Design and Implementation of Programming Languages

Project Specification

Set-up: For this assignment, edit a copy of project.rkt, which is attached. In particular,
replace occurrences of "CHANGE" to complete the problems. Do not use expressions with side effects (set!, set-mcar!, etc.) in your code.

Overview: This project has to do with NUMEX (**Num**ber-**Ex**pression Programming Language). NUMEX programs are written directly in Racket by using the constructors defined by the structs defined at the beginning of project.rkt (Note: you must define missing ones). Here is the definition of NUMEX's syntax:

- If s is a Racket string, then (var s) is a NUMEX expression (variables).
- If n is a Racket integer, then (num n) is a NUMEX expression (number constants).
- If b is a Racket boolean, then (bool b) is a NUMEX expression (boolean constants).
- If $e1$ and $e2$ are NUMEX expressions, then (plus $e1$ $e2$) is a NUMEX expression (addition).
- If $e1$ and $e2$ are NUMEX expressions, then (minus $e1$ $e2$) is a NUMEX expression (subtraction).
- If $e1$ and $e2$ are NUMEX expressions, then (mult $e1$ $e2$) is a NUMEX expression (multiplication).
- If $e1$ and $e2$ are NUMEX expressions, then (div $e1$ $e2$) is a NUMEX expression (division).
- If $e1$ is a NUMEX expression, then (neg $e1$) is a NUMEX expression (negation).
- If $e1$ and $e2$ are NUMEX expressions, then (andalso $e1$ $e2$) is a NUMEX expression (logical conjunction).
- If $e1$ and $e2$ are NUMEX expressions, then (orelse $e1$ $e2$) is a NUMEX expression (logical disjunction).
- If $e1$, $e2$, and $e3$ are NUMEX expressions, then (cnd $e1$ $e2$ $e3$) is a NUMEX expression. It is a condition where the result is $e2$ if $e1$ is true, else the result is $e3$. Only one of $e2$ and $e3$ is evaluated.
- If $e1$ and $e2$ are NUMEX expressions, then (iseq $e1$ $e2$) is a NUMEX expression. (comparison).
- If $e1$, $e2$, and $e3$ are NUMEX expressions, then (ifnzero $e1$ $e2$ $e3$) is a NUMEX expression. It is a condition where the result is $e2$ if $e1$ is not zero, else the result is $e3$. Only one of $e2$ and $e3$ is evaluated.

- If $e1$, $e2$, $e3$, and $e4$ are NUMEX expressions, then (ifleq $e1$ $e2$ $e3$ $e4$) is a NUMEX expression. It is a conditional where the result is $e4$ if $e1$ is strictly greater than $e2$, else the result is $e3$. Only one of $e3$ and $e4$ is evaluated.
- If $s1$ and $s2$ are Racket strings and e is a NUMEX expression, then (lam $s1$ $s2$ e) is a NUMEX expression (a function). In e, $s1$ is bound to the function itself (for recursion) and $s2$ is bound to the only argument. Also, (lam null $s2$ e) is allowed for anonymous nonrecursive functions.
- If $e1$ and $e2$ are NUMEX expressions, then (apply $e1$ $e2$) is a NUMEX expression (function application).
- If s is a Racket string, and $e1$ and $e2$ are NUMEX expressions, then (with s $e1$ $e2$) is a NUMEX expression (a let expression where the value of $e1$ is bound to s in $e2$).
- If $e1$ and $e2$ are NUMEX expressions, then (apair $e1$ $e2$) is a NUMEX expression (pair constructor).
- If $e1$ is a NUMEX expression, then (1st $e1$) is a NUMEX expression (the first part of a pair).
- If $e1$ is a NUMEX expression, then (2nd $e1$) is a NUMEX expression (the second part of a pair).
- (munit) is a NUMEX expression (holding no data, much like () in ML or null in Racket). Notice (munit) is a NUMEX expression, but munit is not.
- If $e1$ is a NUMEX expression, then (ismunit $e1$) is a NUMEX expression (testing for (munit)).
- (closure env f) is a NUMEX value where f is a NUMEX function and env is an environment that maps variables to values. Closures do not appear in programs; they result from evaluating functions.
- If $s1$ is a Racket string and $s2$ is a Racket string and $s3$ is a Racket string and $e1$ is a NUMEX expression and $e2$ is NUMEX expression and $e3$ is a NUMEX expression and $e4$ is NUMEX expression, then (letrec $s1$ $e1$ $s2$ $e2$ $s3$ $e3$ $e4$) is a NUMEX expression (a letrec expression for recursive definitions where the the value of $e1$ is bound to $s1$ and the value of $e2$ is bound to $s2$ and the value of $e3$ is bound to $s3$ in the $e4$).
- If e is a NUMEX expression and m is a NUMEX munit, then (queue e m) is a NUMEX expression. If e is a NUMEX expression and q is a NUMEX queue, then (queue e q) is a NUMEX expression (queue constructor).
- If e is a NUMEX expression and q is a NUMEX queue, then (enqueue e q) is a NUMEX expression (queue after enqueue expression into queue). If q is a NUMEX queue, then (dequeue q) is a NUMEX expression (queue after dequeue expression from queue).
- If q is a NUMEX queue, then (extract q) is a NUMEX expression (top expression in queue).

A NUMEX *value* is a NUMEX number constant, a NUMEX boolean constant, a NUMEX closure, a NUMEX munit, or a NUMEX pair of NUMEX values. Similar to Racket, we can build list values out of nested pair values that end with a NUMEX munit. Such a NUMEX value is called a NUMEX list.

You should *not* assume NUMEX programs are syntactically correct (e.g., things like (num "hi") or (num (num 37)) must be handled). And do *not* assume NUMEX programs are free of type errors like (plus (munit) (num 7)), (1st (num 7)) or (div (bool #t) (num 2)).

Instructions: Upload your modified project.rkt and projectTest.rkt through the Moodle website.

Problems:

1. Warm-Up

(a) Write a Racket function racketlist->numexlist that takes a Racket list, which may even be a list of NUMEX values, and produces a NUMEX list with the same elements in the same order.

(b) Write a Racket function numexlist->racketlist that takes a NUMEX list and produces a Racket list with the same elements in the same order.

2. Implementing NUMEX

Write an interpreter for NUMEX. It should be a Racket function eval-exp that takes a NUMEX expression e and either returns the NUMEX value that e evaluates to under the empty environment or calls Racket's error if evaluation encounters a run-time NUMEX type error or unbound NUMEX variables.

An NUMEX expression is evaluated in an environment (for evaluating variables, as usual). In your interpreter, use a Racket list of Racket pairs to represent this environment (which is initially empty) so that you can use the envlookup function, after completing it. Here is a description of the semantics of NUMEX expressions:

- All values (including closures) evaluate to themselves. For example, (eval-exp (num 17)) would return (num 17), not 17.
- A variable evaluates to the value associated with it in the given environment.
- An arithmetic operation (addition, subtraction, multiplication, and division) evaluates to the result of what its operands evaluate to. Note: the operands must be numbers.
- A logical operation (andalso and orelse) evaluates to the result of what its operands evaluate to. Note: short-circuit evaluations are desired, and the operands must be booleans.

- A negation (neg e) evaluates to the opposite (negation) of what e evaluates to. Note: e can be a number or a boolean.
- For (cnd $e1$ $e2$ $e3$), the expression $e1$ first evaluates to a boolean value. If the resulting value is (bool #t), the whole expression evaluates to what $e2$ evaluates to. The expression evaluates to the value of $e3$ otherwise.
- The evaluation of (iseq $e1$ $e2$) involves the evaluation of $e1$ and $e2$. The resulting values is (bool #t) if the value of $e1$ equals to the value of $e2$. Otherwise, the expression evaluates to (bool #f). Note: $e1$ and $e2$ can be numbers/booleans.
- For (ifnzero $e1$ $e2$ $e3$), the expression $e1$ first evaluates to a value. If the resulting value is not zero, the whole expression evaluates to what $e2$ evaluates to. The expression evaluates to the value of $e3$ otherwise.
- The evaluation of (ifleq $e1$ $e2$ $e3$ $e4$) involves the evaluation of $e1$ and $e2$. If the value of $e1$ is strictly greater than the value of $e3$, then it evaluates to the value of $e4$. Otherwise, $e3$ must be evaluated.
- For (with s $e1$ $e2$), the expression $e2$ evaluates to a value in an environment extended to map the name s to the evaluated value of $e1$.
- Functions are lexically scoped in the sense that a function evaluates to a closure holding the function and the current environment.
- For (apply $e1$ $e2$), the expression $e1$ first evaluates to a value. If the resulting value is not a closure, an error should be arisen. Otherwise, it evaluates the closure's function's body in the closure's environment extended to map the function's name to the closure (unless the name field is null) and the function's argument to the result of the evaluation of $e2$.
- The (apair $e1$ $e2$) construct makes a (new) pair holding the results of the evaluations of $e1$ and $e2$.
- If the result of evaluating $e1$ in (1st $e1$) is an apair, then the first part is returned. Otherwise, it returns an error. Similarly, the evaluation of (2nd $e1$) is the second part of the given pair.
- For (ismunit $e1$), the expression $e1$ first evaluates to a value. If the resulting value is a munit expression, then the result is the NUMEX value (bool #t), else it is the NUMEX value (bool #f).
- For (letrec $s1$ $e1$ $s2$ $e2$ $s3$ $e3$ $e4$) the expression $e4$ evaluates to a value in an environment extended to map the name $s1$ to the evaluated value of $e1$ and the name $s2$ to the evaluated value of $e2$ and the name $s3$ to the evaluated value of $e3$.
- If the result of evaluating e is a NUMEX expression and the result of evaluating m is a munit, the (queue e m) construct makes a queue holding the result of the evaluation of e and the result of the evaluation of m. If the result of evaluating e is a NUMEX expression and the result of evaluating q is a queue, the (queue e q)

construct makes a queue holding the results of the evaluation of e and the evaluation of q. Otherwise, it returns an error.

• If the result of evaluating e is a NUMEX expression and the result of evaluating q is a queue, the (enqueue e q) construct makes a queue holding the results of the evaluation of e and the evaluation of q. Otherwise, it returns an error.

• If the result of evaluating q is a queue, the (dequeue q) construct makes a queue holding the results of the evaluation of q without queue's top element. Otherwise, it returns an error.

• If the result of evaluating q is a queue, the (extract q) construct gives queue's top element. Otherwise, it returns an error.

3. Extending the Language (Bonus)

NUMEX is a small language, but we can write Racket functions that act like NUMEX macros so that users of these functions feel like NUMEX is larger. The Racket functions produce NUMEX expressions that could then be put inside larger NUMEX expressions or passed to eval-exp. In implementing these Racket functions, do not use closure (which is used only internally in eval-exp). Also do not use eval-exp (we are creating a program, not running it).

(a) Write a Racket function ifmunit that takes three NUMEX expressions $e1$, $e2$, and $e3$. It returns a NUMEX expression that first evaluates $e1$. If the resulting value is NUMEX's munit, then it evaluates $e2$ and that is the overall result. Otherwise, $e3$ must be evaluated.

(b) Write a Racket function with* that takes a Racket list of Racket pairs '(($s1$.$e1$)...(si .ei)...(sn .en)) and a final NUMEX expression en+1. In each pair, assume si is a Racket string and ei is a NUMEX expression. with* returns a NUMEX expression whose value is en+1 evaluated in an environment where each si is a variable bound to the result of evaluating the corresponding ei for $1 \le i \le n$. The bindings are done sequentially, so that each ei is evaluated in an environment where $s1$ through $si-1$ have been previously bound to the values $e1$ through $ei-1$.

(c) Write a Racket function ifneq that takes four NUMEX expressions $e1$, $e2$, $e3$, and $e4$ and returns a NUMEX expression that acts like ifleq except $e3$ is evaluated if and only if $e1$ and $e2$ are not equal numbers/booleans. Otherwise, the whole expression evaluates to what $e4$ evaluates to. Assume none of the arguments to ifneq use the NUMEX variables _x or _y. Use this assumption so that when an expression returned from ifneq is evaluated, $e1$ and $e2$ are evaluated exactly once each.

4. Using the Language (Bonus)

We can write NUMEX expressions directly in Racket using the constructors for the structs and (for convenience) the functions we wrote in the previous problem.

(a) Bind to the Racket variable numex-filter a NUMEX function that acts like map (as we use in ML). Your function should be curried: it should take a NUMEX function and return a NUMEX function that takes a NUMEX list and applies the function to every element of the list returning a new NUMEX list with all the elements for which the function returns a number other than zero (causing an error if the function returns a non-number). Recall a NUMEX list is munit or a pair where the second component is a NUMEX list.

(b) Bind to the Racket variable numex-all-gt a NUMEX function that takes a NUMEX number *i* and returns a NUMEX function that takes a NUMEX list of NUMEX numbers and returns a new NUMEX list of NUMEX numbes containing the elements of the input list (in order) that are greater than *i* (hint: Use numex-filter).