МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет радиотехники,
электроники и автоматики»
МГТУ МИРЭА

Факультет информационных технологий

Кафедра МОСИТ

Лабораторная работа №1

«Разработка интерпретатора простого императивного языка»

по дисциплине «Теория языков программирования»

Выполнил: студент группы ИТО-1-10 Кулёв П.С.

Преподаватель: Котович Л.Л.

Москва
2014

1 Постановка задачи

Необходимо разработать программу на языке высокого уровня для интерпретирования простого языка. Язык имеет следующие конструкции:

- Присваивание (`a := 4`)
- Ветвление if-then-else
- Цикл while do end.
- Составные операторы, разделённые точкой с запятой. (a := 1; b := 3)

2 Задание

1. Написать лексический анализатор (лексер), который принимает на вход исходный код программы и возвращает список токенов. Токен состоит из выражения на ЯП и тэга(класс токена). Для проверки принадлежности синтаксических конструкций синтаксису языка используется язык регулярных выражений.
2. Написать парсер, который принимает на вход список токенов, превращает его в абстрактное дерево синтаксического разбора(AST) и затем вычисляет последовательно все его узлы. После вычисления всего дерева возвращает результат в виде словаря окружения переменная : значение.
3. Написать интерактивную оболочку, с помощью которой можно выполнять конструкции языка.
4. Покрыть все модули юнит-тестами.

3 Практическая часть

3.1 Разработка лексического анализатора

Для выполнения задания использовалась среда `JetBrains PyCharm Community Edition V3.1` и интерпретатор языка `Python v3.4.0`.

Сначала разработаем лексический анализатор согласно заданию, а затем протестируем его.

Для лексера нам потребуются стандартные модули:

- `re` — библиотека для работы с регулярными выражениями

- `sys` — библиотека для работы с интерпретатором `Python`

Подключим их:

```
import re
import sys\\
```

Определим основную (и единственную) функцию, которая будет делать всю работу (`lex(source, token_exprs)`), установим начальные значения для текущей позиции, списка токенов, которые мы будем заполнять и вернём в конце работы функции, а также позицию конца строки:

```
def lex(source, token_exprs):
    pos = 0
    tokens = []
    end = len(source)
```

Лексер по очереди применяет каждое определённое нами регулярное выражение к тексту начиная с текущей позиции `pos`.

Если есть совпадение, то лексер запоминает текст, который совпал (группа) и если в принятом наборе правил для этого выражения есть тэг, то лексер записывает тег и выражение в результирующий список; если совпадения нет, то лексер выдаёт ошибку и записывает в `stderr` сообщение `Illegal character` с указанием символа вызвавшего ошибку и завершает работу с кодом ошибки 1.

После успешного распознавания выражения счётчику текущей позиции присваивается позиция конца распознанной строки.

После обработки исходного текста функция возвращает список токенов.

Полный код лексера:

```
import re
import sys


def lex(source, token_exprs):
    pos = 0
    tokens = []
    end = len(source)
    while pos < end:
        match = None
        for token_expr in token_exprs:
            pattern, tag = token_expr
            regex = re.compile(pattern)
            match = regex.match(source, pos)
            if match:
                text = match.group(0)
                if tag:
```

```
                token = (text, tag)
                tokens.append(token)
            break
    if not match:
        sys.stderr.write('Illegal character {s}'.format(
                                                source[pos]))
        sys.exit(1)
    else:
        pos = match.end(0)
return tokens
```

3.2 Разработка юнит-тестов

Создадим новый Python файл, в котором определим базовые классы токенов и соответствия синтаксических конструкций классам токенов, а так же тестовые случаи для лексера.

Использовать будем модуль `unittest` из стандартной библиотеки.

Определим базовые классы:

- `KEYWORD` — для служебных слов языка
- `INT` — для целых чисел
- `ID` — для идентификаторов переменных

Определим с помощью регулярных выражений выражения языка, соответствующие нашим классам токенов:

```
token_exprs = [
    (r'[ \t\n]+', None),
    (r'#[^\n]*', None),
    (r'keyword', KEYWORD),
    (r'[0-9]+', INT),
    (r'[A-Za-z][A-Za-z0-9_]*', ID)
]
```

Первое регулярное выражение определяет пробелы, табы и переносы строки. Класс токена не указан, поэтому будет просто игнорироваться лексером и не попадёт в парсер.

Второе выражение определяет комментарии, которые также не попадут в парсер.

Для того чтобы сделать новый юнит-тест, унаследуем класс `unittest.TestCase` и опишем в нём методы, начинающиеся с `_test`, в которых будем вызвать метафункцию `lexer_test`, вызывающую функцию `lex` с переданными ей параметрами.

Листинг:

```
import unittest
from lexer import *

KEYWORD = 'KEYWORD'
INT = 'INT'
ID = 'ID'

token_exprs = [
    (r'[ \t\n]+', None),
    (r'#[^\n]*', None),
    (r'keyword', KEYWORD),
    (r'[0-9]+', INT),
    (r'[A-Za-z][A-Za-z0-9_]*', ID)
]


class TestLexer(unittest.TestCase):
    def lexer_test(self, code, expected):
        actual = lex(code, token_exprs)
        self.assertEquals(expected, actual)

    def test_empty(self):
        self.lexer_test('', [])

    def test_id(self):
        self.lexer_test('abc', [('abc', ID)])

    def test_keyword_first(self):
        self.lexer_test('keyword', [('keyword', KEYWORD)])

    def test_space(self):
        self.lexer_test(' ', [])

    def test_id_space(self):
        self.lexer_test('abc def', [('abc', ID), ('def', ID)])


if __name__ == '__main__':
    test_names = ['test_lexer']
    suite = unittest.defaultTestLoader.loadTestsFromNames(test_names)
    result = unittest.TextTestRunner().run(suite)
```

После описания всех тестовых случаев запустим их на выполнение и посмотрим результат.

```
┌─most@fezoo  ~/PycharmProjects/yasli/IML  ‹master*›
└─$ python3 test_lexer.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

Все тесты завершены успешно.
Попробуем сымитировать ошибку изменив `self.lexer_test('', [])` на `self.lexer_test('vfds fds ', [])`

Посмотрим результат:

```
─most@fezoo  ~/PycharmProjects/yasli/IML  ‹master*›
└─$ python3 test_lexer.py
....F
======================================================================
FAIL: test_space (test_lexer.TestLexer)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/most/PycharmProjects/yasli/IML/test_lexer.py", line 35ace
    self.lexer_test('vfds fds ', [])
  File "/home/most/PycharmProjects/yasli/IML/test_lexer.py", line 23est
    self.assertEquals(expected, actual)
AssertionError: Lists differ: [] != [('vfds', 'ID'), ('fds', 'ID')..

Second list contains 2 additional elements.
First extra element 0:
('vfds', 'ID')

- []
+ [('vfds', 'ID'), ('fds', 'ID')]

----------------------------------------------------------------------
Ran 5 tests in 0.002s

FAILED (failures=1)
```

По выводу видно где произошла ошибка. Вместо ожидаемого пустого списка (пробелы опускаются) из лексера вернулся список из двух токенов. К счастью, очевидно, что это не ошибка лексера.

Есть утилита `coverage` для оценки покрытия `Python` кода тестами. Проверим покрытие:

```
┌─most@fezoo  ~/PycharmProjects/yasli/IML  ‹master*›
└─$ python3 test_lexer.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK
```

3.3 Работа с интерпретатором

Далее были разработаны парсер и интерактивная оболочка.

Вызов интерпретатора без параметров предоставляет пользователю командную строку:

```
$: ./iml.py
IML interpreter

==> 1 + 5 + 4
1 + 5 + 4 <class 'str'>
[('1', 'INT'), ('+', 'RESERVED'), ('5', 'INT'), ('+', 'RESERVED'), ('4', 'INT')]
10
==> 1 * 5 - 2
1 * 5 - 2 <class 'str'>
[('1', 'INT'), ('*', 'RESERVED'), ('5', 'INT'), ('-', 'RESERVED'), ('2', 'INT')]
3
==>
```

Если передать интерпретатору в качестве параметра имя файла, содержащего программу написанную на языке IML, то интерпретатор выполнит эту программу и в конце выведет пользователю словарь окружения.

Например, файл с названием factorial.iml:

```
a := 1;
b := 2;
while a < 5 do
    b := a;
    a := a + 1
```

```
end
```

Выполняется так:

```
$ python3 iml.py temp.iml
2014-06-21 13:01:16.887964 Final variable values:
2014-06-21 13:01:16.888059 a: 5
2014-06-21 13:01:16.888087 b: 4
```