Problem Set 8: Types

Problem 1

We can define the Scheme syntax of the simply typed lambda calculus with one built-in type \texttt{Int} using the following BNF specification:

\[
\begin{align*}
<\text{Exp}> & ::= 
\langle \text{Identifier} \rangle \ <\text{Type}> \\
& \ | \ ((\lambda \ <\text{Identifier} > \ <\text{Type}> \ <\text{Exp}> ) \ <\text{Exp}> \ <\text{Type}> ) \\
& \ | \ ((<\text{Exp}> \ <\text{Exp}> ) \ <\text{Type}> )
\end{align*}
\]

\[
\begin{align*}
<\text{Type}> & ::= \ \text{Int} \\
& \ | \ (\langle \text{Type} \rangle \rightarrow \langle \text{Type} \rangle )
\end{align*}
\]

The meta-symbol \(\langle \text{Identifier} \rangle\) stands for a Scheme symbol that is different from the symbol \texttt{lambda}.

We can think of every typed lambda calculus term as a pair of a value (of the untyped lambda calculus) and a tag to indicate its type, written \((\text{exp type})\).

- Using the \texttt{define-datatype} abstraction, define the abstract syntax of data types \texttt{Exp} and \texttt{Type} according to the BNF specification given above. Use as variant names the name given in the boxes.
- Define the procedure \((\text{parse-exp lst})\) that when applied to a list representation of a typed lambda calculus expression will return its abstract syntax tree. Apply the principle “Follow the Grammar”. That is, you need to define a procedure for every syntactic category.
- Given a typed lambda calculus expression \(e \in <\text{Exp}>\) define the predicate \texttt{type-annotations-consistent?}, which when applied to \(e\) returns \#t if all
type annotations in e are consistent. Otherwise, the predicate returns #f. You have to implement a case analysis using the cases statement:

- e == (v t): #t
- e == ((lambda (x t1) b) t2):
  t2 is a function type &&
  t1 == argument type of t2 &&
  (type-annotations-consistent? b) &&
  type of b == result type of t2
- e == ((ftn arg) t):
  (type-annotations-consistent? ftn) &&
  (type-annotations-consistent? arg) &&
  type of ftn is a function type &&
  argument type of type of ftn == type of arg &&
  t == result type of type of ftn

Examples:

```lisp
> (define var1 '(y ((Int -> (Int -> Int)) -> (Int -> Int))) )
> (define abs1 '((lambda (x (Int -> Int)) (x (Int -> Int)))
   ((Int -> Int) -> (Int -> Int))) )

> (define appl1 '(((lambda (x (Int -> Int)) (x (Int -> Int)))
   ((Int -> Int) -> (Int -> Int)))
   (y (Int -> Int)))
   (Int -> Int) )

> (define applEl '(((x (Int -> Int)) (y (Int -> Int))) Int) )

> (parse-exp var1)
(t-id y
   (t-ftn
      (t-ftn (t-int) (t-ftn (t-int) (t-int)))
      (t-ftn (t-int) (t-int))))

> (parse-exp abs1)
(t-abs
   x
   (t-ftn (t-int) (t-int))
   (t-id x (t-ftn (t-int) (t-int)))
   (t-ftn (t-ftn (t-int) (t-int)) (t-ftn (t-int) (t-int))))

> (parse-exp appl1)
(t-app
   (t-abs
      x
      (t-ftn (t-int) (t-int))
      (t-id x (t-ftn (t-int) (t-int)))
      (t-ftn (t-ftn (t-int) (t-int)) (t-ftn (t-int) (t-int))))
```
(t-ftn (t-int) (t-int))
(t-id x (t-ftn (t-int) (t-int)))
(t-ftn (t-ftn (t-int) (t-int)) (t-ftn (t-int) (t-int))))
(t-id y (t-ftn (t-int) (t-int)))
(t-ftn (t-int) (t-int)))

> (parse-exp applE1)
(t-app
  (t-id x (t-ftn (t-int) (t-int)))
  (t-id y (t-ftn (t-int) (t-int)))
  (t-int))

> (type-annotations-consistent? (parse-exp var1))
#t
> (type-annotations-consistent? (parse-exp abs1))
#t
> (type-annotations-consistent? (parse-exp appl1))
#t
> (type-annotations-consistent? (parse-exp applE1))
#f

Submission deadline: Thursday, April 26, 2007, 2:10 p.m.
Submission procedure: on paper in class.