Содержание

[1. Постановка задачи 3](#_Toc4170293)

[2. Теория 4](#_Toc4170294)

[3. Результат работы программы 9](#_Toc4170295)

[Выводы 12](#_Toc4170296)

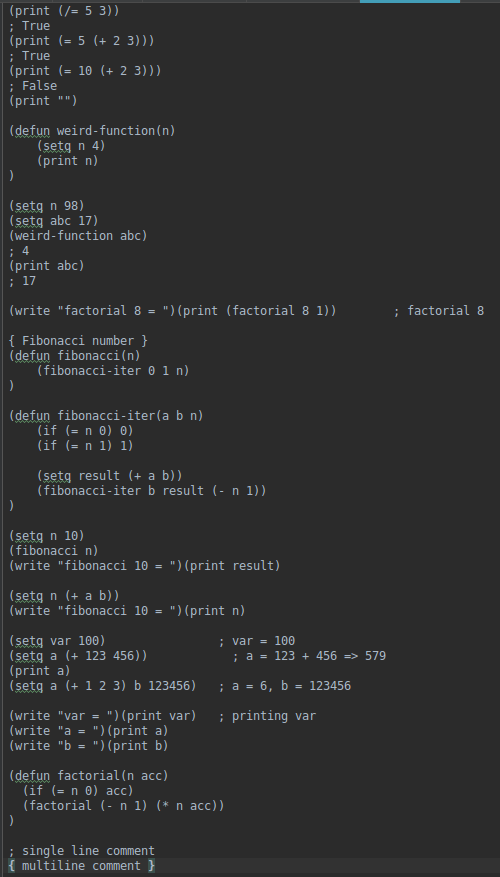
[Приложение A –](#_Toc4170297) Код анализируемой программы 13

[Приложение Б – Код анализатора 1](#_Toc4170297)5

## **Постановка задачи**

В данной работе ставится задача - освоить работу с существующими лексическими анализаторами. Разработать собственный лексический анализатор подмножества языка программирования, для чего определить лексические правила и выполнить перевод потока символов в поток лексем (токенов). Основной целью работы является реализация лексического анализатора. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

Исследуемый код языка представлен ниже на Рис. 1.1.:

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

## **Теория**

Лексический анализатор— это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии:

1. сканирование
2. оценка

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах . Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход синтаксического анализатора.

В информатике лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор, оператор», «часть речи» и т.п.). Лексический анализ используется в компиляторах и интерпритаторах исходного кода языков программирования, и в различных парсерах естественных языков.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы, например для граматического анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

#### ***Фазы***

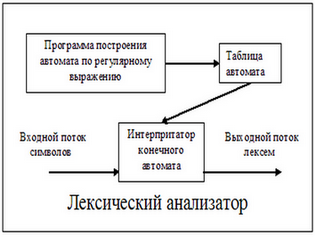
Фаза- это самостоятельная задача в процессе компиляции. Как правило, несколько фаз объединяются в один проход. Фазы лексическогоанализаторакомпилятора (также называемый сканером, лексером) переводит входящие данные в форму, более полезную для остальной части компилятора.

***Токены и лексемы***

Лексический анализатор просматривает входной поток как совокупность основных элементов языка, называемые токенами. Это означает, что токен является лексической неделимой единицей. В Lisp ключевые слова **while** и **for** являются токенами (мы не можем сказать wh ile), такие символы как >, >, >= токены, идентификаторы и числа тоже токены и прочее. Исходную строку, содержащую токен, называют лексемой. Обратите внимание, что токен и лексемы похожи, но не одно и то же. Например, токены идентификатор или число могут иметь множество связанных с ними лексем (10, 2.3, 506 - лексемы, число - токен), в тоже время токен совпадает с лексемой состоящей из одного символа. Ситуацию осложняют токены, которые пересекаются с другими (например, какой токен нужно распознать в потоке ">>=": ">", ">>" или ">=" ?). В общем лексический анализатор распознает токен, который соответствует самой длинной лексеме - во многих языках программирование такое поведение указано в спецификации. Таким образом в нашем случае мы получим токен сдвига ">>", а не два токена больше чем.

#### ***Выбор множества токенов***

Одним из первых проектных решений, которые могут повлиять на структуру всего компилятора, является выбор множества токенов. Вы можете иметь токены для каждаго входного символа или несколько символов могут быть объединены в один токен. Например, символы >, >=, >>, и >>= могут рассматриваться либо как четыре токена, либо как один токен оператор-сравнения. При этом, лексема используется для устранения неоднозначности токена. Первый подход может упростить генерацию кода. Однако, слишком много токенов могут сделать парсер слишком большим и трудным в написании. Нет жестких и быстрых правил для выбора как лучше, но после прочтения книги вы поймете проектные соображения и сможете сделать осознанный выбор. В общем, арифметические операции с одним приоритетом и ассоциативностью могут быть сгруппированы вместе, также ключевые слова для определения типа (вроде int или char) могут быть объединены и т.д.



*Рис. 2.1. Принцип работы лексера*

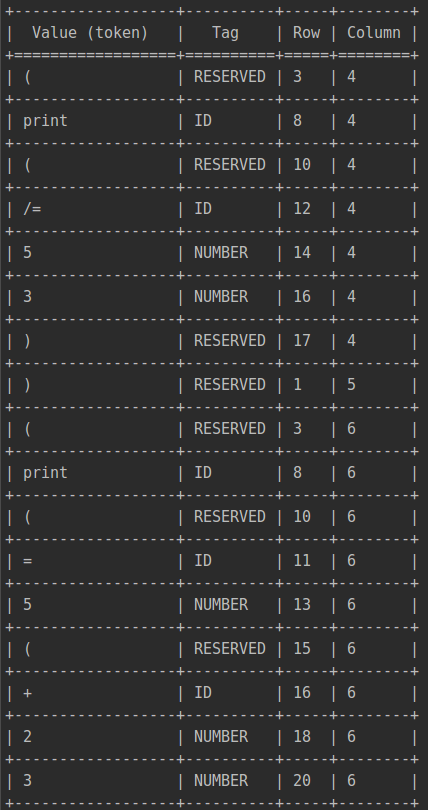
***Анализ текста и разбор кода***

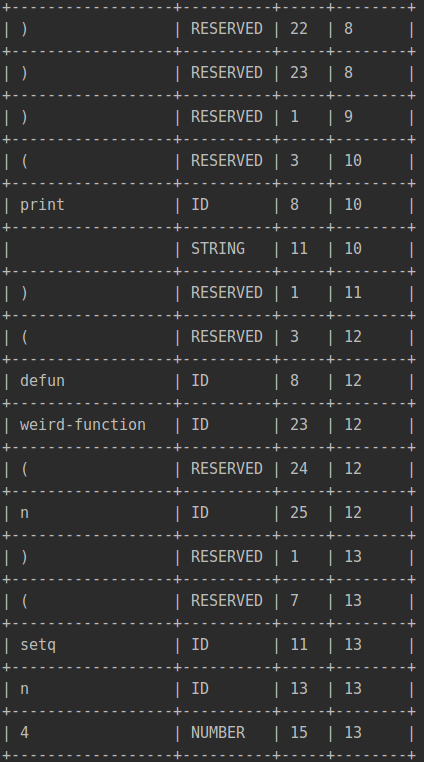
Когда программист пытается написать парсер текста, его естественный подход — рекурсивное углубление: найти начало конструкции (например, **{**); найти её конец (например, **}**на том же уровне вложенности); выделить содержимое конструкции, и пропарсить её рекурсивно.Проблемы с таким подходом — во-первых, избыточная сложность (по одному и тому же фрагменту текста гуляем взад-вперёд); во-вторых, неудобство поддержки (синтаксис языка оказывается рассредоточен по килобайтам и килобайтам ветвистого кода).Синтаксис языка можно задать декларативно; например, всем знакомы регулярные выражения. Идеально было бы написать стопочку регэкспов для всех конструкций языка, напротив каждого — определение узла, который должен создаться в дереве программы; «универсальный парсер» бы просто подставлял программу в один регэксп за другим, и создавал узлы согласно описанию, один за другим.Первая из названных проблем решается тем, что поиск всех регэкспов в тексте можно выполнить за один проход, т.е. нет надобности хранить всю программу в памяти целиком — достаточно читать её по одному символу, обрабатывать символ, и тут же забывать.Вторая — тем, что теперь у нас есть централизованное, формальное описание языка: можем менять регэкспы, вовсе не трогая код; и наоборот, менять код, не рискуя повредить парсер.

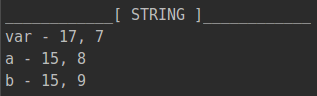
Парсер будет читать входную строку символ за символом, и записывать (сдвигать) прочитанные символы в стек. Как только наверху стека соберётся последовательность (символов и переменных), подходящая к правилу грамматики, автомат вытолкнет всю её из стека, и заменит на переменную, стоящую в левой части подошедшего правила (свёртка). Вся работа автомата заключается в последовательности сдвигов и свёрток.Интересный момент: автомату на самом деле не важно, какие символы лежат в стеке. Всё равно он не может их сравнить с правилами грамматики, потому что видит только верхний; вместо этого он выбирает, какое правило применить для свёртки, по своему текущему состоянию. Стек ему нужен затем, чтобы знать, в какое состояние перейти после свёртки. Для этого он во время сдвига записывает в стек, вместе с символом, своё текущее состояние; а во время свёртки берёт из стека состояние, записанное под всеми стёртыми символами, и в зависимости от него переходит в следующее состояние.

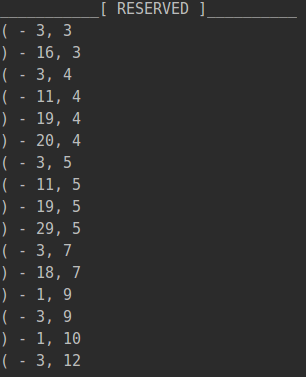
## **Результат работы программы**

Для анализа программы разложим ее на лексемы и токены. Результат работы программы доступен ниже (Рис. 4.1, Рис. 4.2, Рис. 4.3, Рис. 4.4, Рис. 4.5, Рис. 4.6).

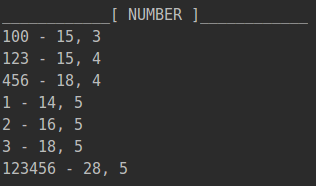
*Рис. 4.1. Результат работы анализатора — полная таблица (часть 1)*

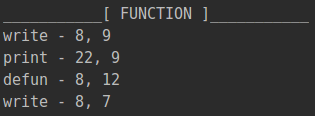
*Рис. 4.2. Результат работы анализатора — полная таблица (часть 2)*

**

*Рис. 4.3. Результат работы анализатора — зарезервированные токены*

*Рис. 4.4. Результат работы анализатора — чиловые токены*

*Рис. 4.5. Резуль**тат работы анализатора — строковые токены*

*Рис. 4.6. Результат работы анализатора — функциональные токены*

**Код с ошибками**

Рассмотрим тот же код программы с добавленными в него ошибок. При

обнаружении их происходит вывод уведомления об ошибке.

Код программы с 1-ой ошибкой Рис. 5.1. Результат работы программы с допущением 1-ой ошибки Рис. 5.2.

*Рис. 5.1. Ошибка некорректного оператора присваивания*

*Рис. 5.2. Результат работы лексера*

Код программы с 2-ой ошибкой Рис. 6.1. Результат работы программы с допущением 2-ой ошибки Рис. 6.2.



*Рис. 6.1. Ошибка порядка ключевых слов*

*Рис. 6.2.. Результат работы лексера*

Код программы с 3-ей ошибкой Рис. 7.1. Результат работы программы с допущением 3-ей ошибки Рис. 7.2.

*Рис. 7.1. Ошибка в названии токена*

*Рис. 7.2. Результат работы лексера*

## **Выводы**

Я провёл анализ выбранного языка, сформировано его подмножество, составлена таблица токенов с информацией о их расположении в исходном коде и показаны 4 отловленные ошибки в коде программы на языке Lisp. Результатом работы программы на данном этапе является набор всех токенов, с указанием их места вхождения. Вниманию уделён момент, связанный с возможным лексическим ошибкам. В данной работе рассмотрены 4 возможных ошибки, успешно отлавливаемые лексическим анализатором.

Сложность лексического анализа в некоторой степени состоит в том, что для корректного определения текущей лексемы (или для определения ошибки) нам нужно понять какие символы идут за ней.

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора.

Приложение А - Код анализируемой программы

(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 - Код анализатора

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