HW4-2022

Homework 4 (Due Monday 9/29/2022 at 11:59pm)

To submit this assignment, simply upload the files in your GitHub repository to you GitHub Classroom repository for this assignment (as you have done for earlier assignments). In contrast to Assignments 1 and 2 where you could put the programs for all problems in a single file (HW01.rkt for Assignment 1 and HW02.rkt for Assignment 2), you must store the solution to each problem in a separate `.rkt` file, namely `HW04-1.rkt`, `HW04-2.rkt`, `HW04-3.rkt`, and `HW04-4.rkt` (if you do the extra credit problem) for problems 1, 2, 3, and 4 (extra credit). Unfortunately, none of the languages supported by DrRacket allow these files to be combined. The *Pretty Big* Racket language allows top-level identifiers (variable names) to be redefined, but it does *not* support `check-expect`. All of the student languages—which are the only ones that support `check-expect`–prohibit redefinition of identifiers.

Embed any answer text that is not executable in a DrRacket block comment or block commenting brackets (#| and |#).

Use the **Intermediate Student with lambda** language.

Given the Racket structure definitions:

```
(define-struct sum (left right))
(define-struct prod (left right))
(define-struct diff (left right))
(define-struct quot (left right))
```

an `ArithExpr` is either:

- a number `n`,
- a sum `(make-sum ae1 ae2)`,
- a product `(make-prod ae1 ae2)`,
- a difference `(make-diff ae1 ae2)`, or
- a quotient `(make-quot ae1 ae2)`

where `n` is a number (as defined in Racket), and `ae1` and `ae2` are `ArithExprs`.

The following 4 exercises involve the data type `ArithExpr`. If you are asked to write a function(s), follow the design recipe: contract, purpose, examples /tests, template instantiation, code, testing (which happens automatically when the examples are presented using `(check-expect ...)`). Follow the same recipe for any help function that you introduce. You may use any of the library functions shown in class lectures including `append`.

1. (40 pts.) Write an evaluator for arithmetic expressions as follows:
   - Write the (function) template for `ArithExpr`
   - Write a function `to-list` that maps an `ArithExpr` to the corresponding "list" representation in Racket. Numbers are unchanged. Some other examples include:

     ```
     (to-list 12) => 12
     (to-list (make-sum (make-prod 4 7) 25)) => '(+ (* 4 7) 25)
     (to-list (make-quot (make-diff 4 7) 25)) => '(/ (- 4 7) 25)
     ```

     Notes:

     a. The notation `'(+ (* 4 7) 25)` abbreviates `(list '+ (list '* 4 7) 25)`.
     b. You need to define the output type (named RacketExpr) for this function, but you can omit the template because this assignment does not include any functions that process this type. There are several mathematically distinct definitions that are correct. Some are more restrictive (narrower) than others but all our correct.
     c. The notation `'(+ (* 4 7) 25)` abbreviates `(list '+ (list '* 4 7) 25)`.
   - Write a function `eval: ArithExpr -> number` that evaluates an `ArithExpr`. Your evaluator should produce exactly the same result for an `ArithExpr E` that Racket evaluation would produce for the list representation `(to-list E)`.

2. (40 pts.) Extend the definition of `ArithExpr` as follows:
   - Add a clause for variables represented as Racket symbols.
   - Write the (function) template for this extended definition; it should similar to your template for `ArithExpr` from Problem 1.
   - Modify your definition of `to-list` to support the expanded definition of `ArithExpr`.
   - Given the Racket structure definition:

     ```
     (define-structure binding (var val))
     ```

     a `binding` is `(make-binding s n)` where `s` is a symbol and `n` is a number. An `environment` is a `(list-of binding)`. Write a (function) template for processing an `environment`.
   - Define a top-level variable `empty-env` that is bound to the empty environment containing no bindings (*i.e.*, the empty list). Note that `empty-env` is really a constant since variables cannot be rebound in our functional subset of Racket.
   - Write a function `extend` that takes environment `env`, a symbol `s`, and a number `n` , and returns an extended environment identical to `env` except that it adds the additional binding of `s` to `n`.
     The definition of `extend` is trivial; it requires no recursion. As a result, `extend` satisfies the invariant

```
(check-expect (extend empty-env s n) (list (make-binding s n)))
```

where s is any symbol and n is any number. Hence,

```
(extend empty-env 'a 4) => (list (make-binding 'a 4))
```

In the remainder of the problem, use `empty-env` and `extend` to define example environments for test cases.

- Write a function `lookup` that takes a symbol `s` and an environment `env` and returns the first `binding` in `env` with a `var` component that equals `s`. If no match is found, `lookup` returns empty. Note that the return type of `lookup` is not simply `binding` because it can return `empty`. Define the a new union type called `option-binding` for the the return type. You do not need to write a template for this type since it is trivial.
- Write a new `eval` function for the expanded definition of `ArithExpr`. The new `eval` takes *two* arguments: an `ArithExpr E` to evaluate and an `environment env` specifying the values of free variables in `E`. For example,

```
(eval 'x (extend empty-env 'x 17)) => 17
(eval (make-prod 4 7) (extend empty-env 'x 17)) = 28
(eval 'y (extend empty-env 'x 17)) => some form of run-time error
```

If an `ArithExpr E` contains a free variable that is not bound in the `environment env` , then `(eval E env)` will naturally produce some form of run-time error if you have correctly coded `eval`. Do *not* explicitly test for this form of error.

3. (20 pts.) An `environment` is really a finite function (a finite set of ordered pairs). It is *finite* in the sense that it can be completely defined by a finite table, which is not true of nearly all the primitive and library functions in Racket (and other programming languages). Even the identity function is *not* finite. For the purpose of this exercise, we redefine the type `environment` as `(symbol -> option-binding)`.
   - Rewrite `eval` to use `environment` defined as a finite function in `(symbol -> option-binding)` instead of as a `(list-of option-binding)`. If you cleanly coded your definition of `eval` in the preceding problem using `lookup`, `make-binding`, and `extend`, all that you have to do to your solution to the previous problem is redefine the bindings of `lookup`, `empty-env`, and `extend`, and revise your test cases for `extend`. You can literally copy the entire text of your solution to problem 2; change the definitions of `lookup`, `empty-env`, and `extend` ; update your documentation (annotations) concerning the `environment` type; and revise your tests for `extend`. Note that `extend` cannot be tested (since the result is a function!) without using `lookup` to examine it. (Note: if you wrote a correct solution to problem 2, you can do this problem is less than 15 minutes!)
     Hint: you can use `lambda`-notation in Racket to define a constant function for `empty-env`, and `extend` can be defined as a functional that takes a function (representing an environment) and adds a new pair to the function--using a `if` embedded inside a `lambda`-abstraction.
4. Extra Credit (50 pts.) Add support for `lambda`-expressions in your evaluator from Problem 2 as follows:
   - Extend the definition of `ArithExpr` by adding a clause for unary **`lambda-abstractions`** and a clause for **unary applications** of an `ArithExpr` to an `ArithExpr`. Use the name `lam` for the structure representing a **`lambda-abstraction`** and the names `var` and `body` for the accessors of this structure. Use the name `app` for the structure representing a **unary application** and the names `rator` and `rand` for the argument of this structure. Note that the `rator` of an `app` is an `ArithExpr` not a `lam` (which is a proper subtype of `ArithExpr`).
   - Write a (function) template for this additional expansion of the definition of `ArithExpr`.
   - Extend the definition of `to-list` to support this expansion of the definition of `ArithExpr`. You print the corresponding "concrete" syntax that would be fed as input to a compiler. Hence,
     `(to-list (make-lam 'x (make-plus 'x 'y))) => (list 'lambda (list 'x) '(+ x y)) = '(lambda (x) (+ x y))`
   - Extend the definition of `eval` to support this expansion of `ArithExpr`. Note that `eval` can now return functions as well as numbers. Your biggest challenge is determining a good representation for function values. What does `eval` return as the value of a `lam` input? That input may contain free variables. In principle, you could represent the value of the `lam` input by a revised `lam` (with no free variables) obtained by substituting the values for free variables from the environment input (just like we do in hand-evaluation). But this approach is tedious and computationally expensive. A better strategy is to define a structure type (called a *closure*) to represent a function value. The structure type must contain the original `lam` and a description of what substitution (of values for identifiers) would have been made, deferring the actual substitution just as `eval` defers substitutions by maintaining an environment.