Type Inference CS345H 2011 Due 9:30am October 27, 2011

You must do this assignment solo.

Write a interpreter and **type inference** function for the TFWAE language we've discussed in class, extended with the language features described below.

As in previous assignments, implement the function `parse`, which consumes an expression in the language's concrete syntax and returns the abstract syntax representation of that expression. `parse` must accept only expressions in the grammar of the language.

In addition to `parse`, you must implement `interp` as in previous assignments, and you must implement the function `infer`, which consumes an abstract syntax expression (as returned by the `parse` function) and returns a `TFWAE-type`, if one exists. Please test cases that amply exercise all of the code you've written.

Features to Implement

Booleans

Introduce constants for true and false. Support operators and, or, not, and equality (=) on booleans. Also extend the operators on numbers to include less than (<) and equality (=).

Types

Your type system will have types number, boolean, and functions $A_1,...A_n \rightarrow B$ where $A_1,...A_n$ and B are types. Your type system will also need to include type variables and implement unification to compute types.

Conditionals

Use a standard `if` using the syntax described by the EBNF below. Note that `if` has three components:

- A test expression
- A "then" expression, which is the result if the *test* expression evaluates to **true**
- An "else" expression, which is the result otherwise.

Evaluation should signal an error for non-boolean test values.

Recursion

Any function defined at the top level of a `with` expression is considered a *recursive* function. Recursive functions can refer to *any identifier defined in the same* `with`. That is, they can refer to all other identifiers, whether or not they come before or after in the list of bindings. A non-recursive binding can only refer to identifiers are defined in *enclosing* `with` expressions. **For example, `(with ((f (fun (x)...) … ) …)` has a function definition at the top level.**

Multi-argument `fun`

Extend the `fun` language feature described in lecture so that functions can accept a list of zero or more arguments instead of just one. All arguments to the function must evaluate with the *same deferred substitutions*. An example of a multi-argument fun:

```
{{fun {x y} {* x y}} 2 3}
```

This evaluates to 6.

As you did for multi-armed `with`, you must ensure that the arguments to a function have distinct names.

Extra credit

Implement implicit polymorphism for functions defined in a `with` expression (See chapter 31).

Syntax of TFWAE

The syntax of the TFWAE language with these additional features can be captured with the following EBNF:

```
<TFWAE> ::= <num> | true | false                    constants
    | {and <TFWAE> <TFWAE>}                         operators
    | {or <TFWAE> <TFWAE>}
    | {not <TFWAE>}
    | {= <TFWAE> <TFWAE>}
    | {< <TFWAE> <TFWAE>}
    | {+ <TFWAE> <TFWAE>}
    | {- <TFWAE> <TFWAE>}
    | {* <TFWAE> <TFWAE>}
    | {/ <TFWAE> <TFWAE>}
    | <id>                                          variables
    | {if <TFWAE> <TFWAE> <TFWAE>}                  conditional
    | {with {{<id> <TFWAE>} ...} <TFWAE>}           variable binding
    | {fun {<id> ...} <TFWAE>}                      function definition
    | {<TFWAE> <TFWAE> ...}                         function application
```

where an `id` is not one of `and`, `or`, `not`, `=`, `<`, `+`, `-`, `*`, `/`, `with`, `if` or `fun`.
In this grammar, the ellipsis (`...`) means that the previous non-terminal is present zero or more times. If a `fun` or a `with` expression has duplicate identifiers, we consider it a syntax error. Therefore, such errors must be detected in `parse`. For example, parsing the following expressions must signal errors:

```
{with {{x 10} {x 20}} 50}

{fun {x x} 10}
```

Your abstract syntax type definition will have to use other names besides “if”, “and” and “or”, because these are reserved words in Racket.

Testing Your Code

You must write and include test cases that amply exercise all of the code you’ve written.

You can assume that the inputs are valid programs and that your program may raise arbitrary errors when given invalid inputs.

Support Code

Please once again use the PLAI language. Your code **must** adhere to the following template, without any changes:

```
;; parse : expression -> TFWAE
;; This procedure parses an expression into a TFWAE
(define (parse sexp)
  ...)

;; interp : TFWAE-> Env -> TFWAE-Value
;; This procedure interprets the given TFWAE in the context
;; of a deferred substitution and produces a result
;; in the form of a TFWAE-Value (either a closureV, boolV, or numV)
(define (interp expr ds)
  ...)

(define-type TFWAE-Type
  [numT]
  [boolT]
  [varT (name symbol?)]
  [funT (domain (listof TFWAE-Type?))
        (range TFWAE-Type?)])

;; infer : TFWAE -> TFWAE-Type
;; This procedure infers a type for the given expression
;; It must generate a set of constraints and then solve
;; the constraints to compute the type of all
(define (infer expr)
  ...)
```