should except:

```
((if (= 0 1) 1 2) bool IF
    ???
    ???
    ???)
```

even though it's not possible to complete the derivation, because there's no information presently available in the derivation that's contradictory.

Task

You should implement functions that will check that individual rule applications are valid. These should be combined to implement:

```
val check_rule : ty_rule -> ty_jmt option list -> ty_jmt -> bool
```

so that `check_rule rname premises concl` is `true` if the conclusion is consistent with `premises` according to the rule `rname`, and `false` otherwise. The premises are options because they may appear as holes in a derivation. You should then implement a function that checks the status of an entire derivation.

```
type status =
  | Complete
  | Partial
  | Invalid

val check_deriv : ty_deriv -> status
```

Note that there are three possible cases, the derivation is `Complete`, it has holes but is consistent (i.e., is `Partial`), or it has an invalid rule application, with or without holes (i.e., is `Invalid`).

1.4 Evaluation

At the end of the day, we still want to know the value of the expression that we're typing. This is the easy part now that we've represented arithmetic expressions as ADTs. Also notice how values are distinct from expressions, i.e., values are a different ADT.

1.4.1 Task

You need to implement the function

```
type value = BoolV of bool | IntV of int

val value_of_expr : expr -> value =
```

so that `value_of_expr e` is the value of `e` according to the language specification above. The behavior of the function is undefined in the case that the expression does not have a value.[3]

1.5 Writing a derivation

Once you've completed the above tasks, you should be able to use the following command to check whether a derivation in a given file is valid.

```
dune exec assign4 [filename]
```

[3] Again, the whole point of typing is that, because we have a typing derivation for the expression, it's guaranteed to have a value.

Task

Write derivation in our derivation syntax for the following typing judgment:

$$\texttt{(if (= (= 5 (+ 1 4)) (= 0 1)) (+ 2 3) (* (+ 4 5) 67))} : \texttt{int}$$

How I would recommend doing this: start with the following in a file called something like `example.d`.

```
((if (= (= 5 (+ 1 4)) (= 0 1)) (+ 2 3) (* (+ 4 5) 67)) int IF
    ???
    ???
    ???)
```

and run the following command.

```
dune exec assign4 example.d
```

Rinse and repeat, replacing each hole (`???`) with a partial derivation , e.g.,

```
((if (= (= 5 (+ 1 4)) (= 0 1)) (+ 2 3) (* (+ 4 5) 67)) int IF
    ???
    ((+ 2 3) int ADDINT
        ???
        ???)
    ???)
```

until the entire derivation has been filled in. Then remember to replace the value of `example_derivation` in `assign4.ml` with the entire derivation (represented as a string).

Happy Coding!