HW 9 **Prof. Caldwell**
COSC 4780

1 An implementation of classes and type structures

In this assignment you will complete the interpreter (code provided) to support classes and records. You can study the code for details of the implementation. We point out some significant features in the discussion here.

1.1 Representing the type structures in ML

Recall the following types used in the specification of classes and types structures.

$$\begin{array}{lcll}
\theta & ::= & \tau \mid \tau exp \mid comm \mid \delta \mid \delta class \mid \pi dec & \\
\delta & ::= & intloc \mid \pi & \\
\pi & ::= & \{i : \theta_i\}_{i \in I} & \text{where} \quad I \subseteq Identifier \;\; I \text{ finite} \\
\tau & ::= & int \mid bool &
\end{array}$$

This is implemented in ML using a mutually recursive type as follows:

```
type types =
    Int
  | Bool
  | Intexp
  | Boolexp
  | Command
  | Type of delta
  | Class of delta
  | Dec of type_assignment
and delta =
    Intloc
  | Pi of type_assignment
and  type_assignment =
    TA of ((string * types) list)
;;
```

1.2 Environments

Recall the semantics of the various types:

name	*syntax classes*	*type of the meaning*
integers	$\theta, \tau,$	$[\![int]\!] : \mathbb{Z}$
Booleans	θ, τ	$[\![bool]\!] : \mathbb{B}$
int expressions	$\theta, \tau exp$	$[\![intexp]\!] : Store \to \mathbb{Z}$
bool expressions	$\theta, \tau exp,$	$[\![boolexp]\!] : Store \to \mathbb{B}$
commands	θ	$[\![comm]\!] : Store \to Store_\perp$
locations	θ, δ	$[\![intloc]\!] : Location$
type assignments	θ, δ, π	$[\![\{i : \theta_i\}_{i\in I}]\!] = \{i : [\![\theta_i]\!]\}_{i\in I}$
class	θ	$[\![\delta class]\!] : Store \to ([\![\delta]\!] \times Store)$

Environments are used to bind identifiers to the meanings of the syntactic phrases they are bound to. Since different types have different meanings associated with them we use an ML record to keep track of all of the environments. Environments are represented as ML functions of the correct type.

```
type env = { int : string -> int;
             bool : string -> bool;
             intexp : string -> store -> int;
             boolexp: string -> store -> bool;
             command: string -> store -> store;
             intloc : string -> loc;
             pi : string -> env;
             intloc_class : string -> store -> (loc * store);
             pi_class : string -> store -> (env * store)
};;
```

For each of function in the environment we need a separate function to update that part of the environment. Here's one of them:

```
let update_boolexp_env (i, v) e =
  {int = e.int;
   bool = e.bool;
   intexp = e.intexp;
   boolexp = update (i,v) e.boolexp;
   command = e.command;
   intloc = e.intloc;
```

```
    pi = e.pi;
    intloc_class = e.intloc_class;
    pi_class = e.pi_class
   }
;;
```

1.3 Identifiers and their meanings

We add the following ML type to represent the syntactic class of identifier-expressions.

```
type identifier = Id of string | XId of identifier * string ;;
```

A useful operation on an identifier is to get the last part.

```
let lastId id =
  match id with
      Id i -> i
    | XId (id,i) -> i
;;
```

Recall the semantics of the identifier-expressions:

$$
\begin{array}{l}
[\![\pi \vdash I : \theta]\!]\, e = v \quad \text{where } \{I : v\} \in e \\
[\![\pi \vdash X.I : \theta]\!]\, e = v \\
\quad \text{where } \{I : v\} \in r \text{ and } r = [\![\pi \vdash X : \pi_1\, class]\!]\, e
\end{array}
$$

We implement this in ML by first finding the environment to use. If the identifier is of the form `Id x` then we use the current environment. If the identifier is complex (of the form `XId(id,i)`) then we recursively find the environment for `id` and use that to look up `i`. The following function finds the correct environment to use.

```
let rec get_lookup_env id pi env err =
  match id with
      Id i -> env
    | XId (id,i) ->
      (match (type_of_identifier id pi) with
           Type (Pi pi') ->
             (get_lookup_env
                id (bar_union_ta pi' pi) env err).pi (lastId id)
         | _ -> raise (Failure
                 ("meaning_of_" ^ err ^
                     ": (-> get_lookup_env) - bad record id.")))
;;
```

If the identifier is complex, XId(id,i) then we compute the type of the identifier using the current type assignment pi. The pi-environment is used to hold the environments that map record fields to their meanings; thus, the *only* possible type a for a well-typed record identifier id is Type(Pi pi') where pi' is the type assignment for the record. From there we recursively get the lookup environment for id using an updated type assignment $\pi' \uplus \pi$. Since this function is used to compute the meanings for all types of identifiers, we pass a string err so that a useful error message can be printed if a lookup fails.

Now, for each kind of environment we implement a meaning function for identifiers of that type which uses get_lookup_env to find the environment to lookup in. Here a a few of those functions; there is one for each instance of the rule (where θ is replaced by a specific type.).
Here's the rule for when θ is the type int.

$$
\begin{array}{l}
[\![\pi \vdash I : \mathit{int}]\!]\, e = k \quad \text{where } \{I : k\} \in e \\
[\![\pi \vdash X.I : \mathit{int}]\!]\, e = k \\
\quad \text{where } \{I : k\} \in r \text{ and } r = [\![\pi \vdash X : \pi_1\, \mathit{class}]\!]\, e
\end{array}
$$

And here is the corresponding ML code.

```
let meaning_of_int_id id pi env =
  let env' = get_lookup_env id pi env "int_id" in
    env'.int (lastId id)
;;
```

The code is uniform – all the implementations are essentially identical; though they differ in part of the environment they invoke and hence the type of functions they return. Here are two more.

```
let meaning_of_pi_id id pi env =
  let env' = get_lookup_env id pi env "pi_id" in
```

```
    env'.pi (lastId id)
;;

let meaning_of_intloc_class_id id pi env =
  let env' = get_lookup_env id pi env "intloc_class_id" in
    env'.intloc_class (lastId id)
;;
```

1.4 Type Structures and Declarations

Recall that the syntax for declarations and type structures defined mutually recursively: var and class declarations are defined in terms of type structures and the record type structure is defined using a declaration.

$$\begin{array}{lcl} D & ::= & D_1, D_2 \mid D_1; D_2 \mid \textbf{fun}\ I{=}E \mid \textbf{const}\ I{=}N \mid \mathsf{proc}\ I = C \\ & & \quad \mid \textbf{var}\ I{:}T \mid \textbf{class}\ I{=}T \\ T & ::= & \mathsf{newint} \mid \textbf{record}\ D\ \mathsf{end} \mid X \end{array}$$

Here is an ML implementation of this syntax class:

```
type declaration =
    Var of string * type_structure
  | TClass of string * type_structure
  | Const of string * expression
  | Fun of string * expression
  | Proc of string * command
  | Comma of declaration * declaration
  | Semi of declaration * declaration
and type_structure =
    Newint
  | Record of declaration
  | X of identifier
;;
```

It is almost always the case that when a type is defined mutually recursively, the functions the operate on those types are also mutually recursive. Following the semantic equations that define the meaning of a declarations and those defining the meaning of type structures we implement the meaning functions mutually recursively. Since the meaning of a type structure is has type ($[\![\delta]\!] \times \mathit{Store}$) and $[\![\delta]\!] = \{\mathit{intloc}, \pi\}$, we will make two meaning functions for type structures: `meaning_of_newint` for the *intloc* case and `meaning_of_record` for the π case.

```
let rec meaning_of_declaration d pi env s  =
  match d with
      Var (id, ts) ->
        if is_intloc_class ts pi then
         [ your code here]
        else if is_pi_class ts pi then
         [ your code here]
         else
          raise (Failure "meaning_of_declaration: bad class.")
    | TClass(name,ts) ->
        if is_intloc_class ts pi then
         [ your code here]
        else if is_pi_class ts pi then
         [ your code here]
        else
          raise (Failure "meaning_of_declaration: bad class.")
    | Const (id, e) -> ...
    | Fun (id, e) -> ...
    | Proc (id, c) -> ...
    | Semi (d1,d2) ->  ...
    | Comma (d1,d2) ->  ...

and meaning_of_newint t pi env s =
  match t with
      Newint -> allocate s
    | Record _  -> raise (Failure "meaning_of_newint: records not intloc class.")
    | X id -> meaning_of_intloc_class_id id pi env s

and meaning_of_record t pi env s =
    match t with
        Newint -> raise (Failure "meaning_of_record: newint not record class.")
      | Record d ->  meaning_of_declaration d pi env s
      | X id -> meaning_of_pi_class_id id pi env s
;;
```

You need to implement the following semantic rules for the *var* and *class* declarations. I have named the environment to add the binding to in the left column.

$$\begin{array}{rl}
\textit{intloc env} & [\![\pi \vdash \textbf{var}\ I{:}T : \{I : \textit{intloc}\}\ \textit{dec}]\!]\, e\, s = \langle \{I : l\}, s' \rangle \\
& \quad \text{where } \langle l, s' \rangle = [\![\pi \vdash T : \textit{intloc}\,\textit{class}]\!]\, e\, s \\[1ex]
\textit{pi env} & [\![\pi \vdash \textbf{var}\ I{:}T : \{I : \pi_1\}\ \textit{dec}]\!]\, e\, s = \langle \{I : e\}, s' \rangle \\
& \quad \text{where } \langle e, s' \rangle = [\![\pi \vdash T : \pi_1\, \textit{class}]\!]\, e\, s \\[1ex]
\textit{intloc class env} & [\![\pi \vdash \textbf{class}\ I{=}T : \{I : \textit{intloc}\,\textit{class}\}\textit{dec}]\!]\, e\, s = \langle \{I : f\}, s \rangle \\
& \quad \text{where } f\, t = [\![\pi \vdash T : \textit{intloc}\,\textit{class}]\!]\, e\, t \\[1ex]
\textit{pi class env} & [\![\pi \vdash \textbf{class}\ I{=}T : \{I : \pi_1\, \textit{class}\}\textit{dec}]\!]\, e\, s = \langle \{I : f\}, s \rangle \\
& \quad \text{where } f\, t\ = [\![\pi \vdash T : \pi_1\, \textit{class}]\!]\, e\, t
\end{array}$$

Note that the type structures for *intloc* or *intloc class* environments are `newint` or identifier expressions. Note that the type structures for π or $\pi\, \textit{class}$ environments are records or identifier expressions.