Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №4**

**Дисциплина: Методы разработки трансляторов**

**Тема: «Построение синтаксического анализатора»**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Э. Д. Козлов

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Е. Усов

**Цель:** построение синтаксического анализатора для исходного Java кода методом рекурсивного спуска.

**Описание программы**

Программа реализует синтаксический анализатор методом рекурсивного спуска для подмножества языка программирования C. Анализатор получает на вход файл tokens\_output.txt, содержащий список токенов, сгенерированных лексическим анализатором (scaner.py), и проверяет синтаксическую корректность программы на языке C.

Анализатор поддерживает следующие конструкции:

* директиву препроцессора #include <...>;
* определение функции main с заголовком int main() { ... };
* объявления переменных типа int с возможной инициализацией (например, int x = 10;);
* оператор return с числом или переменной;
* условную конструкцию if (...) return ...; с простым логическим выражением;

При обнаружении ошибки синтаксический анализатор выводит сообщение, включающее:

* тип ошибки (например, Ожидался тип 'num', найдено 'id');
* номер строки, на которой произошла ошибка;
* ожидаемое и фактически полученное значение или тип токена.

Анализ завершается сообщением об успешной проверке при отсутствии ошибок.

**Приложение**

Исходный С-код:

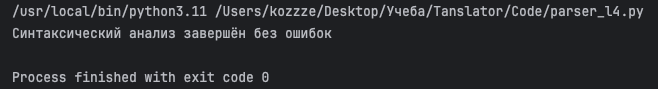
#include <stdio.h>  
int main() {  
 int x = 10;  
 int y = 192;  
 if (x+1 == 2) return 1;  
 return 0;  
}

Синтаксический анализатор:

token\_map = {  
 "W1": "int", "W2": "char", "W3": "float", "W4": "double", "W5": "return",  
 "W6": "if", "W7": "else", "W8": "for", "W9": "while", "W10": "do",  
 "W11": "switch", "W12": "case", "W13": "break", "W14": "continue",  
 "W15": "default", "W16": "void", "W17": "static", "W18": "struct",  
 "W19": "typedef", "W20": "union", "W21": "unsigned", "W22": "signed",  
 "W23": "long", "W24": "short", "W25": "goto", "W26": "sizeof",  
 "W27": "main", "W28": "printf", "W29": "scanf", "W30": "#include",  
  
 "O1": "==", "O2": "!=", "O3": "<=", "O4": ">=", "O5": "&&", "O6": "||",  
 "O7": "+=", "O8": "-=", "O9": "\*=", "O10": "/=", "O11": "%=",  
 "O12": "+", "O13": "-", "O14": "\*", "O15": "/", "O16": "%",  
 "O17": "=", "O18": "<", "O19": ">",  
  
 "R1": "(", "R2": ")", "R3": "{", "R4": "}", "R5": "[", "R6": "]",  
 "R7": ",", "R8": ";", "R9": ".",  
}  
  
def classify\_token(value):  
 if value in {"#", "include", "int", "main", "return", "printf", "scanf", "if"}:  
 return "keyword"  
 elif value.startswith('"'):  
 return "str"  
 elif value.isdigit():  
 return "num"  
 elif value in {"<", ">", "(", ")", "{", "}", ";", ",", ".", "=",  
 "==", "!=", "<=", ">=", "+", "-", "\*", "/", "%"}:  
 return "symbol"  
 else:  
 return "id"  
  
class Token:  
 def \_\_init\_\_(self, line, code):  
 self.line = line  
 self.code = code  
 self.lexeme = token\_map.get(code, code)  
 self.type = classify\_token(self.lexeme)  
  
 def \_\_repr\_\_(self):  
 return f"{self.lexeme} ({self.type}) @ {self.line}"  
  
class Tokenizer:  
 def \_\_init\_\_(self, filename):  
 self.tokens = self.load(filename)  
 self.pos = 0  
  
 def load(self, filename):  
 tokens = []  
 with open(filename, "r") as f:  
 for lineno, line in enumerate(f, 1):  
 for code in line.strip().split():  
 tokens.append(Token(lineno, code))  
 return tokens  
  
 def current(self):  
 return self.tokens[self.pos] if self.pos < len(self.tokens) else None  
  
 def next(self):  
 self.pos += 1  
  
 def expect(self, lexeme=None, type\_=None):  
 tok = self.current()  
 if not tok:  
 raise SyntaxError("Ожидался токен, но достигнут конец файла")  
 if lexeme and tok.lexeme != lexeme:  
 raise SyntaxError(f"[Строка {tok.line}] Ожидалось '{lexeme}', найдено '{tok.lexeme}'")  
 if type\_:  
 if isinstance(type\_, str) and tok.type != type\_:  
 raise SyntaxError(f"[Строка {tok.line}] Ожидался тип '{type\_}', найдено '{tok.type}'")  
 if isinstance(type\_, set) and tok.type not in type\_:  
 raise SyntaxError(f"[Строка {tok.line}] Ожидался один из типов {type\_}, найдено '{tok.type}'")  
 self.next()  
 return tok  
  
class SyntaxAnalyzer:  
 def \_\_init\_\_(self, tokenizer):  
 self.tok = tokenizer  
  
 def parse(self):  
 self.parse\_program()  
 print("Синтаксический анализ завершён без ошибок")  
  
 def parse\_program(self):  
 self.parse\_includes()  
 self.parse\_main()  
  
 def parse\_includes(self):  
 while self.tok.current() and self.tok.current().lexeme == "#include":  
 self.tok.expect("#include")  
 self.tok.expect("<")  
 self.tok.expect(None, {"id", "num"}) # stdio.h  
 self.tok.expect(">")  
  
 def parse\_main(self):  
 self.tok.expect("int")  
 self.tok.expect("main")  
 self.tok.expect("(")  
 self.tok.expect(")")  
 self.tok.expect("{")  
 self.parse\_statements()  
 self.tok.expect("}")  
  
 def parse\_statements(self):  
 while self.tok.current() and self.tok.current().lexeme != "}":  
 self.parse\_statement()  
  
 def parse\_declaration(self):  
 self.tok.expect("int")  
 self.tok.expect(None, "id")  
 self.tok.expect("=")  
 self.tok.expect(None, {"num", "id"})  
 self.tok.expect(";")  
  
 def parse\_statement(self):  
 current = self.tok.current()  
 if current.lexeme == "int":  
 self.parse\_declaration()  
 elif current.lexeme == "printf":  
 self.tok.expect("printf")  
 self.tok.expect("(")  
 self.tok.expect(None, "str")  
 self.tok.expect(")")  
 self.tok.expect(";")  
 elif current.lexeme == "return":  
 self.tok.expect("return")  
 self.tok.expect(None, {"num", "id"})  
 self.tok.expect(";")  
 elif current.lexeme == "if":  
 self.parse\_if()  
 elif current.lexeme == "while":  
 self.parse\_while()  
  
 else:  
 raise SyntaxError(f"[Строка {current.line}] Неожиданный токен: '{current.lexeme}'")  
  
 def parse\_if(self):  
 self.tok.expect("if")  
 self.tok.expect("(")  
 self.tok.expect(None, {"id", "num"})  
 self.tok.expect("+")   
 self.tok.expect(None, {"id", "num"})  
 self.tok.expect("==")  
 self.tok.expect(None, {"id", "num"})  
 self.tok.expect(")")  
 self.tok.expect("return")  
 self.tok.expect(None, {"num", "id"})  
 self.tok.expect(";")  
  
 def parse\_while(self):  
 self.tok.expect("while")  
 self.tok.expect("(")  
 self.tok.expect(None, {"id", "num"})  
 self.tok.expect("<")   
 self.tok.expect(None, {"id", "num"})  
 self.tok.expect(")")  
 self.tok.expect("{")  
 self.parse\_statements()  
 self.tok.expect("}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 try:  
 tokenizer = Tokenizer("tokens\_output.txt")  
 analyzer = SyntaxAnalyzer(tokenizer)  
 analyzer.parse()  
 except SyntaxError as e:  
 print("Синтаксическая ошибка:")  
 print(e)

Результат работы программы:

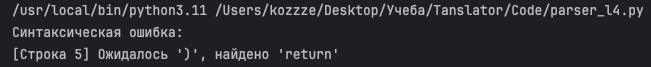
1. Без ошибок в исходном коде:



1. Убираем скобку в условии if:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание



**Вывод:** Разработан синтаксический анализатор для подмножества языка C. Программа реализует метод рекурсивного спуска, выполняет синтаксическую проверку исходного кода на основе заранее сформированных токенов и отображает сообщения об ошибках с указанием номера строки, типа и содержания нарушения.