Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЕТ

к лабораторной работе №2

на тему

**ЛЕКСИЧЕСКИЙ АНАЛИЗ**

Студент А. В. Скворцов

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc159418165)

[2 Краткие теоретические сведения 4](#_Toc159418166)

[3 Результаты выполнения лабораторной работы 5](#_Toc159418167)

[Заключение 7](#_Toc159418168)

[Список использованных источников 8](#_Toc159418169)

[Приложение А (обязательное) Листинг кода 9](#_Toc159418170)

1. **ЦЕЛЬ РАБОТЫ**

Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Также необходимо обнаружить ошибку при определении неверной последовательности символов и сообщить о ней. Всего необходимо показать нахождение четырех лексических ошибок.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Разработанный программный продукт представляет собой лексический анализатор для языка программирования Python. Он разбивает входной исходный код на лексемы (токены) и определяет типы этих токенов.

Лексема (лексическая единица языка) — это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка. Лексемами языков естественного общения являются слова1. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т. п. Состав возможных лексем каждого конкретного языка программирования определяется синтаксисом этого языка [1].

Лексический анализатор (или сканер) – это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. То, какие функции должен выполнять лексический анализатор и какие типы лексем он должен выделять во входной программе, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев, незначащих пробелов, символов табуляции и перевода строки, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей [2].

Работа программы строится на принципе разбиения входного текста программы на токены и дальнейшего их анализа. В частности, обрабатываются такие ошибки, как неверный порядок операторов, неверное количество отступов, так как в питоне это довольно важный аспект и закрытие блоков, который «ловят» исключения.

1. **РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**

В рамках лабораторной работы был разработан лексический анализатор языка Python, для тестирования был создан файл, который содержит некоторое количество лексических ошибок и содержащий основные конструкции языка (рисунок 1).

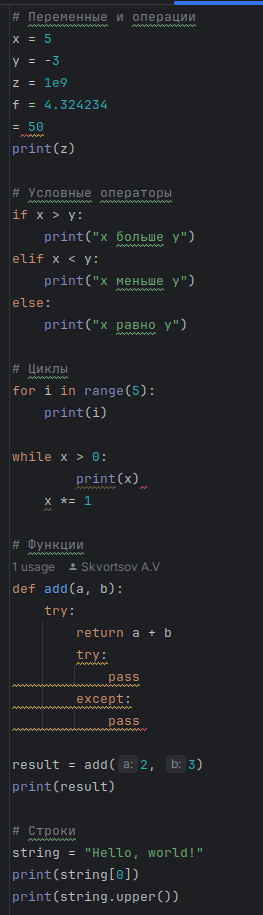


Рисунок 1 – Тестовый код

После того, как на этот код был запущен лексический анализатор, можно увидеть результаты выполнения программы.

На рисунке ниже приведен пример вывода программы для операторов и констант (рисунок 2).

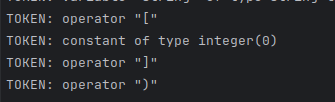


Рисунок 2 – Операторы и константы

На следующем рисунке приведены примеры определения переменных и их значений после инициализации (рисунок 3).

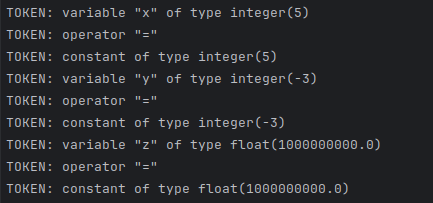


Рисунок 3 – Переменные

На следующем рисунке приведен пример вывода лексических ошибок в коде (рисунок 4).

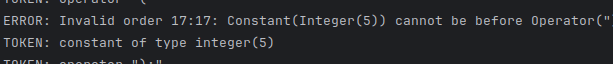


Рисунок 4 – Ошибки

## 

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы был разработан лексический анализатор подмножества языка программирования Python. Также были обнаружены ошибки при определении неверной последовательности символов. Был продемонстрирован результат работы программы при обнаружении некорректного количества отступов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Лексические анализаторы [Электронный ресурс]. – Режим доступа: https://studfile.net/preview/7382388/page:3/. – Дата доступа: 21.02.2024.
2. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/LabWork2.pdf – Дата доступа: 15.02.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – lexer.rs

use std::collections::HashMap;

use std::fmt::{Debug, Display, Formatter, write};

use std::fs::File;

use std::io::Read;

use std::ops::AddAssign;

use std::str::Chars;

#[derive(Debug, PartialEq, Clone)]

enum Token {

Keyword(String),

Identifier(String),

Variable((Type, String)),

Operator(String),

Constant(Type),

Indent,

Dedent,

Newline,

LexicalError(String),

}

impl Display for Token {

fn fmt(&self, f: &mut Formatter<'\_>) -> std::fmt::Result {

match self {

Token::Keyword(name) => write!(f, "TOKEN: default Python keyword \"{name}\""),

Token::Identifier(name) => write!(f, "TOKEN: identifier \"{name}\""),

Token::Variable((t, name)) => write!(f, "TOKEN: variable \"{name}\" of type {t}"),

Token::Operator(name) => write!(f, "TOKEN: operator \"{name}\""),

Token::Constant(t) => write!(f, "TOKEN: constant of type {t}"),

Token::LexicalError(s) => write!(f, "ERROR: {}", s),

\_ => { Ok(()) }

}

}

}

#[derive(Debug, PartialEq, Clone)]

enum Type {

Integer(i32),

Float(f64),

String(String),

Char(char),

}

impl Display for Type {

fn fmt(&self, f: &mut Formatter<'\_>) -> std::fmt::Result {

match self {

Type::Integer(v) => write!(f, "integer({v})"),

Type::Float(v) => write!(f, "float({:?})", v),

Type::String(v) => write!(f, "string literal(\"{v}\")"),

Type::Char(v) => write!(f, "char({v})")

}

}

}

struct Lexer<'a> {

input: Chars<'a>,

current\_char: Option<char>,

current\_index: i32,

}

impl<'a> Lexer<'a> {

fn new(input: &'a str) -> Lexer<'a> {

let mut lexer = Lexer {

input: input.chars(),

current\_char: None,

current\_index: 0,

};

lexer.advance();

lexer

}

fn advance(&mut self) {

self.current\_char = self.input.next();

self.current\_index += 1;

}

fn tokenize(&mut self) -> Result<Vec<Token>, String> {

let mut tokens = Vec::new();

let mut indentation\_stack = Vec::new();

let mut current\_indentation = 0;

loop {

let current\_index = self.current\_index;

match self.current\_char {

Some(' ') => {

self.advance();

}

Some('\n') => {

self.current\_index = 0;

self.advance();

tokens.push((Token::Newline, self.current\_index));

let mut spaces\_count = 0;

while let Some(' ') = self.current\_char {

self.advance();

spaces\_count += 1;

}

if spaces\_count > current\_indentation {

for \_ in (current\_indentation..spaces\_count).step\_by(4) {

tokens.push((Token::Indent, self.current\_index));

}

indentation\_stack.push(current\_indentation);

current\_indentation = spaces\_count;

} else {

while spaces\_count < current\_indentation {

tokens.push((Token::Dedent, self.current\_index));

current\_indentation = indentation\_stack.pop().unwrap();

}

}

}

Some(ch) => {

if ch.is\_ascii\_digit() || ch == '-' {

tokens.push((self.consume\_number(), current\_index));

} else if ch.is\_alphabetic() {

tokens.push((self.consume\_identifier(), current\_index));

} else {

match ch {

'#' => {

self.advance();

self.consume\_comment();

}

'"' => {

self.advance();

tokens.push((self.consume\_string\_literal(), current\_index));

}

\_ => {

tokens.push((self.consume\_operator(), current\_index));

}

}

}

}

None => break,

}

}

while current\_indentation > 0 {

tokens.push((Token::Dedent, self.current\_index));

current\_indentation = indentation\_stack.pop().unwrap();

}

let mut verified\_tokens = self.verify\_output(&mut tokens);

self.add\_data\_types(verified\_tokens.clone().len(), &mut verified\_tokens);

Ok(verified\_tokens)

}

fn consume\_number(&mut self) -> Token {

let mut num\_str = String::new();

let mut is\_float = false;

let mut is\_negative = false;

if let Some(ch) = self.current\_char {

if ch == '-' {

is\_negative = true;

self.advance();

}

}

while let Some(ch) = self.current\_char {

if ch.is\_ascii\_digit() {

num\_str.push(ch);

self.advance();

} else if ch == '.' || ch == 'e' && !is\_float {

num\_str.push(ch);

self.advance();

is\_float = true;

} else {

break;

}

}

if is\_float {

if let Ok(f) = num\_str.parse::<f64>() {

if is\_negative {

Token::Constant(Type::Float(-f))

} else {

Token::Constant(Type::Float(f))

}

} else {

panic!("Invalid float number: {}", num\_str);

}

} else {

if let Ok(i) = num\_str.parse::<i32>() {

if is\_negative {

Token::Constant(Type::Integer(-i))

} else {

Token::Constant(Type::Integer(i))

}

} else {

panic!("Invalid integer: {}", num\_str);

}

}

}

fn consume\_identifier(&mut self) -> Token {

let mut identifier = String::new();

while let Some(ch) = self.current\_char {

if ch.is\_alphanumeric() || ch == '\_' {

identifier.push(ch);

self.advance();

} else {

break;

}

}

match identifier.as\_str() {

"if" | "else" | "elif" | "for" | "while" | "def" | "class" | "and" | "or" | "is" | "not" | "try" | "except" | "finally" | "pass" => Token::Keyword(identifier),

\_ => Token::Identifier(identifier),

}

}

fn consume\_comment(&mut self) {

while let Some(ch) = self.current\_char {

if ch == '\n' {

break;

}

self.advance();

}

}

fn consume\_string\_literal(&mut self) -> Token {

let mut literal = String::new();

let mut is\_escaped = false;

while let Some(ch) = self.current\_char {

if ch == '\n' {

panic!("Unterminated string literal");

}

if is\_escaped {

match ch {

'n' => literal.push('\n'),

't' => literal.push('\t'),

'\\' => literal.push('\\'),

'"' => literal.push('"'),

\_ => literal.push(ch),

}

is\_escaped = false;

} else if ch == '\\' {

is\_escaped = true;

} else if ch == '"' {

self.advance();

break;

} else {

literal.push(ch);

}

self.advance();

}

Token::Constant(Type::String(literal))

}

fn consume\_operator(&mut self) -> Token {

let mut operator = String::new();

while let Some(ch) = self.current\_char {

if ch.is\_ascii\_whitespace() || ch.is\_alphanumeric() || ch == '"' {

break;

}

if !operator.is\_empty() && (ch == ')' || ch == ']' || ch == '}') {

break;

}

operator.push(ch);

self.advance();

}

if operator == "\"" {

self.advance();

return self.consume\_string\_literal();

}

Token::Operator(operator)

}

fn verify\_output(&mut self, tokens: &mut Vec<(Token, i32)>) -> Vec<Token> {

let mut line\_number = 1;

let mut new\_tokens: Vec<Token> = Vec::new();

let tokens\_clone = tokens.clone();

let try\_except\_diff = tokens\_clone.iter().filter(|&v| v.0 == Token::Keyword("try".to\_string())).count() - tokens.iter().filter(|&v| v.0 == Token::Keyword("except".to\_string())).count();

if try\_except\_diff > 0 {

let error\_tryies = tokens\_clone.iter().filter(|&v| v.0 == Token::Keyword("try".to\_string())).take(try\_except\_diff);

for error in error\_tryies {

let position = tokens.clone().iter().position(|el| el == &error.clone());

tokens.insert(position.unwrap(), (Token::LexicalError(format!("All \"try\" need \"except\" {:?}:{:?}", tokens[0..position.unwrap()].iter().filter(|&v| v.0 == Token::Newline).count() + 1, error.1)), error.1));

}

}

for (index, (token, \_)) in tokens.iter().enumerate() {

if index == tokens.len() - 1 {

break;

}

if index == 0 {

continue;

}

match tokens[index - 1].clone() {

(Token::Operator(a), \_) => {

if a == ":" {

let (indent, column\_number) = tokens[index + 1].clone();

let (next\_intend, \_) = tokens[index + 2].clone();

if \*token != Token::Newline || indent != Token::Indent || next\_intend == Token::Indent {

new\_tokens.push(Token::LexicalError(format!("Invalid intent {:?}:{:?}", line\_number, column\_number)))

}

}

}

\_ => {}

}

match tokens[index + 1].clone() {

(Token::Operator(a), column\_num) => {

if a != ":" && a != ")" && a != "]" {

match token {

Token::Identifier(\_) => {}

\_ => {

new\_tokens.push(Token::LexicalError(format!("Invalid order {:?}:{:?}: {:?} cannot be before {:?}", line\_number, column\_num, \*token, tokens[index + 1].0)));

}

}

}

}

(Token::Newline, \_) => {

line\_number += 1;

}

(Token::Indent, \_) => {}

\_ => {}

}

if \*token != Token::Indent && \*token != Token::Newline && \*token != Token::Dedent {

new\_tokens.push(token.clone())

}

}

return new\_tokens;

}

fn add\_data\_types(&self, size: usize, tokens: &mut [Token]) {

let mut variables: HashMap<String, Token> = HashMap::new();

for i in 0..size {

match tokens[i].clone() {

Token::Identifier(name) => {

if i < size - 3 && tokens[i + 1] == Token::Operator("=".to\_string()) {

match tokens[i + 2].clone() {

Token::Constant(t) => {

tokens[i] = Token::Variable((t, name.clone()));

variables.insert(name, tokens[i].clone());

}

\_ => {}

}

} else {

if variables.contains\_key(&name) {

tokens[i] = variables.get(&name).unwrap().clone()

}

}

}

Token::Variable((t, str)) => {

if i < size - 3 && tokens[i + 1] == Token::Operator("=".to\_string()) {}

}

\_ => {}

}

}

}

}

fn main() {

let mut file = File::open("./test.py").expect("Unable to open the file");

let mut contents = String::new();

file.read\_to\_string(&mut contents).expect("Unable to read the file");

let mut lexer = Lexer::new(contents.as\_str());

let tokens = lexer.tokenize();

match tokens {

Ok(tokens) => {

for token in tokens {

println!("{token}");

}

}

Err(err) => {

println!("{}", err);

}

}

}