Практика 9. Интерпретаторы и хвостовые вызовы

материалы преподавателя

Calculator

Вопрос 1

Запиши синтаксически верные Calculator-выражения, соответствующие приведённым цепочкам вызовов Pair. Нарисуй диаграмму.

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

Ответ

```
> (+ 1 2 3 4)
```

```
>>>  Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

Ответ

```
> (+ 1 (* 2 3))
```

Вопрос 2

Ответь на следующие вопросы об экземпляре `Pair`, представляющем Calculator-выражение `(+ (- 2 4) 6 8)`:

А. Запиши Python-выражение, которое вернёт экземпляр `Pair` для указанного выше выражения. Нарисуй диаграмму.

Ответ

```
>>> Pair('+', Pair(Pair('-', Pair(2, Pair(4, nil))), Pair(6, Pair(8, nil))))
```

Б. Как отыскать оператор в вызывающем выражении? Если экземпляр `Pair`, созданный чуть выше, связан с именем `p`, то как извлечь из него оператор?

Ответ

```
>>> p.first
```

В. Как отыскать операнды в вызывающем выражении? Если экземпляр `Pair`, созданный чуть выше, связан с именем `p`, то как извлечь из него список операндов? А первый операнд?

Ответ

```
>>> p.rest — список всех операндов, p.rest.first — первый операнд.
```

Вычисление

Вычислительная часть интерпретатора определяет тип выражения и вычисляет его по заданным правилам:

1. **Числа**. Они равны сами себе. Например, выражения `3.14` и `165` имеют значения равные `3.14` и `165` соответственно.
2. **Имена** ищутся в словаре `OPERATORS`. Каждое имя (например, `+`) связано с Python-функцией, выполняющей соответствующую операцию над списком чисел (например, `sum`).
3. **Вызывающие выражения** обрабатываются так:
   a. **Вычисляется** значение оператора (должна получиться функция).
   b. **Вычисляются** значения операндов слева направо.
   c. Значение оператора (функция) **применяется** к значениям операндов.

Функция `calc_eval` принимает Calculator-выражение, представленное Python-объектами и применяет следующие правила:

```
def calc_eval(exp):
    """Вычисление выражения, представленного экземплярами Pair."""
    if isinstance(exp, Pair): # Вызывающее выражение
        fn = calc_eval(exp.first)
        args = list(exp.rest.map(calc_eval))
        return calc_apply(fn, args)
    elif exp in OPERATORS: # Имена
        return OPERATORS[exp]
    else: # Числа
        return exp
```

Заметь, что `calc_eval` рекурсивна! Для обработки вызывающего выражения следует вызвать `calc_eval` для оператора и операндов.

Применение (`calc_apply`) в языке Calculator выглядит довольно просто, поскольку набор процедур прост и ограничен. Этот шаг в интерпретаторе Scheme более развесист из-за возможности использования пользовательских функций.

Передаваемая функция реализует соответствующую операцию над Python-списком чисел. Функция `calc_apply` просто вызывает эту функцию.

```
def calc_apply(fn, args):
    """Применяет fn к списку чисел args."""
    return fn(args)
```

Вопрос 3

Сколько вызовов `calc_eval` и `calc_apply` произойдёт при вычислении следующего выражения?

```
> (+ 2 4 6 8)
```

Ответ

```
6 вызовов calc_eval: один общий и по одному для каждого оператора и операнда
1 вызов calc_apply для вычисления сложения
```

```
> (+ 2 (* 4 (- 6 8)))
```

Ответ

```
10 вызовов calc_eval: один общий и по одному для каждого оператора и операнда
3 вызова calc_apply для вычисления каждого вызывающего выражения
```

Вопрос 4

Представь, что ты хочешь добавить в Calculator действия `<`, `>` и `=`. Они должны работать так же как и в Scheme.

```
> (and (= 1 1) 3)
3
> (and (+ 1 0) (< 1 0) (/ 1 0))
#f
```

А. Возможно ли использовать существующую механику `calc_eval` для обработки выражений операций сравнения (`<`, `>` и `=`). Почему?

Ответ

```
Comparison expressions are regular call expressions, so we need to evaluate the
operator and operands and then apply a function to the arguments. Therefore, we do not
need to change calc eval. We simply need to add new entries to the OPERATORS
dictionary that map "<", ">", and "=" to functions that perform the appropriate
comparison operation.
```

Б. Возможно ли обрабатывать `and`-выражения с существующей реализацией `calc_eval`. Почему?

Ответ

```
Since and is a special form that short circuits on the first false-y operand, we
cannot handle these expressions the same way we handle call expressions. We need to
add special handling for combinations that don't evaluate all the operands.
```

В. Дополни код приведённый ниже для обработки `and`-выражений. Считай, что операторы сравнения (`<`, `>` и `=`) уже реализованы.

```
def calc_eval(exp):
    if isinstance(exp, Pair):

        if _________________________: # and-выражения
            return eval_and(exp.rest)
        else:                        # вызывающие выражения
            return calc_apply(calc_eval(exp.first), list(exp.rest.map(calc_eval)))
    elif exp in OPERATORS:           # имена
        return OPERATORS[exp]
    else:                            # числа
        return exp

def eval_and(operands):
```

Ответ

```
def calc_eval(exp):
    if isinstance(exp, Pair):

        if exp.first == 'and':       # and-выражения
            return eval_and(exp.rest)
        else:                        # вызывающие выражения
            return calc_apply(calc_eval(exp.first), list(exp.rest.map(calc_eval)))
    elif exp in OPERATORS:           # имена
        return OPERATORS[exp]
    else:                            # числа
        return exp

def eval_and(operands):
    curr, val = operands, True
    while curr is not nil:
        val = calc_eval(curr.first)
        if val is False:
            return False
        curr = curr.rest
    return val
```

Оптимизация хвостовой рекурсии

Scheme поддерживает оптимизацию хвостовой рекурсии. Она позволяет создавать рекурсивные функции, требующие константного пространства. **Хвостовой вызов** происходит когда вызов функции — **последнее действие в текущем фрейме**. В случае, если фрейм более не нужен, можно его удалить из памяти. То есть если последнее действие — это вызов функции, то вместо создания нового фрейма можно переиспользовать существующий. Посмотри на код `fact`.

```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

Рекурсивный вызов происходит в последней строке, но это не последнее действие. После выполнения `(fact (- n 1))` процедуре нужно ещё умножить результат на `n`. Таким образом, последним вызовом в `fact` является вызов процедуры перемножения, а не `fact`. В данном случае вызов `fact` **не является** хвостовым вызовом.

Можно переписать эту процедуру с использованием вспомогательной процедуры `fact-tail`, которая будет обеспечивать хранение промежуточного результата в каждом рекурсивном вызове.

```
(define (fact n)
    (define (fact-tail n result)
        (if (= n 0) result
        (fact-tail (- n 1) (* n result))))
    (fact-tail n 1))
```

Процедура `fact-tail` делает единственный рекурсивный вызов `fact-tail` для вычисления последнего выражения. Это хвостовой вызов.

Хвостовой контекст

Для того, чтобы определить, является ли вызов функции хвостовым вызовом, нужно удостовериться, что этот вызов располагается в **хвостовом контексте**.

Если выражение является последним в теле функции, то оно находится в одном из хвостовых контекстов:

- второй или третий операнд `if`-выражения;
- любое подвыражение в `cond`-выражении, кроме предикатов;
- последний операнд в `and` и `or` выражениях;
- последний операнд в `begin`-выражении;
- последний операнд в `let`-выражении.

Например, в выражении `(begin (+ 2 3) (- 2 3) (* 2 3))` вызов `(* 2 3)` является хвостовым.

Вопрос 5

Для каждой из следующих процедур определи является ли рекурсивный вызов хвостовым. Так же определи будет ли процедура требовать постоянного числа фреймов.

А.

```
(define (question-b x y)
    (if (= x 0) y
        (question-b (- x 1) (+ y x))))
```

Ответ

```
The recursive call is the third operand in the if expression, so it is in tail
context. This means that the last expression that will be evaluated in the body of
this function is the recursive function call, so this function uses θ(1) frames.
```

Б.

```
(define (question-c x y)
    (if (> x y)
        (question-c (- y 1) x)
        (question-c (+ x 10) y)))
```

Ответ

```
The recursive calls are the second and third operands of the if expression. Only one
of these calls is actually evaluated, and whichever one it is will be the last
expression evaluated in the body of the function. This function therefore uses Θ(1)
frames.

Note that if you actually try and evaluate this function, it will never terminate. But
at least it won't crash from hitting max recursion depth!
```

В.

```
(define (question-d n)
    (if (question-d n)
        (question-d (- n 1))
        (question-d (+ n 10))))
```

Ответ

```
The second and third recursive calls are in tail context, but the first is not. Since
not all the recursive calls are tail calls, this function does not use a constant
number of frames.
```

Г.

```
(define (question-e n)
    (cond ((= n 0) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

Ответ

```
The second and third recursive calls are the second expressions in a clause, so they
are in tail context. However, the first recursive call is not in tail context. Since
not all recursive calls are tail calls, this function is not tail recursive and does
not use a constant number of frames.
```

Вопрос 6

Напиши процедуру с хвостовой рекурсией, которая принимает Scheme-список и возвращает сумму всех его значений. Считай, что список плоский и содержит только числа.

```
(define (sum lst)
```

Ответ

```
(define (sum lst)
    (define (sum-sofar lst current-sum)
        (if (null? lst)
            current-sum
            (sum-sofar (cdr lst) (+ (car lst) current-sum))))
    (sum-sofar lst 0))
```

Вопрос 7

Напиши процедуру с хвостовой рекурсией, которая возвращает `n`-ое число Фибоначчи. Считай, что `fib(0) = 0` и `fib(1) = 1`

```
(define (fib n)

    (define (fib-sofar ___________________________________________)

            (if ___________________________________________________

                ___________________________________________________

                (fib-sofar ___________________________________________))

    (fib-sofar ____________________________________________________))
```

```
(define (fib n)
    (define (fib-sofar i curr next)
        (if (= i n)
        curr
        (fib-sofar (+ i 1) next (+ curr next)))
    (fib-sofar 0 0 1))
```

Вопрос 8

Напиши рекурсивную процедуру с хвостовой рекурсией, которая принимает число и отсортированный список. Функция должна вернуть сортированную копию списка с числом, вставленным в нужную позицию.

A. Сначала напиши процедуру, которая обращает список.

```
(define (reverse lst)
    (define (reverse-sofar lst lst-sofar)

        (if (null? lst) ____________________________________________________

            ____________________________________________________________________))

    ____________________________________________________________________________)
```

```
(define (reverse lst)
    (define (reverse-sofar lst lst-sofar)
        (if (null? lst)
            lst-sofar
            (reverse-sofar (cdr lst) (cons (car lst) lst-sofar))))
    (reverse-sofar lst nil))
```

Б. Теперь напиши функцию, объединяющую два списка. Вероятно нужно использовать `reverse`.

```
(define (append a b)
    (define (rev-append-tail a b)

        (if (null? a) ____________________________________________________

            ____________________________________________________________________))

    ____________________________________________________________________________)
```

```
(define (append a b)
    (define (rev-append-tail a b)
        (if (null? a)
        b
        (rev-append-tail (cdr a) (cons (car a) b))))
    (rev-append-tail (reverse a) b))
```

В. Наконец, реализуй `insert`. Используй `reverse` и `append`.

```
(define (insert n lst)
    (define (rev-insert lst rev-lst)

        (cond ((null? lst) ____________________________________________________)

            ((> (car lst) n) ____________________________________________________)

            (else ____________________________________________________________)))

    ____________________________________________________________________________)
```

```
(define (insert n lst)
    (define (rev-insert lst rev-lst)
        (cond ((null? lst) (cons n rev-lst))
            ((> (car lst) n) (append (reverse lst) (cons n rev-lst)))
            (else (rev-insert (cdr lst) (cons (car lst) rev-lst)))))
    (reverse (rev-insert lst nil)))
```