Design and Implementation of Programming Languages Project Specification
Fall 2023

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 e_1 and e_2 are NUMEX expressions, then (`plus` e_1 e_2) is a NUMEX expression (addition).
- If e_1 and e_2 are NUMEX expressions, then (`minus` e_1 e_2) is a NUMEX expression (subtraction).
- If e_1 and e_2 are NUMEX expressions, then (`mult` e_1 e_2) is a NUMEX expression (multiplication).
- If e_1 and e_2 are NUMEX expressions, then (`div` e_1 e_2) is a NUMEX expression (division).
- If e_1 is a NUMEX expression, then (`neg` e_1) is a NUMEX expression (negation).
- If e_1 and e_2 are NUMEX expressions, then (`andalso` e_1 e_2) is a NUMEX expression (logical conjunction).
- If e_1 and e_2 are NUMEX expressions, then (`orelse` e_1 e_2) is a NUMEX expression (logical disjunction).
- If e_1, e_2, and e_3 are NUMEX expressions, then (`cnd` e_1 e_2 e_3) is a NUMEX expression. It is a condition where the result is e_2 if e_1 is true, else the result is e_3. Only one of e_2 and e_3 is evaluated.

- If e_1 and e_2 are NUMEX expressions, then (`iseq` e_1 e_2) is a NUMEX expression. (equality).
- If e_1 and e_2 are NUMEX expressions, then (`isls` e_1 e_2) is a NUMEX expression. (less-than).
- If e_1 and e_2 are NUMEX expressions, then (`isgt` e_1 e_2) is a NUMEX expression. (greater-than).
- If s_1 and s_2 are Racket strings and e is a NUMEX expression, then (`lam` s_1 s_2 e) is a NUMEX expression (a function). In e, s_1 is bound to the function itself (for recursion) and s_2 is bound to the only argument. Also, (`lam null` s_2 e) is allowed for anonymous non recursive functions.
- If e_1 and e_2 are NUMEX expressions, then (`apply` e_1 e_2) is a NUMEX expression (function application).
- If s is a Racket string, and e_1 and e_2 are NUMEX expressions, then (`with` s e_1 e_2) is a NUMEX expression (a let expression where the value of e_1 is bound to s in e_2).
- If e_1 and e_2 are NUMEX expressions, then (`apair` e_1 e_2) is a NUMEX expression (pair constructor).
- If e_1 is a NUMEX expression, then (`1st` e_1) is a NUMEX expression (the first part of a pair).
- If e_1 is a NUMEX expression, then (2nd e_1) 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 e_1 is a NUMEX expression, then (`ismunit` e_1) 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 s_1 is a Racket string and s_2 is a Racket string and e_1 is a NUMEX expression and e_2 is NUMEX expression and e_3 is a NUMEX expression and e_4 is NUMEX expression and e_5 is a NUMEX expression, then (letrec s_1 e_1 s_2 e_2 s_3 e_3 s_4 e_4 e_5) is a NUMEX expression (a letrec expression for recursive definitions where the the value of e_1 is bound to s_1 and the value of e_2 is bound to s_2 and the value of

e_2 is bound to s_3 and the value of e_4 is bound to s_4 in the e_5).

- If s is a Racket string and e is a NUMEX expression, then (key s e) is a NUMEX expression (key contructor).
- If k is a NUMEX key and m is a NUMEX munit, then (record k m) is a NUMEX expression (record contructor).
- If k is a NUMEX key and r is a NUMEX record, then (record k r) is a NUMEX expression (record constructor).
- If s is a Racket string and r is a NUMEX record, then (value s r) is a NUMEX expression (value of string in record).

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` 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.

A 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` e_1 e_2 e_3), the expression e_1 first evaluates to a boolean value. If the resulting value is (`bool #t`), the whole expression evaluates to what e_2 evaluates to. The expression evaluates to the value of e_3 otherwise.
- The evaluation of (`iseq` e_1 e_2) involves the evaluation of e_1 and e_2. The resulting value is (`bool #t`) if the value of e_1 equals the value of e_2. Otherwise, the expression evaluates to (`bool #f`). Note: e_1 and e_2 can be numbers/booleans.
- The evaluation of (`isls` e_1 e_2) involves the evaluation of e_1 and e_2. The resulting value is (`bool #t`) if the value of e_1 is less than the value of e_2. Otherwise, the expression evaluates to (`bool #f`). Note: e_1 and e_2 can only be numbers.
- The evaluation of (`isgt` e_1 e_2) involves the evaluation of e_1 and e_2. The resulting value is (`bool #t`) if the value of e_1 is greater than the value of e_2. Otherwise, the expression evaluates to (`bool #f`). Note: e_1 and e_2 can only be numbers.
- For (`with` s e_1 e_2), the expression e_2 evaluates to a value in an environment extended to map the name s to the evaluated value of e_1.
- Functions are lexically scoped in the sense that a function evaluates to a closure holding the function and the current environment.
- For (`apply` e_1 e_2), the expression e_1 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 e_2.

- The (`apair` e_1 e_2) construct makes a (new) pair holding the results of the evaluations of e_1 and e_2.
- If the result of evaluating e_1 in (`1st` e_1) is an `apair`, then the first part is returned. Otherwise, it returns an error. Similarly, the evaluation of (`2nd` e_1) is the second part of the given pair.
- For (`ismunit` e_1), the expression e_1 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 s_1 e_1 s_2 e_2 e_3 s_4 e_4 e_5) the expression e_5 evaluates to a value in an environment extended to map the name s_1 to the evaluated value of e_1 and the name s_2 to the evaluated value of e_2 and the name s_3 to the evaluated value of e_3 and the name s_4 to the evaluated value of e_4.
- If s is a Racket string and the result of evaluating e is a NUMEX expression, the (key s e) construct makes a key holding the corresponding value of s which is e. Otherwise, it returns an error.
- If the result of evaluating k is a key and the result of evaluating m is a munit, the (record k m) construct makes a record holding the result of the evaluation of k and the result of the evaluation of m. Otherwise, it returns an error.
- If the result of evaluating k is a key and the result of evaluating r is a record, the (record k r) construct makes a record holding the results of the evaluation of k and the evaluation of r. Otherwise, it returns an error.
- If the result of evaluating s is a string and the result of evaluating r is a record, the (value s r) returns the corresponding value of s in r. If s does not exist in r, the (value s r) returns (munit). Otherwise, it returns an error.

3. Implementing a simple type system for NUMEX

Write a simple type system for NUMEX. It should be a Racket function `infer-exp` that takes a NUMEX expression e and either returns the NUMEX type that e evaluates to, under the empty environment or calls Racket's error if evaluation encounters a type error.

The NUMEX type system consists of three primitive types:

1. **Integer**: a type for numeric values which is represented as a Racket string "`int`".

2. **Boolean**: a type for boolean values (true and false) which is represented as a Racket string "`bool`".
3. **Null**: a type for null value in NUMEX (munit) which is represented as a Racket string "`null`".

A NUMEX expression's type is inferred in an environment (for storing each variable's type). The mentioned environment in your type system is exactly like the environment you implemented in NUMEX interpreter.

NOTE1: We are trying to infer types from expressions and not from their values or execution progress, **you should not execute the expression and you should not use eval-exp in the NUMEX type system**.

NOTE2: You should implement the type system for just a part of NUMEX language in which its type inference rules are explained, not for the whole NUMEX language.

Here is a description of the type inference rules of NUMEX expressions:

- All numeric values are inferred as integer types. For example, (`infer-exp (num 17)`) would return "`int`".
- All boolean values are inferred as boolean types. For example, (`infer-exp (bool #t)`) would return "`bool`".
- A variable's type is the type associated with it in the given environment.
- The type of an arithmetic operation (addition, subtraction, multiplication, and division) is inferred as integer if both operands are inferred as integers, otherwise, an error should be raised (implement this inference rule just for addition).
- The type of a logical operation (andalso and orelse) is inferred as boolean if both operands are inferred as booleans, otherwise, an error should be raised (implement this inference rule just for andalso).
- The type of a negation (`neg` e) is inferred as the type of its argument expression (e).
- For (`cnd` e_1 e_2 e_3), if e_1 is inferred as a boolean and e_2 and e_3 are both inferred as the same type, the whole expression's type is inferred as the type of e_2 (or e_3). Otherwise, an error should be raised.
- The type of (`iseq` e_1 e_2) is inferred as boolean if e_1 and e_2 are both inferred as the same type. Otherwise, an error should be raised.
- The type of (`isls` e_1 e_2) is inferred as boolean if e_1 and e_2 are both inferred as integers. Otherwise, an error should be raised.
- The type of (`isgt` e_1 e_2) is inferred as boolean if e_1 and e_2 are both inferred as integers. Otherwise, an error should be raised.
- For (`with` s e_1 e_2), the expression e_2 is inferred as a type in an

environment extended to map the name s to the inferred type of e_1.

- For functions we consider two assumptions: The first assumption is that the function expression is constructed with `tlam` instead of `lam` which contains the function argument type in it. The second assumption is that there is no use of `apply` in the function's body expression. The function's type is inferred as `(function T1 T2)` which `T1` is the type of the function's argument and `T2` is the type of the function's body expression which is inferred as a type in an environment extended to map the function's argument to its type that is given in `tlam` structure.
- For (`apply` e_1 e_2), if e_1 is inferred as a function (let's call it f) and e_2 is inferred as a type which is equal to the type of f's argument, the whole expression's type is inferred as f's output type. Otherwise, an error should be raised.
- Note: In the NUMEX type system, we assume that `apair` is just for NUMEX list construction.
- The type of (`apair` e_1 e_2) is `(collection T)` if e_1 is inferred as a `T` and e_2 is inferred as either `(collection T)` or `null`. Otherwise, an error should be raised.
- If an expression like e is inferred as a `(collection T)`, (`1st` e) is inferred as a `T`. Otherwise, an error should be raised.
- If an expression like e is inferred as a `(collection T)`, (2nd e) is inferred as a `(collection T)`. Otherwise, an error should be raised.
- The type of (`ismunit` e_1) is inferred as boolean if the expression e_1 is inferred as either `(collection T)` or `null`. Otherwise, an error should be raised.

4. Extending the Language

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 e_1, e_2, and e_3. It returns a NUMEX expression that first evaluates e_1. If the resulting value is NUMEX's `munit`, then it evaluates e_2 and that is the overall result. Otherwise, e_3 must be evaluated.

(b) Write a Racket function `with*` that takes a Racket list of Racket pairs $'\left(\left(s_1 \,.\, e_1\right) \ldots \left(s_i \,.\, e_i\right) \ldots \left(s_n \,.\, e_n\right)\right)$ and a final NUMEX expression e_{n+1}. In each pair, assume s_i is a Racket string and e_i is a NUMEX expression. `with*` returns a NUMEX expression whose value is e_{n+1} evaluated in an environment where each s_i is a variable bound to the result of evaluating the corresponding e_i for $1 \leq i \leq n$. The bindings are done sequentially, so that each e_i is evaluated in an environment where s_1 through s_{i-1} have been previously bound to the values e_1 through e_{i-1}.

(c) Write a Racket function `ifneq` that takes four NUMEX expressions e_1, e_2, e_3, and e_4 and returns a NUMEX expression in which e_3 is evaluated if and only if e_1 and e_2 are not equal numbers/booleans. Otherwise, the whole expression evaluates to what e_4 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, e_1 and e_2 are evaluated exactly once each.

5. Using the Language

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. In this problem, we want to implement some well-known functions in functional programming.

(a) Bind to the Racket variable `numex-foldr` a NUMEX function that acts like fold. Fold is a function that takes a binary function, an initial value (an accumulator), and a list. If the list is empty fold returns the initial value. Otherwise, fold applies its input function to the first element of the list and the result of folding the rest of the list. Recall a NUMEX list is `munit` or a pair where the second component is a NUMEX list. Note that your function should be curried

(b) Using `numex-foldr` bind to the Racket variable `numex-concatenate` a NUMEX function that takes a NUMEX list `xs` and returns a NUMEX function that takes a NUMEX list `ys` and returns a new NUMEX list that is the concatenation of `xs` and `ys`.

(c) A list can represent a non-deterministic value. For example `[3, 4, 5]` represent 3, 4 and 5 at the same time. Moreover, the lambda function `λ x -> [x, -x]` represents a non-deterministic function that takes a number and produces two results: one negated and one unchanged. Using `numex-concatenate` bind to the Racket variable `numex-bind`. `numex-bind` is a function that takes a non-deterministic value (a NUMEX list) and a non-deterministic function and returns a new

non-deterministic value (a NUMEX list) by applying the non-deterministic function to the non-deterministic value. For example, calling `numex-bind` with `[3,4,5]` and `λ x -> [x, -x]` must return `[3,-3,4,-4,5,-5]`. Note that your function should be curried.

6. Exploring the type system

Try to write two NUMEX expressions directly in Racket using the constructors for the structs we wrote in the previous problems. the expressions explanations are as below:

(a) Write an expression that the type system infers it as a type error but the interpreter evaluates it fine (with no errors).

(b) Write an expression that the type system infers it fine but the interpreter evaluates it to an error.

7. Challenging Problem

Write a second version of `eval-exp` (bound to `eval-exp-c`) that builds closures with smaller environments: When building a closure, it uses an environment that is like the current environment but holds only variables that are free variables in the function part of the closure. (A free variable is a variable that appears in the function without being under some shadowing binding for the same variable.)

Avoid computing a function's free variables more than once. Do this by writing a function `compute-free-vars` that takes an expression and returns a different expression that uses `fun-challenge` everywhere in place of `fun`. The new struct `fun-challenge` (provided to you; do not change it) has a field `freevars` to store exactly the set of free variables for the function. Store this set as a Racket set of Racket strings. (Sets are predefined in Racket's standard library; consult the documentation for useful functions such as `set`, `set-add`, `set-member?`, `set-remove`, `set-union`, and any other functions you wish.)

You must have a top-level function `compute-free-vars` that works as just described – storing the free variables of each function in the `freevars` field – so the grader can test it directly. Then write a new "main part" of the interpreter that expects the sort of NUMEX expression that `compute-free-vars` returns. The case for function definitions is the interesting one.