Программные проекты

Современные проблемы компьютерных наук

Осенний семестр 2022 г.

Указания

Решение каждого задания нужно проверить на двух или более примерах. Если вы добавляете форму в качестве синтаксического расширения, следует определить переменную, значением которого является форма, которую вы выражаете через другие средства языка. Также следует определить переменную, значением которой является законченное выражение-пример, которое можно подать на вход функции `meval`. Например:

```
> (define for-test1
   '(for [i (cdr '(0 1 2 3))]
      (set! sum (+ sum i))
      (displayln sum)))
> (define for-test2 `(let ([sum 0]) ,for-test1 sum))
> (transform-for for-test1)
(let ()
  (define (go lst)
    (unless (null? lst)
      (let ((i (car lst)))
        (set! sum (+ sum i))
        (displayln sum)
        (go (cdr lst)))))
  (go (cdr '(0 1 2 3))))

> (meval for-test2)
1
3
6
6
```

Если вы добавляете форму в ядро интерпретатора, то достаточно определить выражение-пример.

При определении синтаксического расширения можно пользоваться квазикавычками. Однако конкретный синтаксис должен быть изолирован от функций, осуществляющих перевод, с помощью функций из первой части программы. Так, во второй части программы при создании выражений нельзя использовать символы `if`, `let` и т.п. Вместо этого нужно пользоваться функциями `make-if` и `make-let`. Также следует избегать

определения переменных, которые могут входить свободно в подвыражения обрабатываемого терма. Например, если бы в теле цикла `for` выше использовалась переменная `go`, определенная где-то в окружающем контексте, то определение рекурсивной функции `go` скрыло бы эту переменную. Поэтому для порождения имен вспомогательных функций следует пользоваться функцией `gensym` по аналогии с файлом `eval-for.rkt`.

Функции, которые вычисляют значения особых форм и которые вызываются из `m-eval`, в качестве аргумента получают все выражение целиком. Например, при вычислении выражения вида `(if pred thn els)` функция `eval-if` получает все это выражение, включая символ `if`, а не подвыражения `pred`, `thn` и `els` по отдельности.

Если вы устраиваете рекурсию по выражению, его подвыражения должны иметь тот же вид, что и выражение целиком. Например, неправильно делать рекурсию по списку условий в выражении `(cond clause1 clause2 ...)` и в качестве аргумента передавать функции все выражение, включая символ `cond`. В этом случае нужно написать вспомогательную рекурсивную функцию (желательно, с помощью локального `define`) и подавать ей на вход только список условий.

Практически никогда не имеет смысл писать `(eq? exp #t)` или `(eq? exp #f)`. Вместо этого следует писать `exp` или `(not exp)`. Следует активно пользоваться формами `and` и `or`. Например, вместо `(if pred pred alt)` лучше написать `(or pred alt)`.

Код, написанный вами, следует выделять заметными комментариями, как это сделано в `eval-for.rkt`.

Задания

1. Стандартная модель вычисления лямбда-выражений с окружениями описана в лекциях 7 и 8. Рассмотрим другие правила, при которых вычисление лямбда-абстракции f в окружении e возвращает не замыкание (f, e), а само f; при этом окружение e теряется. Такое правило называется правилом вычисления с динамической областью видимости, в отличие от стандартного правила, которое называются правилом со статической, или лексической, областью видимости. Что касается правила для аппликаций, то применение лямбда-выражения f к фактическим параметрам $a_1, \ldots, a_n$ в окружении e вычисляется как и раньше, но используется само окружение e, а не окружение, взятое из замыкания. То есть вычисляются значения $v_1, \ldots, v_n$ аргументов $a_1, \ldots, a_n$, создается новый кадр, где формальные параметры лямбда-выражения f связываются с $v_1, \ldots, v_n$, этот кадр присоединяется к окружению e, и в новом окружении вычисляется тело f.

1) Объясните, каковы будет значения выражений на слайдах 7–9 в лекции 6 при вычислении с динамической областью видимости.
2) Измените функции `make-procedure`, `m-eval` и `m-apply` так, чтобы они реализовывали вычисление с динамической областью видимости.
3) В обычном лямбда-исчислении значение терма не меняется при переименовании связанных переменных. Приведите пример двух термов, отличающихся только именами связанных переменных и возвращающих разные значения в интерпретаторе с динамической областью видимости.

4) Интерпретатор с динамической областью видимости позволяет написать рекурсивную программу без `define`, `letrec` или присваивания. Напишите функцию, вычисляющую факториал в таком интерпретаторе, используя трюк, похожий на «узел Ландина» (задание 9).

2. Особая форма `fix` имеет вид `(fix (f params) body)`. Она определяет рекурсивную функцию `f` с формальными параметрами `params` и телом `body`. При этом переменная `f` может использоваться в `body`. Например, значением выражения

```
((fix (fact x) (if (= x 0) 1 (* x (fact (- x 1))))) 5)
```

является 120.

Определите функцию `(fix? exp)`, которая проверяет является ли выражение особой формой `fix`, а также функции `fix-name`, `fix-params` и `fix-body`, которые возвращают, соответственно, имя функции, список ее формальных параметров и ее тело. Как и для особой формы `lambda`, тело является списком выражений.

Обработка `fix` в `m-eval` вызывает функцию `(make-rec-procedure name params body env)`, аналогичную `make-procedure`. Эта функция также создает замыкание, но в отличие от `make-procedure` она добавляет к окружению `env` кадр с переменной `name`, а затем использует присваивание для того, чтобы сделать полученной замыкание значением `name`. Этого можно добиться следующим образом. (Для реализации последовательности шагов ниже удобно использовать `let` или `let*`.)

1) С помощью функции `make-binding` создать связывание переменной `name` с произвольным значением, например, `#f`.
2) Превратить это связывание в кадр с помощью функции `make-frame-from-bindings`.
3) Добавить этот кадр к окружению `env`.
4) С помощью `make-procedure` сформировать замыкание из `params`, `body` и полученного окружения.
5) С помощью присваивания заменить значение переменой `name` в связывании из пункта 1 на созданное замыкание и вернуть это замыкание.

Особая форма `let` не позволяет в правой части определения использовать идентификатор из левой части. В этом случае вместо `let` нужно использовать `letrec` (см. лекцию 5). Объясните, почему

```
(letrec ([f1 (lambda params1 body1)]
         ...
         [fn (lambda paramsn bodyn)])
  body)
```

эквивалентно следующему выражению.

```
(let ([f1 (fix f1 params1 body1)]
      ...
      [fn (fix fn paramsn bodyn)])
  body)
```

Используя этот факт, добавьте `letrec` в интерпретатор в качестве синтаксического расширения. Проверьте вашу реализацию и `fix`, и `letrec` на примерах.

3. Другая реализация особой формы `fix` из задания 2 состоит в изменении понятия замыкания. Значением выражение `(fix (f params) body)` в окружении `env` становится четверка `(f, params, body, env)`. Значением нерекурсивной функции `(lambda (params) body)` в окружении `env` является `(placeholder, params, body, env)`, где значением переменной `placeholder` является идентификатор, который не используется в теле функций. Например, можно дать следующее определение.

```
(define placeholder '___rec_func_name___)
```

Вычисление аппликации `(rator rands)` в окружении `env` состоит из следующих шагов (ср. слайд 11 в лекции 8).

1) Вычислить значение `rator` в `env`; оно должно быть замыканием `c = (f, params, body, env1)`.
2) Вычислить значения фактических параметров `rands` в `env`; обозначим результаты через `args`.
3) Создать новый кадр, где переменная `f` связана с замыканием `c`, а формальные параметры `params` — с фактическими параметрами `args`.
4) В качестве объемлющего окружения указать `env1` (из замыкания).
5) В полученном окружении вычислить `body`.

Реализуйте вычисление `fix` и добавьте `letrec` в интерпретатор в качестве синтаксического расширения аналогично заданию 2. Проверьте вашу реализацию и `fix`, и `letrec` на примерах.

4. Еще одна реализация особой формы `fix` из задания 2 состоит введении понятия *задумки* (thunk), аналогичного понятию замыкания. Вычислении выражение `(fix (f params) body)` в окружении `env` состоит из следующих шагов.

1) Сформировать задумку `t = (f, params, body, env)`.
2) Создать новый кадр, где переменная `f` связана с `t`.
3) В качестве объемлющего окружения указать `env`.
4) В полученном окружении вычислить выражение `(lambda (params) body)`.

Изменения также требуются в части `m-eval`, отвечающей за вычисление переменной `x` в окружении `env`. Если `x` в `env` связана с задумкой `t = (f, params, body, env1)`, то значением `x` является результат вычисления `t` в окружении `env1`.

Реализуйте вычисление `fix` и добавьте `letrec` в интерпретатор в качестве синтаксического расширения аналогично заданию 2. Проверьте вашу реализацию и `fix`, и `letrec` на примерах.

5. Добавьте особую форму `fix` из задания 2 в качестве синтаксического расширения. Выражение `(fix (f x) body)` транслируется в `(z (lambda (f) (lambda (x) body)))`, где `z` — так называемый комбинатор неподвижной точки, обозначенный через `fix` в [1, с. 68]. Можно ли таким образом определить рекурсивную функции более, чем одного аргумента?

6. Особая форма `letrec` описана в лекции 5. Добавьте ее в ядро интерпретатора аналогично заданию 2.

Вычисление выражения

```
(letrec ([f1 (lambda params1 body1)]
         ...
         [fn (lambda paramsn bodyn)])
  body)
```

в окружении `env` состоит из следующих шагов.

1) Создать рекурсивные замыкания `(fi paramsi bodyi env)`, как описано в задании 2.
2) Создать новый кадр, где переменные `fi` связаны с этими замыканиями. В качестве объемлющего окружения указать `env`.
3) В полученном окружении вычислить `body`.

7. Добавьте `letrec` в ядро интерпретатора аналогично заданиям 6 и 3. Вычисление выражения

```
(letrec ([f1 (lambda params1 body1)]
         ...
         [fn (lambda paramsn bodyn)])
  body)
```

в окружении `env` приводит к созданию нового кадра, где переменные `fi` связаны с замыканиями `(fi paramsi bodyi env)`. Новый кадр добавляется к `env`, и в этом окружении вычисляется `body`. Необходимо реализовать аппликацию замыкания к аргументам, как описано в задании 3. Также надо изменить вычисление лямбда-выражения, чтобы его значением было замыкание из четырех элементов.

8. Добавьте `letrec` в ядро интерпретатора аналогично заданиям 6 и 4. Вычисление выражения

```
(letrec ([f1 (lambda params1 body1)]
         ...
         [fn (lambda paramsn bodyn)])
  body)
```

в окружении `env` состоит из следующих шагов.

1) Для каждого `i` сформировать задумку `ti = (fi, paramsi, bodyi, env)`.
2) Создать кадр, где переменные `fi` связаны с задумками `ti`. В качестве объемлющего окружения указать `env`.
3) В полученном окружении вычислить выражения `(lambda paramsi bodyi)`. Обозначим их значения через `vi`.
4) Создать кадр, где переменные `fi` связаны со значениями `vi`. В качестве объемлющего окружения указать `env`.
5) В полученном окружении вычислить `body`.

Также нужно изменить вычисление переменных, как описано в задании 4.

9. Используя присваивание, можно написать рекурсивную функцию, даже если в правой части `define` или `let` нельзя использовать функцию из левой части. Следующий трюк, используемый при определении факториала, называется «узел Ландина» в честь Питера Ландина (1930–2009), британского специалиста по компьютерным наукам, пионера в области функционального программирования.

```
(define fact
  (let ([f (lambda (x) 42)])
    (let ([g (lambda (n)
               (if (= n 0) 1 (* n (f (sub1 n)))))])
      (set! f g)
      g)))
```

Нарисуйте диаграмму окружений и объясните, как работает функция `fact`.

Напишите функцию `(letrec->let exp)`, которая использует «узел Ландина» для сведения `letrec` к `let`. Эта функция переводит выражение

```
(letrec ([x1 exp1]
         ...
         [xn expn])
  body)
```

в

```
(let ([x1 'undefined]
      ...
      [xn 'undefined])
  (let ([y1 exp1]
        ...
        [yn expn])
    (set! x1 y1)
    ...
    (set! xn yn)
    body))
```

Здесь `y1`, ..., `yn` — новые переменные, а `body` может состоять более, чем из одного выражения. Добавьте форму `letrec` к интерпретатору в качестве синтаксического расширения и проверьте его работу на примере из лекции 5.

10. Функцию `letrec->let` из задания 9 можно упростить следующим образом. Она переводит

```
(letrec ([x1 exp1]
         ...
         [xn expn])
  body)
```

в

```
(let ([x1 'undefined]
      ...
      [xn 'undefined])
  (set! x1 exp1)
  ...
  (set! xn expn)
  body)
```

Здесь `body` может состоять более, чем из одного выражения. Измените определение факториала в задании 9 в соответствии с описанным правилом. Нарисуйте диаграмму окружений и объясните, как работает полученная функция. Добавьте форму `letrec` к интерпретатору в качестве синтаксического расширения и проверьте его работу на примере из лекции 5.

11. Выражение

```
(while test
  exp1
  ...
  expn)
```

проверяет, истинно ли значение `test`. Если оно ложно (то есть равно `#f`), то вычисление заканчивается с значением `#void`. Если значение `test` истинно (отлично от `#f`), последовательно вычисляются `exp1`, ..., `expn`, после чего снова проверяется `test` и т.д.

Напишите функцию `transform-while`, переводящую `while` в следующее выражение.

```
(let ()
  (define (loop)
    (if (not test)
        (begin
          exp1
          ...
          expn
          (loop))
        (void)))
  (loop))
```

Добавьте форму `while` к интерпретатору в качестве синтаксического расширения.

12. Выражение

```
(until
  exp1
  ...
  expn
  test)
```

последовательно вычисляются `exp1`, ..., `expn`, после чего вычисляет значение `test`. Если оно ложно (то есть равно `#f`), то снова вычисляются значения `exp1`, ..., `expn` и т.д. Если значение `test` истинно (отлично от `#f`), вычисление заканчивается с неопределенным значением `#void`. Проверьте работу `until` на примерах.

Напишите функцию `transform-until`, переводящую особую форму `until` в следующее выражение.

```
(let ()
  (define (loop)
    exp1
    ...
    expn
```

```
      (if test (void) (loop)))
    (loop))
```

Добавьте форму `until` к интерпретатору в качестве синтаксического расширения.

13. Напишите функцию `(let*->let exp)`, которая превращает выражение `exp` вида

```
(let* ([x1 exp1]
       ...
       [xn expn])
  body)
```

в следующее выражение.

```
(let ([x1 exp1])
  ...
    (let ([xn expn])
      body)...)
```

Здесь `body` может состоять более, чем из одного выражения. См. лекцию 5 о `let` и `let*`. Добавьте форму `let*` к интерпретатору в качестве синтаксического расширения и проверьте его работу на примерах, в том числе на тех, которые выдают разные значения на `let` и `let*`.

14. В Racket есть две дополнительные формы вариантов в форме `cond`: `[test-expr => proc-expr]` и `[test-expr]`. Если очередной вариант `cond` имеет вид `[test-expr => proc-expr]`, то вычисляется `test-expr`. Если его значение `v` отлично от `#f`, то вычисляется `proc-expr`. Его значением должна быть функция `f` с одним аргументом. Значением всего выражения `cond` становится результат применения `f` к `v`.

Если очередное предложение `cond` имеет вид `[test-expr]`, то результат вычисления `test-expr` становится значением всего выражения `cond`.

Например, функция `(member x lst)` находит первый элемент списка `lst`, равный `x` в смысле `equal?`. В этом случае она возвращает хвост `lst`, начиная с этого элемента. Если такого элемента нет, возвращается `#f`. Поэтому имеют место следующие вычисления.

```
> (let ([lst '(1 2 3)])
    (cond
      [(member 4 lst) (display 4) 4]
      [(member 2 '(1 2 3)) => (lambda (l) (map - l))]))
'(-2 -3)

> (cond
    [(member 2 '(1 2 3))])
'(2 3)
```

Добавьте обработку этого расширенного синтаксиса в функцию `cond->if`.

15. Особая форма `cond` реализована в интерпретаторе с помощью трансляции во вложенные формы `if`. Вместо этого добавьте форму `cond` в ядро интерпретатора. Сделайте обработку как обычного, так и расширенного синтаксиса из задания 14.

16. Именованный `let` — это выражение вида

```
(let name ([x1 exp1]
           ...
           [xn expn])
  body)
```

Здесь `name` — это произвольный идентификатор, а `body` может состоять более, чем из одного выражения. Эта форма эквивалентна выражению

```
(let ()
  (define (name x1 ... xn) body)
  (name exp1 ... expn))
```

Таким образом, именной `let` можно использовать для написания рекурсивных функций.

Добавьте обработку именованного `let` в функцию `let->application`.

17. Особая форма `let` реализована в интерпретаторе с помощью трансляции в аппликацию лямбда-абстракции. Вместо этого добавьте форму `let` в ядро интерпретатора. Правило вычисления `let` см. в лекции 8. Сделайте обработку как обычного, так и именованного `let` из задания 16. Проверьте работу интерпретатора на примерах `let` из лекции 5.

18. По аналогии с файлом `eval-for.rkt` добавьте в ядро интерпретатора особую форму `for` с несколькими переменными. Например,

```
(for ([i '(1 2 3)] [j '(a b c)]) (displayln (list i j)))
```

печатает

```
(1 a)
(2 b)
(3 c)
```

Во время цикла переменные параллельно продвигаются каждая по своему списку. Цикл заканчивается, когда заканчивается самый короткий список. Значение каждого выражения в теле цикла игнорируется, и значением всего выражения является `#<void>`. Выражение `(for () body)` эквивалентно `(for ([x '(0)]) body)`, где переменная `x` не входит в `body`. Таким образом, в этом случае `body` выполняется один раз и возвращается `#<void>`.

19. Добавьте к интерпретатору форму `for` из задания 18 в качестве синтаксического расширения.

20. Добавьте в ядро интерпретатора особую форму `for/or`. Она работает аналогично `for` из задания 18, но в отличие от `for` цикл работает до тех пор, пока значением последнего выражения в теле цикла является `#f`. Если значение `v` последнего выражения отлично от `#f`, цикл заканчивается и значением всего выражения становится `v`. Если тело цикла не вычисляется ни одного раза или если его значение равно `#f` в каждой итерации, значением всего выражения становится `#f`.

Примеры:

```
> (for/or ([i '(1 2 3 4 x)]) (> i 3))
#t
> (for/or ([i '(1 2 3 4 x)]) i)
1
```

```
> (for/or ([i '()]) (/ 1 0))
#f
> (for/or ([i '(1 2 3 5)] [j '(1 2 4 5)])
    (displayln (list i j)) (< i j))
(1 1)
(2 2)
(3 4)
#t
```

21. Добавьте к интерпретатору форму `for/or` из задания 20 в качестве синтаксического расширения.

22. Добавьте в ядро интерпретатора особую форму `for/and`. Она работает аналогично `for` из задания 18, но в отличие от `for` цикл работает до тех пор, пока значение последнего выражения в теле цикла отлично от `#f`. Если значение последнего выражения в теле равно `#f`, то значением всего выражения также становится `#f`. Если тело цикла не вычисляется ни одного раза, значением всего выражения является `#t`. В противном случае значением всего выражения становится результат последнего вычисления тела цикла.

Примеры:

```
> (for/and ([i '(4 3 2 1 x)]) (> i 1))
#f
> (for/and ([i '(4 3 2)]) (> i 1))
#t
> (for/and ([i '()]) (/ 1 0))
#t
> (for/and ([i '(1 2 4)] [j '(2 3 4)])
    (displayln (list i j)) (< i j))
(1 2)
(2 3)
(4 4)
#f
```

23. Добавьте к интерпретатору форму `for/and` из задания 22 в качестве синтаксического расширения.

24. Добавьте в ядро интерпретатора особую форму `for/sum`. Она работает аналогично `for` из задания 18, но в отличие от `for` значением всего выражения становится сумма по всем итерациям значений последнего выражения тела цикла.

Примеры:

```
> (for/sum ([i '(1 2 3 4)]) (* i i))
30
> (for/sum ([i '()]) (* i i))
0
> (for/sum ([i '(1 2 3)] [j '(4 5 6)])
    (displayln (list i j)) (* i j))
(1 4)
```

```
(2 5)
(3 6)
32
```

25. Добавьте к интерпретатору форму `for/sum` из задания 24 в качестве синтаксического расширения.

26. Добавьте к ядру интерпретатора форму `(unset! var)`, вычисление которой в некотором окружении делает следующее.

1) Если переменная `var` не была определена, результатом является ошибка.
2) Если `var` была ранее определена с помощью `define` или являлась формальным параметром функции, то ее значением становится то, которое оно имело до последнего `set!`.
3) Если `(unset! var)` вызвать больше раз, чем было вызвано `(set! var ...)`, то это не является ошибкой и значение переменной не меняется.

Такого поведения можно добиться, изменив понятие связывания (binding). В связывании для каждой переменной нужно хранить не последнее значение, а список всех предыдущих значений.

27. Кадры в интерпретаторе реализованы как списки связываний (bindings). Измените реализацию кадров так, чтобы кадр представлял собой тройку `(frame vars values)`, где символ `frame` — это метка, а `vars` и `vals` — это списки переменных и их значений, соответственно.

28. Булевы константы, пары, натуральные числа и операции на них можно представить в виде термов в минимальном лямбда-исчислении, как описано в лекции 6 и в [1, с. 61–65] (представление Чёрча). Напишите функции `nat->church` и `church->nat`, которые преобразуют натуральные числа в числа Чёрча и наоборот. Первая из этих функций пишется непосредственно с помощью рекурсии, а вторая принимает число Чёрча `n`, формирует выражение `(n add1 0)` и вычисляет его значение в глобальном окружении с помощью `m-eval`.

Определите особые формы `(add e1 e2)`, `(sub e1 e2)` и `(mult e1 e2)`. Вычисление `(add e1 e2)` в окружении `env` состоит из следующих шагов.

1) Вычислить значения `e1` и `e2` в `env`.
2) Преобразовать их в числа Чёрча `n1` и `n2`.
3) Вычислить значение выражения `(plus n1 n2)` с помощью функции `church->nat`. Здесь `plus` есть терм в [1, с. 63].

Вычисление `(sub e1 e2)` и `(mult e1 e2)` осуществляется аналогично.

Список литературы

[1] Пирс Б. Типы в языках программирования. М.: Добросвет, Лямбда пресс, 2011