Домашнее задание №2

Дедлайн 1 (16 баллов): 20 сентября, 13:00
Дедлайн 2 (8 баллов): 27 сентября, 13:00

Домашнее задание нужно написать на Python 3 и сдать в виде одного файла. Правило именования файла: `name_surname_02.py`. Например, если вас зовут Иван Петров, то имя файла должно быть: `ivan_petrov_02.py`.

Обратите внимание, что при выполнении задания можно использовать только то, что мы уже успели обсудить в рамках курса.

1 В следующих заданиях можно, но не обязательно, использовать анонимные функции.

а Напишите функцию `compose`, которая строит композицию двух переданных ей функций.

```
>>> f = compose(lambda x: x ** 2, lambda x: x + 1)
>>> f(42)
1849
```

б Напишите функцию `constantly`, которая по заданному значению строит константную функцию, то есть функцию, возвращающую одно и то же значение вне зависимости от аргументов.

```
>>> f = constantly(42)
>>> f()
42
>>> f(range(4), range(2), foo="bar")
42
```

в Напишите функцию `flip`, которая принимает функцию `func`. Результатом её работы является функция, применяющая `func` к позиционным аргументам в обратном порядке. Ключевые аргументы передаются без изменений.

```
>>> f = flip(map)
>>> list(f(range(10), range(10), lambda x, y: x ** y))
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
>>> list(map(lambda x, y: x ** y, range(10), range(10)))
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
```

г Напишите функцию `curry`, которая принимает функцию `func`. `curry` должна возвращать новую функцию, у которой часть позиционных аргументов зафиксирована. Эти аргументы также должны быть переданы в функцию `curry`.

```
>>> f = curry(filter, lambda x: x < 5)
>>> list(f(range(10)))
[0, 1, 2, 3, 4]
>>> g = curry(filter, lambda x: x < 5, range(10))
>>> list(g())
[0, 1, 2, 3, 4]
```

2 Следующие задания нужно реализовать либо с помощью `map`, `filter` и `zip`, либо с помощью генераторов списков.

а Напишите функцию `enumerate`, которая перенумеровывает элементы переданной последовательности, начиная со `start`, если он есть, иначе с нуля.

```
>>> list(enumerate("abcd"))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
>>> list(enumerate("abcd", 1))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

б Напишите функцию `which`, которая принимает предикат и последовательность и возвращает индексы элементов последовательности, удовлетворяющих предикату.

```
>>> list(which(lambda x: x % 2 == 0, [4, 9, 15]))
[0]
```

в Напишите функцию `all`, которая принимает предикат и последовательность и проверяет, что все элементы последовательности удовлетворяют предикату.

```
>>> all(lambda x: x % 2 == 0, [4, 9, 15])
False
>>> all(lambda x: x % 2 == 0, [])
True
```

г Дополните функцию `all` функцией `any`, которая проверяет, что в последовательности встречается хотя бы один элемент, удовлетворяющий предикату.

```
>>> any(lambda x: x % 2 == 0, [4, 9, 15])
True
>>> any(lambda x: x % 2 == 0, [])
False
```

Парсер-комбинаторы и S-выражения

Парсером называется часть программы, которая преобразует последовательность простых объектов в более сложную структуру, неявно содержащуюся во входной последовательности, например, преобразует исходный код в абстрактное синтаксическое дерево.

Парсер-комбинаторы — популярная техника создания парсеров, суть которой заключается в построении парсеров путём комбинирования других парсеров.

Мы будем представлять парсер как функцию, принимающую на вход строку и возвращающую тройку вида `(tag, result, leftover)`, где

- `tag` — `OK`, если парсер принял переданную ему строку, или `ERROR`, если произошла ошибка;
- `result` содержит результат работы парсера или текст собщения об ошибке;
- `leftover` — это входная строка без изменений, если произошла ошибка, ли часть строки, не использованная парсером, если ошибки не было.

В качестве примера рассмотрим реализацию парсера `dot`, который ожидает, что первым символом входной строки будет `"."`.

```
# Глобальные константы для tag.
OK, ERROR = "OK", "ERROR"


def dot(input):
    if not input:
        return ERROR, "eof", input
    elif input[0] != ".":
        return ERROR, "expected . got " + input[0], input
    else:
        return OK, ".", input[1:]
```

Попробуем парсер `dot` в деле:

```
>>> dot("...")
('OK', '.', '..')
>>> dot("!..")
('ERROR', 'expected . got !', '!..')
>>> dot("")
('ERROR', 'eof', '')
```

В этом задании мы реализуем классический вариант парсер-кобинаторов и применим его для разбора арифметических S-выражений.

S-выражения (*англ.* symbolic expressions) представляют из себя выражения вида

```
(op arg1 arg2 ...),
```

где `op` — символ операции, например, `+` или `*`, а `arg1 arg2 ...` — последовательность аргументов, которые могут быть числами или другими S-выражениями.

Пример S-выражения для $42 + 2 * 4$:

```
(+ 42 (* 2 4))
```

а Обобщим логику парсера `dot`, реализовав парсер-комбинатор `char`, который принимает символ и возвращает парсер, ожидающий указанный символ в начале входной строки.

```
def char(ch):
    def inner(input):
        if not input:
            return ERROR, "eof", input
        elif input[0] != ch:
            return ERROR, "expected " + ch + " got " + input[0], input
        else:
            return OK, ch, input[1:]
    return inner
```

Посмотрим, что получилось:

```
>>> p = char("(")
>>> p("()")
('OK', '(', ')')
```

Реализацию `char` можно также обобщить на случай произвольного числа символов. Напишите парсер-комбинатор `any_of`, который принимает произвольное количество символов в виде строки и возвращает парсер, ожидающий любой из указанных символов во входной строке.

```
>>> p = any_of("()")
>>> p("(")
('OK', '(', '')
>>> p(")")
('OK', ')', '')
>>> p("[]")
('ERROR', 'expected any of () got [', '[]')
```

б Реализуйте комбинатор `chain`. Он принимает на вход произвольное количество других парсеров и возвращает новый парсер, являющийся их композицией. Если хотя бы один из парсеров в композиции вернул ошибку, то нужно вернуть её в качестве результата всей композиции.

```
>>> p = chain(char("("), char(")"))
>>> p("()")
('OK', ['(', ')'], '')
>>> p("(")
('ERROR', 'eof', '(')
```

в Следующий полезный комбинатор называется `choice`. Он принимает произвольное количество других парсеров и возвращает новый парсер, который применяет переданные парсеры в заданном порядке и возвращает результат, как только один из них принял входную строку, то есть вернул `tag == OK`.

```
>>> p = choice(char("."), char("!"))
>>> p(".")
('OK', '.', '')
>>> p("!")
('OK', '!', '')
>>> p("?")
('ERROR', 'none matched', '?')
```

г Реализуйте комбинатор `many`, который принимает парсер и возвращает новый парсер, применяющий переданный пока это возможно: ноль или более раз.

```
>>> p = many(char("."))
>>> p("...?!")
('OK', ['.', '.', '.'], '?!')
>>> p(".")
('OK', ['.'], '')
>>> p("I have no idea what this is.")
('OK', [], 'I have no idea what this is.')
```

д Добавьте к комбинатору `many` ключевой аргумент `empty` (по умолчанию **`True`**) и измените логику возвращаемого парсера таким образом, что если `empty` установлен в **`False`** и переданный парсер не принимает входную строку, то результатом является ошибка переданного парсера.

```
>>> p = many(char("."), empty=False)
>>> p(".!")
('OK', ['.'], '!')
>>> p("!!")
('ERROR', 'expected . got !', '!!')
>>> p("")
('ERROR', 'eof', '')
```

е Реализуйте парсер-комбинатор `skip`, который заменяет результат переданного ему парсера на **`None`**. Если в переданном парсере произошла ошибка, её нужно вернуть без изменений.

```
>>> p = skip(many(char("."), empty=False))
>>> p("...")
('OK', None, '')
>>> p("")
('ERROR', 'eof', '')
```

ё Выразите в терминах реализованных парсер-комбинаторов комбинатор `sep_by`. Комбинатор принимает два парсера: `p` и `separator` — и возвращает новый парсер. Этот парсер применяет ко входной строке парсер `p`, затем, если не было ошибки, композицию парсеров `separator` и `p` и так далее. Обратите внимание, что последовательность парсеров всегда заканчивается на `p`.

```
>>> p = sep_by(any_of("1234567890"), char(","))
>>> p("1,2,3")
('OK', ['1', '2', '3'], '')
>>> p("1")
('OK', ['1'], '')
>>> p("")
('ERROR', 'eof', '')
```

Вам может быть полезна функция `transform`, преобразующая результат работы парсера, если он принял входную строку.

```
def transform(p, f):
    def inner(input):
        tag, res, leftover = p(input)
        return tag, f(res) if tag == OK else res, leftover
    return inner
```

ж Реализуйте функцию `parse`, которая принимает парсер и строку, и, если парсер принял **весь** вход, возвращает результат работы парсера, иначе падает с ошибкой.

```
>>> p = sep_by(any_of("1234567890"), char(","))
>>> parse(p, "1,2,3")
['1', '2', '3']
>>> parse(p, "...")
Traceback (most recent call last):
  # ...
AssertionError: ('expected any of 1234567890 got .', '...')
>>> parse(p, "1!")
Traceback (most recent call last):
  # ...
AssertionError: (['1'], '!')
#                ^ res,  ^ leftover
```

Вам будет полезен оператор **`assert`**, который принимает условие и произвольное выражение и поднимает **`AssertionError`**, если условие — *falsy* значение:

```
>>> assert g.is_connected(), ("graph is disconnected", g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: ('graph is disconnected', [...])
```

Всё готово для того, чтобы построить парсер арифметических S-выражений. Сделаем это:

```
lparen, rparen = skip(char("(")), skip(char(")"))
ws = skip(many(any_of(" \r\n\t"), empty=False))
number = transform(many(any_of("1234567890"), empty=False),
                   lambda digits: int("".join(digits)))
op = any_of("+-*/")

def sexp(input):
    args = sep_by(choice(number, sexp), ws)

    # Уберём лишние None из результата chain.
    p = chain(lparen, op, skip(ws), args, rparen)
    p = transform(p, lambda res: (res[1], res[3]))
    return p(input)
```

Проверим, что всё работает:

```
>>> parse(sexp, "(+ 42 (* 2 4))")
('+', [42, ('*', [2, 4])])
```

з* Это задание необязательно для выполнения и не оценивается баллами. Вы можете реализовать его исключительно для своего удовольствия.

Реализуйте функцию `eval_sexp`, которая принимает результат разбора S-выражения и вычисляет его результат.

```
>>> eval_sexp(parse(sexp, "(+ 42 (* 2 4))"))
50
```