CS 496: Makeup Homework Assignment 7
Due: 2 May 2018, 11:55pm

1 Assignment Policies

Collaboration Policy. This assignment may be solved in groups of up to two students.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

Makeup Policy. This assignment is optional, you are not required to hand it in. If you do, it will help contribute towards your grade. It has two parts: interpreter and type-checker. The interpreter is worth 100 points (i.e. full marks); the type-checker is extra-credit (extra 50 points). If you correctly implement both parts you obtain 150 points. If you obtain a grade lower than your current overall average for assignments, then it will not be computed. Hence there is no reason to give it a try!

2 Assignment

This is an optional makeup assignment. In this assignment you are asked to extend the interpreter for EXPLICIT-REFS to support *Algebraic Data Types*. The new language is called ADT. You shall be given the grammar for this language as part of the stub; you have to concentrate on two things:

- Implement the interpreter.
- Implement the type-checker.

3 The ADT

The new grammar is:

<Expression> ::= ...
| <Identifier> < {<Expression>,* }>
| case <Expression> of { {<Branch> }$^{+}$ }
| type <Identifier> = { | <Identifier> ({ <Type> },*)}*

<Branch> | <Identifier> < {<Identifier>},* > -> <Expression>

<Type> ::= ...
| <Identifier>

Examples of programs:

```
Node<3,Nil<>,Nil<>>

type treeInt =
   | Nil<>
   | Node<int,treeInt,treeInt>

let getRoot =
   proc (t:treeInt) {
     case t of {
       Nil -> error
       Node<x,y,z> -> x
   } }
in (getRoot (Node<1,Nil,Nil>))

begin
   type treeInt =
     | Nil
     | Node<int,treeInt,treeInt>;
   let isEmpty=
    proc(t:treeInt) {
       case t of {
         Nil -> zero?(0)
         Node<x,y,z> -> zero?(1)
       } }
   in (isEmpty (Node<1,Nil,Nil>))
 end


begin
   type treeInt =
     | Nil
     | Node<int,treeInt,treeInt>;
   letrec ((int -> int) -> treeInt) mapT(t:treeInt) =
        proc (f:(int -> int)) {
          case t of {
            Nil -> Nil
            Node<x,y,z> ->
                 Node<(f x), ((mapT y) f), ((mapT z) f)>
 } }
    in ((mapT (Node<2,Nil,Nil>)) (proc(x:int) {x+1}))
end
```

Here is an example of the AST produced by one of the above mentioned examples:

```
utop # parse "
let getRoot =
   proc (t:treeInt) {
     case t of {
       Nil -> error
       Node<x,y,z> -> x
   } }
in (getRoot (Node<1,Nil,Nil>))";;


AProg
 (Let ("getRoot",
   Proc ("t", UserType "treeInt",
    Case (Var "t",
     [Branch ("Nil", [], Var "error");
      Branch ("Node", ["x"; "y"; "z"], Var "x")])),
   App (Var "getRoot",
    Variant ("Node", [Int 1; Variant ("Nil", []); Variant ("Nil", [])]))))
```

4 The Interpreter

Implement the interpreter. The new run-time values are *tagged-variant values*, sometimes also called tagged-union values:

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | UnitVal
  | RefVal of int
  | TaggedVariantVal of string*exp_val list
```

Regarding the evaluation of the new expressions, first note that the `type` construct has no run-time behavior and hence may be ignored by the interpreter (returning `UnitVal`). The other two productions of the new grammar will require your attention.

Examples of program evaluation are:

```
utop # interp " Node<3,Nil<>,Nil<>>";;
- : exp_val =
TaggedVariantVal ("Node",
 [NumVal 3; TaggedVariantVal ("Nil", []); TaggedVariantVal ("Nil", [])])

interp "
type treeInt =
   | Nil<>
   | Node<int,treeInt,treeInt>";;
- : exp_val = UnitVal

utop # interp "
begin
   type treeInt =
     | Nil
     | Node<int,treeInt,treeInt>;
   letrec ((int -> int) -> treeInt) mapT(t:treeInt) =
        proc (f:(int -> int)) {
          case t of {
```

```
            Nil -> Nil
            Node<x,y,z> ->
                 Node<(f x), ((mapT y) f), ((mapT z) f)>
 } }
    in ((mapT (Node<2,Nil,Nil>)) (proc(x:int) {x+1}))
end";;
- : exp_val =
TaggedVariantVal ("Node",
 [NumVal 3; TaggedVariantVal ("Nil", []); TaggedVariantVal ("Nil", [])])
```

5 The Type-Checker

Implement the type-checker. Below are the typing rules that you should use as a guideline. We first introduce some notation that will be used in the typing rules.

5.1 Preliminaries

We use the letter Σ to denote an environment of type declarations and call it a *type declaration environment*. Each type declaration in Σ is represented as a pair (id, cs) where id is the name of the user defined type and cs is the list of all its constructors together with their types. For example, Σ could be:

$$\begin{array}{l}\{(\texttt{treeInt}, [(\texttt{Nil}, []); (\texttt{Node}, [\texttt{int}; \texttt{treeInt}; \texttt{treeInt}])]),\\ \;(\texttt{daysOfWeek}, [(\texttt{Monday}, []); \ldots; (\texttt{Sunday}, [])])\}\end{array} .$$

The expression $\Sigma(\texttt{treeInt})$ denotes a function that returns the types of the arguments of the constructors. For example, $\Sigma(\texttt{treeInt})(\texttt{Node}) = [\texttt{int}; \texttt{treeInt}; \texttt{treeInt}]$.

We assume each constructor is unique. We write $\Sigma^{-1}(\texttt{C})$ for the type of constructor `C`. For example, $\Sigma^{-1}(\texttt{Node}) = \texttt{treeInt}$.

5.2 Typing rules

Below we assume that Σ contains the type declarations of the entire program that is to be typed. Thus typing judgements take the form (notice the new component Σ):

$$\texttt{tenv} \vdash_{\Sigma} \texttt{e} :: \texttt{t}$$

where `tenv` is a type environment, `e` is an expression, `t` is a type and Σ is a type declaration environment. In your implementation, you would store them in an additional argument to `type_of_expr`, namely a hash table that maps type names to their declarations. Every time you see a declaration, you would then add it to the table.

$$\frac{\Sigma^{-1}(\texttt{C}) = \texttt{t} \quad \Sigma(\texttt{t})(\texttt{C}) = (\texttt{t1}, \ldots, \texttt{tm}) \quad \texttt{m} = \texttt{n} \quad \texttt{tenv} \vdash_{\Sigma} \texttt{ei} :: \texttt{ti}}{\texttt{tenv} \vdash_{\Sigma} \texttt{C<e1,...,en>} :: \texttt{t}} \; \texttt{T-Constructor}$$

$$\frac{\texttt{tenv} \vdash_{\Sigma} \texttt{e} :: \texttt{t} \quad \Sigma(\texttt{t}) = \{\texttt{C1}, \ldots, \texttt{Cm}\} \quad \forall \texttt{i} \in 1 \ldots \texttt{m}.\ \text{if } \Sigma(\texttt{t})(\texttt{Ci}) = \vec{\texttt{ti}} \text{ then } [\vec{\texttt{xi}} \texttt{:=} \vec{\texttt{ti}}]\texttt{tenv} \vdash_{\Sigma} \texttt{ei} :: \texttt{s}}{\texttt{tenv} \vdash_{\Sigma} \texttt{case e of } \{ \texttt{ C1<}\vec{\texttt{x1}}\texttt{> -> e1 ... Cm<}\vec{\texttt{xm}}\texttt{> -> em } \} :: \texttt{s}} \; \texttt{T-Case}$$

6 Submission instructions

Submit a file named `HW6_<SURNAME>.zip` through Canvas which includes all the source files required to run the interpreter and type-checker. Please include the names of the members of the team in the file `top.scm`.