|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**Лабораторная работа № 4**

**«Объектно-ориентированный лексический анализатор»**

***по курсу «Конструирование компиляторов»***

Студент *Ионов Т.Р. 61Б*

Преподаватель *Коновалов А.B.*

*Москва, 2022 г.*

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc106224508)

[2 Индивидуальное задание 3](#_Toc106224509)

[3 Реализация 3](#_Toc106224510)

[4 Тестирование 4](#_Toc106224511)

[Вывод 4](#_Toc106224512)

# 1 Цель работы

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

# 2 Индивидуальное задание

Строковые литералы: ограничены двойными кавычками, не могут пересекать границы текста, содержат escape-последовательности “\n”, “\””, “\t”, “\\”.

Числовые литералы: последовательности десятичных знаков и знаков “\_”, начинающихся с цифры .

3 Реализация

use std::collections::HashMap;  
use std::fmt;  
use std::fmt::format;  
use std::fs::File;  
use std::io::{BufRead, BufReader};  
use std::process::id;  
  
const *EOF*: char = '\0';  
  
fn is\_letter(ch: char) -> bool {  
 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z'  
}  
  
fn is\_digit(ch: char) -> bool {  
 '0' <= ch && ch <= '9'  
}  
  
*// fn is\_ident\_char(ch: char) -> bool {  
// ch == '$'  
// }*fn is\_break(ch: char) -> bool {  
 ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n' || ch == *EOF*}  
  
#[derive(Copy, Clone)]  
pub struct Position {  
 row: i32,  
 col: i32,  
 i: usize  
}  
  
impl Position {  
 pub fn to\_string(&self) -> String {  
 return format!("({}, {})", self.row, self.col)  
 }  
}  
  
  
  
  
#[derive(Clone)]  
pub struct Token {  
 start: Position,  
 end: Position,  
 token\_type: TokenType,  
}  
  
impl Token {  
 pub fn to\_string(&self) -> String {  
 return format!("{} {}-{}", self.token\_type.to\_string(), self.start.to\_string(),  
 self.end.to\_string())  
 }  
}  
  
#[derive(Clone)]  
pub enum TokenType {  
 *EOF*,  
 *ERROR*(String),  
 *STR*(String),  
 *IDENT*(i32),  
 *NUMBER*(i32),  
}  
  
impl fmt::Display for TokenType {  
 fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {  
 match self {  
 TokenType::*EOF* => write!(f, "EOF"),  
 TokenType::*ERROR*(error) => write!(f, "error: {}", error.to\_string()),  
 TokenType::*STR*(str) => write!(f, "STR: {}", str),  
 TokenType::*IDENT*(ident) => write!(f, "IDENT: {}", ident.to\_string()),  
 TokenType::*NUMBER*(number) => write!(f, "NUMBER: {}", number.to\_string()),  
  
 \_ => write!(f, "unknown")  
 }  
 }  
}  
  
pub struct Lexer {  
 input: Vec<char>,  
 pub pos: Position,  
 pub char: char,  
 pub token: TokenType,  
 pub idents: HashMap<String, i32>,  
 pub ident\_i: i32  
  
}  
  
impl Lexer {  
  
 pub fn read\_char(&mut self){  
 self.char = self.input[self.pos.i];  
 }  
  
 pub fn inc\_pos(&mut self) {  
 let ch = self.char;  
 if ch == *EOF* {  
 return  
 }  
 self.pos.i += 1;  
 if self.char == '\n' {  
 self.pos.row += 1;  
 self.pos.col = 1;  
 } else {  
 self.pos.col += 1;  
 }  
 }  
  
 pub fn next\_token(&mut self) -> Token{  
 self.skip\_spaces();  
 if self.char == *EOF* {  
 return self.eof\_token();  
 }  
 if self.char == '"' {  
 return self.read\_str();  
 }  
 if self.char == '$' {  
 return self.read\_hex();  
 }  
 if is\_letter(self.char) {  
 return self.read\_ident();  
 }  
 if is\_digit(self.char) && !(self.char == '0') {  
 return self.read\_number();  
 }  
  
 return self.error\_token(self.char.to\_string());  
 }  
  
 pub fn read\_ident(&mut self) -> Token {  
 let start = self.pos.clone();  
 let mut ident\_vec: Vec<char> = Vec::*new*();  
  
 loop {  
  
 ident\_vec.push(self.char);  
  
 self.inc\_pos();  
 self.read\_char();  
  
  
 let ch = self.char;  
 if !(is\_letter(ch) || ch == '$') {  
 break;  
 }  
 }  
 let token\_type: TokenType;  
 let ident = String::*from\_iter*(ident\_vec);  
 let key = self.idents.get(&ident);  
 match key {  
 *Some*(key) => {  
 token\_type = TokenType::*IDENT*(\*key)  
 },  
 *None* => {  
 self.idents.insert(ident, self.ident\_i);  
 token\_type = TokenType::*IDENT*(self.ident\_i);  
 self.ident\_i += 1;  
 }  
 }  
  
 let mut end = self.pos.clone();  
 end.col -= 1;  
 return Token{  
 start,  
 end,  
 token\_type,  
 }  
 }  
  
 pub fn read\_number(&mut self) -> Token {  
 let start = self.pos.clone();  
 let mut number: i32 = 0;  
  
 loop {  
 let s = self.char.to\_string();  
 let ch = self.char;  
  
 if is\_break(ch) {  
 break;  
 }  
  
 if !is\_digit(ch){  
 return self.error\_token(number.to\_string());  
 }  
  
 number = number \* 10 + s.parse::<i32>().unwrap();  
 self.inc\_pos();  
 self.read\_char();  
  
  
 }  
 let mut end = self.pos.clone();  
 end.col -= 1;  
 return Token{  
 start: start,  
 end: end,  
 token\_type: TokenType::*NUMBER*(number),  
 }  
 }  
  
 pub fn read\_hex(&mut self) -> Token {  
 let start = self.pos.clone();  
 let mut number: i32 = 0;  
  
 self.inc\_pos();  
 self.read\_char();  
 loop {  
 let s = self.char.to\_string();  
 let ch = self.char;  
  
 if is\_break(ch) {  
 break  
 }  
 if !(is\_digit(ch) || 'a' <= ch && ch <= 'f' || 'A' <= ch && ch <= 'F') {  
 return self.error\_token(number.to\_string());  
 }  
  
 number = number \* 16 + i32::*from\_str\_radix*(&\*s, 16).unwrap();  
  
 self.inc\_pos();  
 self.read\_char();  
  
 }  
  
 let mut end = self.pos.clone();  
 end.col -= 1;  
  
 return Token{  
 start: start,  
 end: end,  
 token\_type: TokenType::*NUMBER*(number)  
 }  
 }  
  
 pub fn read\_str(&mut self) -> Token {  
 let start = self.pos.clone();  
 let mut str: Vec<char> = Vec::*new*();  
  
 self.inc\_pos();  
 self.read\_char();  
  
 loop {  
 if self.char == '"' {  
 self.inc\_pos();  
 self.read\_char();  
 if self.char != '"' {  
 break;  
 }  
 }  
  
 if self.char == '\\' {  
 self.inc\_pos();  
 self.read\_char();  
 if self.char == '\n' {  
 self.inc\_pos();  
 self.read\_char();  
 } else {  
 str.push(self.char);  
 return self.error\_token(String::*from\_iter*(str));  
 }  
 }  
  
 str.push(self.char);  
  
 self.inc\_pos();  
 self.read\_char();  
 }  
  
  
 let mut end = self.pos.clone();  
 end.col -= 1;  
  
 return Token {  
 start: start,  
 end: end,  
 token\_type: TokenType::*STR*(String::*from\_iter*(str)),  
 };  
 }  
  
 pub fn eof\_token(&mut self) -> Token {  
 self.read\_char();  
 return Token {  
 start: self.pos.clone(),  
 end: self.pos.clone(),  
 token\_type: TokenType::*EOF*,  
 }  
 }  
  
 pub fn error\_token(&mut self, left: String) -> Token {  
 let start = self.pos.clone();  
 let mut val: Vec<char> = Vec::*new*();  
 while ! is\_break(self.char) {  
 val.push(self.char);  
 self.inc\_pos();  
 self.read\_char();  
  
 }  
 return Token {  
 start: start,  
 end: self.pos.clone(),  
 token\_type: TokenType::*ERROR*(format!("{}{}", left, String::*from\_iter*(val))),  
 }  
 }  
  
  
 pub fn skip\_spaces(&mut self) {  
 let mut again = true;  
 while again {  
 again = false;  
  
 self.read\_char();  
 let ch = self.char;  
  
 if ch == *EOF* {  
 break;  
 }  
  
 if ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n'{  
 self.inc\_pos();  
 again = true;  
 }  
 }  
 }  
}  
  
fn main() {  
 let filename = "input";  
  
 let file = File::*open*(filename).unwrap();  
 let reader = BufReader::*new*(file);  
 let mut lines: Vec<String> = Vec::*new*();  
 for line in reader.lines() {  
 let line = line.unwrap();  
 lines.push(line);  
 }  
 lines.push(*EOF*.to\_string());  
 let program = lines.join("\n");  
 println!("{}\n\n", program);  
  
 let mut lexer = Lexer {  
 input: program.chars().collect(),  
 pos: Position{  
 row: 1,  
 col: 1,  
 i: 0  
 },  
 char: '^',  
 token: TokenType::*EOF*,  
 idents: HashMap::*new*(),  
 ident\_i: 0,  
 };  
 loop {  
 let token = lexer.next\_token();  
 println!("{}", token.to\_string());  
 match token.token\_type {  
 TokenType::*EOF* => break,  
 \_ => {}  
 }  
 }  
 for key in lexer.idents.keys() {  
 println!("{}: {}", key, lexer.idents.get(key).unwrap())  
 }  
}

# 4 Особенности языка Rust

Из основных особенностей rust следует выделить кодовый анализатор, позволяющих отлавливать ошибки утечки памяти и при работе с многопоточностью, отсутствие сборщика мусора и мощную статическую типизацию. Ключевые приоритеты языка: безопасность и скорость.

Отличительная черта компилятора Rust – знание о том. Кто владеет определенной сущностью (owner), кто лишь одалживает (mutable borrow) и кто лишь просматривает (immutable borrow).

Атрибуты деривации (derive), позволяющие автоматически генерировать новые элементы для структур данных. В программе данная особенность использовалась при работе с позициями

#[derive(Copy, Clone)]  
pub struct Position {  
 row: i32,  
 col: i32,  
 i: usize  
}

По сути, derive заменяет собственноручную реализацию copy и clone в данном примере.

Option перечисление: использование Some и None – способ выразить отсуствие чего бы то ни было.

pub enum Option<T> {

None,

Some(T),

}

Для предоставление безопасного доступа

Например, использовалось в лабораторной для работы с hashmap idents

match key {  
 *Some*(key) => {  
 token\_type = TokenType::*IDENT*(\*key)  
 },  
 *None* => {  
 self.idents.insert(ident, self.ident\_i);  
 token\_type = TokenType::*IDENT*(self.ident\_i);  
 self.ident\_i += 1;  
 }

Метод unvwap, по сути, вернуть результат вычислений, а если произошла ошибка, то panic!!!. Возвращает тип Option<t>

Использовалось в лабораторной повсеместно.

Интересные альтернатив: unwrap\_or – вернуть значение по умолчанию в случае None (не нашел случаев, где было бы к месту)

Rust показался интересным языком, позволяющим писать безопасный и быстрый код. В перспективе, он может быть применим для написания ядер ОС.

# 5 Тестирование

Вход:

123  
 32a1  
 abc$a  
 bcde  
 bcde  
$a1  
 $f  
"simple"  
"str with """  
"str with \  
gap"  
  
"as bas\"

Результат:

NUMBER: 123 (1, 1)-(1, 3)

error: 32a1 (2, 5)-(2, 7)

IDENT: 0 (3, 2)-(3, 6)

IDENT: 1 (4, 2)-(4, 5)

IDENT: 1 (5, 2)-(5, 5)

NUMBER: 161 (6, 1)-(6, 3)

NUMBER: 15 (7, 3)-(7, 4)

STR: simple (8, 1)-(8, 8)

STR: str with " (9, 1)-(9, 13)

STR: str with gap (10, 1)-(11, 4)

error: as bas"" (13, 9)-(13, 10)

EOF (14, 1)-(14, 1)

bcde: 1

abc$a: 0

# Вывод

В результате выполнения данной лабораторной работы были приобретены навыки реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.