Extended Interpreter CS345H 2011 Due 9:30am Sept 29, 2011

You must do this assignment solo.

Extended Interpreter

Write a parser and interpreter for the CFWAE language we've discussed in class, extended with the language features described below. Your interpreter should have eager application semantics and use deferred substitution. Call the new language CFWAE (conditionals, functions, with, and arithmetic expressions).

In each part of the assignment, 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 the function `interp`, which consumes an abstract syntax expression (as returned by the `parse` function) and returns a `CFWAE-value`. Please include a contract for every function that you write and include test cases that amply exercise all of the code you've written.

Features to Implement

Conditionals

To save the trouble of having to add boolean values and operators over them, create the construct `if0` using the syntax described by the EBNF below. Note that `if0` has three branches:

- A test expression
- A "then" expression, which evaluates if the *test* expression evaluates to zero
- An "else" expression, which evaluates for any other number.

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

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.

Syntax of CFWAE

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

```
<CFWAE> ::= <num>
    | {+ <CFWAE> <CFWAE>}
    | {- <CFWAE> <CFWAE>}
    | {* <CFWAE> <CFWAE>}
    | {/ <CFWAE> <CFWAE>}
    | <id>
    | {if0 <CFWAE> <CFWAE> <CFWAE>}
    | {with {{<id> <CFWAE>} ...} <CFWAE>}
    | {fun {<id> ...} <CFWAE>}
    | {<CFWAE> <CFWAE> ...}
where an id is not +, -, *, /, with, if0 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}
```

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:

```
(define-type Binding
  [binding (name symbol?) (named-expr CFWAE?)])

(define-type CFWAE
  [num (n number?)]
  [binop (op procedure?) (lhs CFWAE?) (rhs CFWAE?)]
  [with (lob (listof Binding?)) (body CFWAE?)]
  [id (name symbol?)]
  [if0 (c CFWAE?) (t CFWAE?) (e CFWAE?)]
  [fun (args (listof symbol?)) (body CFWAE?)]
  [app (f CFWAE?) (args (listof CFWAE?))])

(define-type Env
  [mtEnv]
  [anEnv (name symbol?) (value CFWAE-Value?) (env Env?)])

(define-type CFWAE-Value
  [numV (n number?)]
  [closureV (params (listof symbol?))
            (body CFWAE?)
            (env Env?)])

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

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