Содержание

[1. Постановка задачи 3](#_Toc4172825)

[2. Теория 4](#_Toc4172826)

[3. Результат работы анализатора 6](#_Toc4172827)

[Выводы 8](#_Toc4172828)

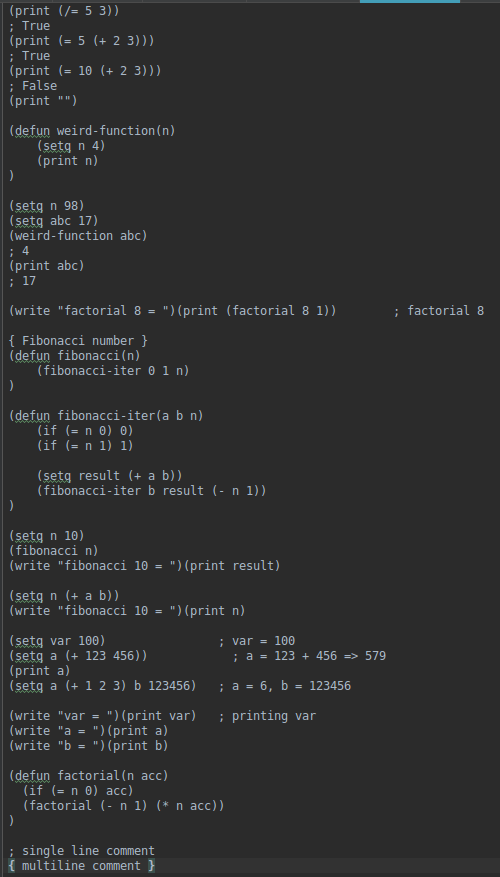
[Приложение А. - Исходный код анализируемой программы 9](#_Toc4172829)

[Приложение Б. - Исходный код анализатора](#_Toc4172829) 11

## **Постановка задачи**

В данной работе ставится задача исследования области семантических ошибок, подробному изучению теории семантики: приведение типов, операции с различными типами. Основной целью работы является семантика, то есть возможность интерпретатора распознавать типы. Это является следующим шагом анализа текста программы – семантический, существенно отличающийся от двух предыдущих – лексического и синтаксического, и дополнена файлом vm.py, где содержатся основные возможности работы с основными операторами и методами языка Lisp. Таким образом, программа выполнит 3 фазу – выполнение привидения типов и затронет некоторый функционал, связанный с работой интерпретатора

Исследуемый код языка представлен ниже:



*Рис. 1.1. Исходный код программы на языке Lisp*

## **Теория**

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а содержательными методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со структурными, т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над внутренними объектами, с которыми имеет дело программа. Для начала перечислим все, что их касается и лежит на поверхности:

* большинство объектов являются именованными. Имя объекта позволяет его идентифицировать, существуют различные области действия имен, соглашения об именах, различные умолчания и т.п. Все это относится к семантике;
* виды, сложность и набор характеристик объектов различаются в разных языках программирования и сильно зависят от области приложения языка (в этом смысле семантика языков программирования более разнообразна, нежели синтаксис и лексика).
* объекты связаны между собой (ссылаются друг на друга).

Семантика программы – внутренняя модель множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей.

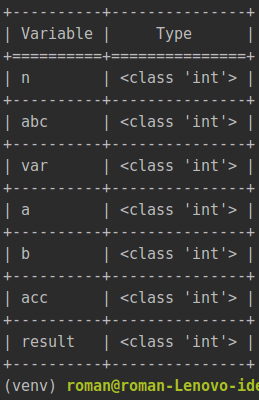
Теперь, когда у нас есть представление о синтаксической фазе, можно оценить ее центральную роль в организации процесса трансляции. Лексические единицы независимы друг от друга и являются терминальными символами синтаксиса. Семантика программы тоже не обладает структурной целостностью и представлена фрагментарно, но при этом связана с синтаксисом следующим образом:

* один и тот же семантический объект (например, переменная) может встречаться в различных, синтаксически несвязанных частях программы;
* синтаксические конструкции описаний, определений и объявлений являются источником семантики объектов программы, они «заявляют» о существовании объектов и задают их свойства;
* синтаксические конструкции, связанные с действиями, выполняемыми над объектами, являются потребителями семантики, их интерпретация, корректность, «смысл» зависят от семантических свойств объекта. Забегая вперед, можно заметить, что заключительная фаза трансляции (генерация кода, интерпретация) может рассматриваться как особые семантические действия, производимые над объектами;
* первичным источником семантики является лексический анализ. Значением лексемы является сама распознанная цепочка литер, она и представляет семантическую составляющую лексемы, которая и обрабатывается;
* лексемы, или то же самое, что терминальные символы входной строки (в терминах синтаксического анализа), ссылаются в семантические таблицах на свою семантику. Формирование семантической составляющей связано с движением снизу-вверх по синтаксическому дереву, от вершин – потомков к предкам.

Таким образом, семантическая составляющая транслятора тоже является фрагментарной (набор семантических процедур, соответствующих правилам грамматики) и объединяется в единое целое только в рамках синтаксического дерева.

## **Результат работы анализатора**

Чтобы продемонстрировать работу семантического анализа был приведён код, где показаны переменные и их тип (Рис. 2.1.)

*Рис. 2.1. Слева указано имя переменной, после стрелочки тип*

Из рисунка видно, что последний анализатор способен распознать тип и как следствие появляется возможность отловить последний тип ошибок – семантических, то есть сложение строк с числами, деление на 0 и многих других. Исходя из этих возможностей, можно сделать вывод о заключающем этапе и приближению к программе-интерпретатору.

**Код с ошибками:**

*Изменим исходный код программы, сознательно сделав несколько семантических ошибок.*

Код программы с 1-ой ошибкой Рис. 3.1. Результат работы программы с допущением 1-ой ошибки Рис. 3.2.

*Рис. 3.1. Семантическая ошибка c попыткой сложения строки и числа*

*Рис. 3.2. Результат работы программы с допущением 1-ой ошибки*

Код программы с 2-ой ошибкой Рис. 4.1. Результат работы программы с допущением 2-ой ошибки Рис. 4.2.

*Рис. 4.1. Семантическая ошибка c попыткой умножения строки и числа*

*Рис. 4.2. Результат работы программы с допущением 2-ой ошибки*

Код программы с 3-ей ошибкой Рис. 5.1. Результат работы программы с допущением 3-ей ошибки Рис. 5.2.

*Рис. 5.1. Семантическая ошибка c попыткой ссылки на функцию со строкой*

*Рис. 5.2. Результат работы программы с допущением 3-ей ошибки*

Код программы с 4-ой ошибкой Рис. 6.1. Результат работы программы с допущением 4-ой ошибки Рис. 6.2.

*Рис. 6.1. Семантическая ошибка cо сложением ссылки одного метода на ссылку второго*

*Рис. 6.2. Результат работы программы с допущением 4-ой ошибки*

## **Выводы**

Я провёл семантический анализ выбранного языка, сформировано его подмножество, придуман способ, который и способен отлавливать ошибки между разными типами и показаны 4 отловленные ошибки в коде программы на языке Lisp, успешно отлавливаемые синтаксическим анализатором.

Сложность заключалась в создании отдельного словаря, куда и записываются все переменные с их значениями, которые язык программирования Python, благодаря своей динамической типизации, способен отловить ошибку в преобразованиях, а программа только выводит её. Язык Lisp также динамический язык программирования как и Python, поэтому не было проблем в данной лабораторной работе.

Семантический анализатор представляет собой третью фазу

компилятора, основная задача которого состоит в создании блока программы, решающей проблему с типами и работой их в коде. В дальнейшем это станет необходимым для создания виртуальной машины или, другими словами, интерпретатора.

Язык разбора Python — [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/Высокоуровневый_язык_программирования) общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, в то же время [стандартная библиотека](https://ru.wikipedia.org/wiki/Стандартная_библиотека_Python) включает большой объём полезных функций. Основные архитектурные черты — [динамическая типизация](https://ru.wikipedia.org/wiki/Динамическая_типизация), [автоматическое управление памятью](https://ru.wikipedia.org/wiki/Сборка_мусора_(программирование)), полная [интроспекция](https://ru.wikipedia.org/wiki/Интроспекция_(программирование)), механизм [обработки исключений](https://ru.wikipedia.org/wiki/Обработка_исключений), поддержка [многопоточных вычислений](https://ru.wikipedia.org/wiki/Многопоточность) и удобные высокоуровневые [структуры данных](https://ru.wikipedia.org/wiki/Структура_данных). Код в Python организовывается в функции и [классы](https://ru.wikipedia.org/wiki/Класс_(программирование)), которые могут объединяться в [модули](https://ru.wikipedia.org/wiki/Модуль_(программирование)) (они в свою очередь могут быть объединены в пакеты).

Популярной реализацией Python является интерпретатор [CPython](https://ru.wikipedia.org/wiki/CPython), поддерживающий большинство активно используемых платформ. Он распространяется под [свободной лицензией](https://ru.wikipedia.org/wiki/Свободное_программное_обеспечение) Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях. Есть реализации интерпретаторов для [JVM](https://ru.wikipedia.org/wiki/JVM), [MSIL](https://ru.wikipedia.org/wiki/MSIL), [LLVM](https://ru.wikipedia.org/wiki/LLVM) и других

## **Приложение А. - Исходный код анализируемой программы**

Данный листинг с кодом демонстрирует исходный анализируемый код с различными типами, такие как унарные и бинарные операции, вывод в консоль, присваивание и т.д.

(print (/= 5 3))

; True

(print (= 5 (+ 2 3)))

; True

(print (= 10 (+ 2 3)))

; False

(print "")

(defun weird-function(n)

(setq n 4)

(print n)

)

(setq n 98)

(setq abc 17)

(weird-function abc) mtran

; 4

(print abc)

; 17

(write "factorial 8 = ")(print (factorial 8 1)) ; factorial 8

{ Fibonacci number }

(defun fibonacci(n)

(fibonacci-iter 0 1 n)

)

(defun fibonacci-iter(a b n)

(if (= n 0) 0)

(if (= n 1) 1)

(setq result (+ a b))

(fibonacci-iter b result (- n 1))

)

(setq n 10)

(fibonacci n)

(write "fibonacci 10 = ")(print result)

(setq n (+ a b))

(write "fibonacci 10 = ")(print n)

(setq var 100) ; var = 100

(setq a (+ 123 456)) ; a = 123 + 456 => 579

(print a)

(setq a (+ 1 2 3) b 123456) ; a = 6, b = 123456

(write "var = ")(print var) ; printing var

(write "a = ")(print a)

(write "b = ")(print b)

(defun factorial(n acc)

(if (= n 0) acc)

(factorial (- n 1) (\* n acc))

))

; single line comment

{ multiline comment }

## **Приложение Б. - Исходный код анализатора**

Данный листинг с кодом демонстрирует методы с различными типами, такие как унарные и бинарные операции, вывод в консоль, присваивание и т.д.

import os

import sys

import texttable as tt

from lib.lexer import Lexer

from lib.parser import Parser

from lib.vm import GLOBAL, execute

def lab\_1*(*args, tokens*)*:

if "lab\_1" in args:

tab = tt.Texttable*()*

headings = *[*'Value (token)', 'Tag', 'Row', 'Column'*]*

tab.header*(*headings*)*

values = list*()*

tags = list*()*

rows = list*()*

columns = list*()*

for token in tokens:

values.append*(*token.value*)*

tags.append*(*token.tag*)*

rows.append*(*token.row*)*

columns.append*(*token.col*)*

for row in zip*(*values, tags, rows, columns*)*:

tab.add\_row*(*row*)*

s = tab.draw*()*

print*(*s*)*

def lab\_2*(*args, ast, tabs*)*:

if "lab\_2" in args:

for i in ast:

if isinstance*(*i, list*)*:

lab\_2*(*args, i, tabs + 1*)*

else:

result = tabs \* ' |'

print*(*'{}{}'.format*(*result, i.value*))*

def lab\_3*(*args*)*:

if "lab\_3" in args:

tab = tt.Texttable*()*

headings = *[*'Variable', 'Type'*]*

tab.header*(*headings*)*

variables = list*()*

types = list*()*

for key, value in GLOBAL.items*()*:

if isinstance*(*value, int*)* or isinstance*(*value, str*)*:

variables.append*(*key*)*

types.append*(*type*(*value*))*

for row in zip*(*variables, types*)*:

tab.add\_row*(*row*)*

s = tab.draw*()*

print*(*s*)*

def main*(*args*)*:

*"""*

*main*

*"""*

path = args*[*0*]*

if len*(*path*)* > 1:

if not os.path.exists*(*path*)*:

print*(*'Error! File "%s" not found!' % path*)*

exit*(*1*)*

else:

print*(*'Error! Expected file, but given nothing!'*)*

exit*(*1*)*

args = args*[*1:*]*

lexer = Lexer*(*path*)*

parser = Parser*()*

tokens = lexer.tokens*()*

if lexer.errors\_list:

lexer.errors*()*

lab\_1*(*args, tokens*)*

try:

ast = parser.build*(*tokens*)*

lab\_2*(*args, ast, 3*)*

""" Lab 4 """

for i in ast:

execute*(*i, GLOBAL*)*

lab\_3*(*args*)*

except Exception as ex:

print*(*ex*)*

exit*(*1*)*

if \_\_name\_\_ == '\_\_main\_\_':

main*(*sys.argv*[*1:*])*

import operator as op

from .lexer import ID, Token

GLOBAL = dict*()*

GLOBAL*[*'+'*]* = lambda env, \*x: obs*(*env, op.add, \*x*)*

GLOBAL*[*'-'*]* = lambda env, \*x: obs*(*env, op.sub, \*x*)*

GLOBAL*[*'\*'*]* = lambda env, \*x: obs*(*env, op.mul, \*x*)*

GLOBAL*[*'/'*]* = lambda env, \*x: obs*(*env, op.truediv, \*x*)*

GLOBAL*[*'//'*]* = lambda env, \*x: obs*(*env, op.floordiv, \*x*)*

GLOBAL*[*'%'*]* = lambda env, \*x: obs*(*env, op.mod, \*x*)*

GLOBAL*[*'='*]* = lambda env, \*x: obs*(*env, op.eq, \*x*)*

GLOBAL*[*'/='*]* = lambda env, \*x: obs*(*env, op.ne, \*x*)*

GLOBAL*[*'>'*]* = lambda env, \*x: obs*(*env, op.gt, \*x*)*

GLOBAL*[*'<'*]* = lambda env, \*x: obs*(*env, op.lt, \*x*)*

GLOBAL*[*'>='*]* = lambda env, \*x: obs*(*env, op.ge, \*x*)*

GLOBAL*[*'<='*]* = lambda env, \*x: obs*(*env, op.le, \*x*)*

GLOBAL*[*'~'*]* = lambda env, \*x: obs*(*env, op.ne, \*x*)*

GLOBAL*[*'setq'*]* = lambda env, \*x: setq*(*env, \*x*)*

GLOBAL*[*'defun'*]* = lambda env, \*x: defun*(*env, \*x*)*

GLOBAL*[*'if'*]* = lambda env, \*x: compare*(*env, \*x*)*

GLOBAL*[*'write'*]* = lambda env, \*x: write*(*env, \*x*)*

GLOBAL*[*'print'*]* = lambda env, \*x: write\_line*(*env, \*x*)*

GLOBAL*[*'readint'*]* = lambda env, \*x: readint*(*env, \*x*)*

class Procedure*(*object*)*:

*"""*

*procedure*

*"""*

def \_\_init\_\_*(*self, params, \*body*)*:

*"""*

*params, expr, env*

*"""*

self.params, self.body = params, body

def \_\_call\_\_*(*self, env, \*args*)*:

if len*(*args*)* != len*(*self.params*)*:

msg = "Too many args! Expected %s, given %s" % *(*len*(*self.params*)*, len*(*args*))*

msg += ' in line {}, column {}'.format*(*args*[*0*]*.col, args*[*0*]*.row*)*

raise TypeError*(*msg*)*

for i, par in enumerate*(*self.params*)*:

env*[*par.value*]* = execute*(*args*[*i*]*, env*)*

magic = False

while True:

if magic:

for i, par in enumerate*(*self.params*)*:

env*[*par.value*]* = args*[*i*]*

# вычисляем тело функции

length = len*(*self.body*)* - 1

for i, expr in enumerate*(*self.body*)*:

if i < length: # если это не последнее выражение

result = execute*(*expr, env*)*

magic = True

if magic and result:

return result

else:

if isinstance*(*env*[*expr*[*0*]*.value*]*, Procedure*)*:

proc = env*[*expr*[*0*]*.value*]*

self.params = proc.params

self.body = proc.body

args = *[*execute*(*i, env*)* for i in expr*[*1:*]]*

magic = True

else:

result = execute*(*expr, env*)*

return result

def obs*(*env, fun, \*args*)*:

*"""*

*obs*

*"""*

result = execute*(*args*[*0*]*, env*)*

for i in args*[*1:*]*:

result = fun*(*result, execute*(*i, env*))*

return result

def defun*(*env, \*args*)*:

*"""*

*defune new function*

*"""*

name, params, \*body = args

proc = Procedure*(*params, \*body*)*

if not name.value in env:

env*[*name.value*]* = proc

else:

msg = 'Function "%s" already exists!' % name.value

msg += 'in line {}, column {}'.format*(*name.col, name.row*)*

raise Exception*(*msg*)*

def compare*(*env, \*args*)*:

*"""*

*get condition and execute first or second body*

*"""*

if execute*(*args*[*0*]*, env*)*:

return execute*(*args*[*1*]*, env*)*

elif len*(*args*)* == 3:

return execute*(*args*[*2*]*, env*)*

def write*(*env, \*args*)*:

*"""*

*write line*

*"""*

from sys import stdout

stdout.write*(*str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def write\_line*(*env, \*args*)*:

*"""*

*write new line*

*"""*

from sys import stdout

stdout.write*(*'%s\n' % str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def readint*(*env, \*args*)*:

*"""*

*read line*

*"""*

i = 0

env*[*args*[*i*]*.value*]* = int*(*input*())*

from sys import stdout

if isinstance*(*args*[*i*]*.value, str*)*:

stdout.write*(*str*(*execute*(*args*[*0*]*, env*)))*

stdout.flush*()*

def setq*(*env, \*args*)*:

*"""*

*define new variables*

*"""*

i = 0

while i < len*(*args*)*:

env*[*args*[*i*]*.value*]* = execute*(*args*[*i + 1*]*, env*)*

i += 2

def execute*(*expr, env*)*:

*"""*

*execute*

*"""*

if isinstance*(*expr, Token*)*:

if expr.tag == ID and expr.value in env:

return env*[*expr.value*]*

else:

return expr.value

else:

first, \*second = expr

if first.value in env and callable*(*env*[*first.value*])*:

return env*[*first.value*](*env, \*second*)*

else:

msg = 'Function "%s" not exists!' % first.value

msg += 'in line {}, column {}'.format*(*first.col, first.row*)*

raise Exception*(*msg*)*

RESERVED = 'RESERVED'

UNKNOWN = 'UNKNOWN'

NUMBER = 'NUMBER'

STRING = 'STRING'

QUOTE = 'QUOTE'

ID = 'ID'

class Token:

*"""*

*docstring for Token*

*"""*

def \_\_init\_\_*(*self, value, tag, row, col*)*:

self.value = value

self.tag = tag

self.row = row

self.col = col

def \_\_str\_\_*(*self*)*:

return '<{}, {}, {}, {}>'.format*(*self.value, self.tag, self.row, self.col*)*

def \_\_repr\_\_*(*self*)*:

return self.\_\_str\_\_*()*

class Lexer*(*dict*)*:

*"""*

*docstring for Lexer*

*"""*

def \_\_init\_\_*(*self, file, \*args*)*:

super*()*.\_\_init\_\_*(*\*args*)*

self.pos, self.row, self.col = 0, 1, 1

self.char = ''

self.file = open*(*file, 'r'*)*

self.string = self.file.readline*()*

self.errors\_list = list*()*

def errors*(*self*)*:

*"""*

*print all errors*

*"""*

import sys

self.file.close*()*

sys.stderr.write*(*'Lexer errors:\n'*)*

for i in self.errors\_list:

sys.stderr.write*(*'\t%s\n' % i*)*

sys.stderr.flush*()*

exit*(*1*)*

def error*(*self, text*)*:

*"""*

*print error*

*"""*

self.errors\_list.append*(*

'{} in line {}, column {}'.format*(*text, self.row, self.col*))*

def next\_char*(*self*)*:

*"""*

*set next char*

*"""*

if self.pos < len*(*self.string*)*:

self.char = self.string*[*self.pos*]*

if self.char != '\n':

self.col += 1

self.pos += 1

else:

self.string = self.file.readline*()*

self.col = 1

self.row += 1

self.pos = 0

else:

self.char = '#0'

def skip\_space*(*self*)*:

*"""*

*skip spaces*

*"""*

while self.char.isspace*()*:

self.next\_char*()*

def next\_token*(*self*)*:

*"""*

*return token*

*"""*

self.skip\_space*()*

lexem = ''

# if current char is alpha or \_

if self.char.isalpha*()* or self.char == '\_' or self.char in '+-\*/%><=^!?':

lexem = self.char

self.next\_char*()*

# adding all alpha and digit

while self.char.isalpha*()* or self.char.isdigit*()* or self.char in '+-\*/%><=^!?':

lexem += self.char

self.next\_char*()*

return Token*(*lexem, ID, self.col, self.row*)*

# if current char is digit

elif self.char.isdigit*()*:

# while is digit

count = 0

while self.char.isdigit*()* or self.char == '.':

if self.char == '.':

count += 1

if count > 1:

self.error*(*'Incorrect format of number: "%s"' % self.char*)*

lexem += self.char

self.next\_char*()*

return Token*(*int*(*lexem*)* if count == 0 else float*(*lexem*)*, NUMBER, self.col, self.row*)*

elif self.char in *(*'(', ')'*)*:

lexem = self.char

self.next\_char*()*

return Token*(*lexem, RESERVED, self.col, self.row*)*

elif self.char == '#0':

return Token*(*'EOF', None, self.col, self.row*)*

elif self.char == '-':

lexem = self.char

self.next\_char*()*

if self.char.isdigit*()*:

count = 0

while self.char.isdigit*()* or self.char == '.':

if self.char == '.':

count += 1

if count > 1:

self.error*(*'Incorrect format of number: "%s"' %

self.char*)*

lexem += self.char

self.next\_char*()*

return Token*(*int*(*lexem*)* if count == 0 else float*(*lexem*)*, NUMBER, self.col, self.row*)*

elif self.char in *(*';', '{'*)*:

# skip comments in file

return self.skip\_comments*(*'\n' if self.char == ';' else '}'*)*

elif self.char == '"':

self.next\_char*()*

while self.char != '"':

if self.char == '\\':

lexem += self.char

self.next\_char*()*

lexem += self.char

self.next\_char*()*

continue

lexem += self.char

self.next\_char*()*

self.next\_char*()*

return Token*(*lexem, STRING, self.col, self.row*)*

elif self.char == "'":

lexem = self.get\_quote*()*

return Token*(*lexem*[*1:*]*, QUOTE, self.col, self.row*)*

elif self.char in self:

lexem = self.char

self.next\_char*()*

return Token*(*lexem, self*[*lexem*]*, self.col, self.row*)*

else:

lexem = self.char

self.error*(*'Unknown character: "%s"' % self.char*)*

self.next\_char*()*

return Token*(*lexem, UNKNOWN, self.col, self.row*)*

return None

def get\_quote*(*self, skip\_spaces=True*)*:

*"""*

*return quote*

*"""*

lexem = self.char

if skip\_spaces:

self.skip\_space*()*

self.next\_char*()*

while True:

if self.char == '(':

lexem += self.get\_quote*(*False*)*

if self.char == ')':

lexem += self.char

self.next\_char*()*

if self.char != ')':

lexem += self.char

self.next\_char*()*

else:

return lexem

def skip\_comments*(*self, char*)*:

*"""*

*skip comments*

*"""*

while self.char != char:

self.next\_char*()*

self.next\_char*()*

return self.next\_token*()*

def gettoken*(*self*)*:

*"""*

*return token*

*"""*

self.next\_char*()*

while True:

result = self.next\_token*()*

if not result:

continue

if result.value == 'EOF':

break

yield result

def tokens*(*self*)*:

*"""*

*retun list of tokens*

*"""*

result = *[*i for i in self.gettoken*()]*

return result

def raw\_input*(*self, user\_string*)*:

*"""*

*return raw user input*

*"""*

self.string = user\_string

return self.tokens*()*

class Parser*(*object*)*:

*"""*

*class Parser*

*"""*

def \_\_init\_\_*(*self*)*:

self.tokens = None

def \_node*(*self, pos*)*:

*"""*

*return new node and pos*

*"""*

node = list*()*

while self.tokens*[*pos*]*.value != ')':

if self.tokens*[*pos*]*.value == '(':

new\_node, pos = self.\_node*(*pos + 1*)*

node.append*(*new\_node*)*

else:

node.append*(*self.tokens*[*pos*])*

pos += 1

return node, pos

def build*(*self, tokens*)*:

*"""*

*return ast*

*"""*

ast = list*()*

if tokens:

pos = 0

self.tokens = tokens

while pos < len*(*tokens*)*:

if tokens*[*pos*]*.value == '(':

node, pos = self.\_node*(*pos + 1*)*

pos += 1

ast.append*(*node*)*

else:

msg = 'Parser error! Expected "(" but given "%s"' % tokens*[*pos*]*.value

msg += ' in line {}, column {}'.format*(*tokens*[*pos*]*.col - 1, tokens*[*pos*]*.row*)*

raise Exception*(*msg*)*

return ast