Молчать на семи языках

Язык: Racket

Максимальное количество баллов: 17

Файл **`hw5.rkt`** для вас уже предоставлен, просто добавьте в него свой код. Не используйте изменяемые данные и мутирующие функции: **`set!`**, **`set-mcar!`** и т. п.

В этой работе нужно иметь дело с MUPL (a Made Up Programing Language). MUPL программы пишутся прямо в Racket выражениях (для этого определён набор **`struct`**). Синтаксис MUPL определяется следующим образом:

- Если **`s`** - строка, то **`(var s)`** это MUPL выражение (использование переменной).
- Если **`n`** - это целое число, то **`(int n)`** это MUPL выражение (целочисленная константа).
- Если **`e1`** и **`e2`** - это MUPL выражения, то **`(add e1 e2)`** это MUPL выражение (сложение).
- Если **`s1`** и **`s2`** - строки, а e - MUPL выражение, то **`(fun s1 s2 e)`** это MUPL выражение (определение функции). В выражении **`e s1`** привязывается к этой функции (для рекурсии), а **`s2`** - к единственному аргументу. Для определения безымянных нерекурсивных функций разрешена конструкция **`(fun #f s2 e)`**.
- Если **`e1`**, **`e2`**, **`e3`**, **`e`**4 - MUPL выражения, то **`(ifgreater e1 e2 e3 e4)`** это MUPL выражение (условное выражение). Если результат вычисления **`e1`** строго больше результата **`e2`**, то результат условного выражения **`e3`**, иначе - **`e4`**. Вычисляется только одно из выражений **`e3`**, **`e4`**.
- Если **`e1`** и **`e2`** - MUPL выражения, то **`(call e1 e2)`** это MUPL выражение (вызов функции).
- Если **`s`** - строка, а **`e1`** и **`e2`** - MUPL выражения, то **`(mlet s e1 e2)`** это MUPL выражение (выражение **`let`**, где значение выражения **`e1`** привязывается к имени **`s`**, а затем вычисляется тело **`e2`**).
- Если **`e1`** и **`e2`** - MUPL выражения, то **`(apair e1 e2)`** это MUPL выражение (конструктор пары).

- Если **e** - MUPL выражение, то **(fst e)** это MUPL выражение (получение первого элемента пары).
- Если **e** - MUPL выражение, то **(snd e)** это MUPL выражение (получение второго элемента пары).
- **(aunit)** это MUPL выражение (не содержит данных). Обратите внимание, что **(aunit)** это MUPL выражение, а **aunit** - нет.
- Если **e** - MUPL выражение, то **(isaunit e)** это MUPL выражение (тестирование на **(aunit)**).
- **(closure env f)** это MUPL значение, где **f** - MUPL функция (выражение полученное с помощью **fun**, а **env** - это окружение, связывающее переменные с их значениями.

MUPL значение - это MUPL целочисленная константа, MUPL замыкание, MUPL aunit, или MUPL пара из MUPL значений. Аналогично Racket, мы можем сконструировать MUPL списки из вложенных MUPL пар, заканчивающихся aunit.

Считайте, что программы на MUPL синтаксически корректны (не содержат выражений вроде **(int (int 42))**. Однако, в MUPL программах могут ошибки типов, например, **(fst (int 7))**. В данном случае нужно вывести сообщение об ошибке.

Задания:

1. (+1) Разминка
    a. Напишите Racket функцию **racketlist->mupllist**, которая принимает Racket список (состоящий из MULP значений, но это не должно влиять на решение), и возвращает аналогичный MUPL список (с теми же значениями, в том же порядке).
    b. Напишите Racket функцию **mupllist->racketlist**, которая принимает MUPL список (состоящий из MULP значений, но это не должно влиять на решение), и возвращает аналогичный Racket список (с теми же значениями, в том же порядке).
2. (+8) Реализация языка MUPL: напишите интерпретатор MUPL, то есть Racket функцию **eval-exp**, которая принимает MUPL выражение e, и либо возвращает MUPL значение, в которое вычисляется e в пустом окружении, либо генерирует **Racket**

`error`, если выражение вызывает ошибку типов, либо обращается к переменной, которая не находится в текущем окружении.

Выражения MUPL, как обычно, вычисляются в окружении (для обращений к переменным). В вашем интерпретаторе используйте Racket список из Racket пар для того, чтобы хранить текущее окружение (изначально пустое). Семантика MUPL выражений следующая:

a. Все значения (values), включая замыкания, вычисляются сами в себя. Например, **`(eval-exp (int 17))`** вычисляется в **`(int 17)`**, не в 17.
b. Переменная вычисляется в значение, привязанное к ней в текущем окружении.
c. Сложение вычисляет оба подвыражения, и, если они оба возвращают **`int`**, возвращает **`int`** из суммы их значений.
d. Функция вычисляется в замыкание, которое содержит саму функцию и текущее окружение (lexical scope)
e. Выражение **`ifgreater`** вычисляет первые два подвыражения в значения **`v1`** и **`v2`**. Если оба значения - целые числа, вычисляется третье подвыражение, если **`v2 > v1`**, иначе вычисляется четвёртое подвыражение.
f. Выражение **`mlet`** вычисляет своё первое подвыражение в значение v. Затем вычисляется второе подвыражение в окружении, дополненном привязкой **`v`** к имени **`s`** (окружение дополняется парой **`(s . v)`**)
g. **`call`** вычисляет свои первое и второе подвыражение в значения **`v1`** и **`v2`**. Если **`v1`** - не замыкание, то это ошибка типов. Иначе, вычисляется тело функции из замыкания в окружении, расширенном связыванием имени функции и замыкания (если имя не **`#f`**) и имени аргумента и его значения **`v2`**.
h. Выражение **`apair`** вычисляет свои подвыражения и конструирует новую пару из результатов.
i. Выражение **`fst`** вычисляет своё подвыражение. Если результат - пара, то результатом всего выражения **`fst`** будет первый элемент этой пары.

j. Выражение **snd** вычисляет своё подвыражение. Если результат - пара, то результатом всего выражения **snd** будет второй элемент этой пары.
k. Выражение **isaunit** вычисляет своё подвыражение. Если его результат - выражение **aunit**, то результатом всего выражения **isaunit** будет MUPL выражение **(int 1)**, иначе - **(int 0)**.

3. (+4) Расширение языка: язык MUPL маленький и простой, но можно писать Racket функции, которые будут вести себя как макросы, что предоставит пользователю языка MUPL дополнительные возможности. Эти Racket функции сгенерируют MUPL выражения, которые затем можно будет встроить в более крупные MUPL выражения или передать в **eval-exp**. Пожалуйста, не используйте в этом задании выражение closure (это служебный тип значений). Кроме того, не используйте **eval-exp** (в этом задании программы нужно только генерировать, а не исполнять).
   a. Напишите Racket функцию **ifaunit**, которая принимает MUPL выражения **e1, e2, e3**, и возвращает MUPL выражение, которое сначала вычисляет **e1**, и если его результат - **aunit**, то вычисляется **e2**, иначе - **e**3.
   b. Напишите Racket функцию **mlet***, которая принимает Racket список пар **'((s1 . e1), (s2 . e2)** … **(sn . en))** и MUPL выражение $\mathbf{e}_{n+1}$. В каждой паре $\mathbf{s}_i$- Racket строка, а $\mathbf{e}_i$ - MUPL выражение. **mlet*** возвращает MUPL выражение, результатом вычисления которого будет результат $\mathbf{e}_{n+1}$, вычисленный в окружении, где каждая строка $\mathbf{s}_i$ связана с результатом соответствующего выражения $\mathbf{e}_i$. Все связывания делаются последовательно, и каждое $\mathbf{e}_i$ вычисляется в окружении, содержащем все предыдущие связывания.
   c. Напишите Racket функцию **ifeq**, которая принимает четыре MUPL выражения **e1, e2, e3** и **e4**, и возвращает MUPL выражение, которое ведёт себя как **ifgreater**, только **e3** вычисляется тогда и только тогда, когда результаты **e1** и **e2** - равные целые числа. Считайте, что никакие аргументы **ifeq** не используют MUPL переменные **_x** и **_y**. В выражении,

которое вернёт **`ifeq`**, **`e1`** и **`e2`** должны вычисляться ровно по одному разу.

4. (+4) Использование языка: выражения на языке MUPL прямо на языке Racket, с помощью struct конструкторов и функций из предыдущего задания.
    a. Привяжите к Racket переменной **`mupl-map`** MUPL функцию, которая ведёт себя как map. Ваша функция должна быть каррированой: принимать MUPL функцию и возвращать MUPL функцию, которая принимает MUPL список и применяет функцию ко всем его элементам, возвращая новый список в качестве результата.
    b. Привяжите к Racket переменной **`mupl-mapAddN`** MUPL функцию, которая принимает MUPL **`int i`** и возвращает MUPL функцию, которая принимает MUPL список из MUPL **`int`** и прибавляет **`i`** к каждому элементу списка. Используйте **`mupl-map`** (**`mlet`** для этой функции уже есть в коде).