Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме «Лексический анализатор»

Выполнил:
студент гр. 953501
Кореневский С. А.

Проверил:
ст. преподаватель Шиманский В. В.

Минск 2022

Содержание

1 Постановка задачи

Освоение работы с существующими лексическими анализаторами (по желанию). Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Определяются лексические правила. Выполняется перевод потока символов в поток лексем (токенов).

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

На вход программы подается текстовый файл (напр. с именем INPUT.TXT), содержащий строки символов анализируемой программы. Например, строка может задавать переменной значения арифметического выражения в виде

ПЕРЕМЕННАЯ = ВЫРАЖЕНИЕ.

Выражение может включать:

- Знаки сложения и умножения («+» и «*»);
- Круглые скобки («(» и «)»);
- Константы (например, 5; 3.8; 1e+18, 8.41E–10);
- Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы.

Разбор выражения *COST* = (*PRICE*+*TAX*)*0.98.

Проанализируем выражение:

- *COST*, *PRICE* и *TAX* – лексемы-идентификаторы;
- 0.98 – лексема-константа; =, +, * – просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем $<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>$.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) – показана в виде индексов. Символы «=», «+» и «*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

РАБОТА С ТАБЛИЦЕЙ ИМЁН

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имён.

Для нашего примера *COST*, *PRICE* и *TAX* – переменные с плавающей точкой. Рассмотрим вариант такой таблицы. В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 1).

Номер элемента	Идентификатор	Информация
1	*COST*	Переменная с плавающей точкой
2	*PRICE*	Переменная с плавающей точкой
3	*TAX*	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

Табл. 1 – Таблица имён

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

Возможно создание таблиц констант, ключевых слов, разделителей, логических/математических операторов

При определении неверной последовательности символов, необходимо обнаружить эту ошибку и выдать сообщение о ней. Всего необходимо показать скриншоты нахождения 4-х **лексических** ошибок.

2 Теория

В информатике лексический анализ («токенизация», от англ. tokenizing) — 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 интерпретаторах исходного кода языков программирования, и в различных парсерах слов 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 грамматического

анализатора, и избавить его от определения лексических подробностей в контекстно-свободной грамматике (что привело бы к усложнению грамматики).

3 Созданный лексический анализатор

Код исследуемой программы в файле подается на вход программы-анализатора и считывается построчно. С помощью строки регулярных выражений в каждой строке выделяются следующие типы токенов:

- Комментарии
- Строковые константы
- Целочисленные константы
- Константы типа float
- Символы-операторы, слова (потенциальные ID).

Таким образом ни один символ входного файла не будет потерян из-за несоответствия какой-то части заданных правил регулярных выражений.

4 Демонстрация

Код программы для анализа.

```
data = """
void SortAlgo::mergeSort(int data[], int lenD)\n
{\n
  iff (lenD>1){\n
    int middle = lenD/2;\n
    int rem = lenD-middle;\n
    int *L = new int [middle];\n
    int *R = new int [rem];\n
    for (int i=0;i<lenD;i++){\n
      if (i<middle){\n
        L[i] = data[i];\n
      }\n
      elsee {\n
        R[i-middle] = data[i];\n
      }\n
    }\n
    mergeSort@(L,middle);\n
    mergeSort(R,rem);\n
    merge(data, lenD, L, middle, R, rem);\n
  }\n
}\n

voi SortAlgo::merge(int merged[], int lenD, int L[], int lenL, int R[], int
lenR){\n
  inttt i = 0;\n
  int j = 0;\n
  while123(i<lenL||j<lenR){\n
    iff (i<lenL & j<lenR){\n
      if (L[i]<=R[j]){\n
        merged[i+j] === L[i];\n
        i++;\n
      }\n
      else {\n
        merged[[i+j] = R[j];\n
        j++;\n
      }\n
    }\n
    else if (i<lenL){\n
      merged[i+j] =!= L[i];\n
      i++;\n
    }\n
    elses if (j<lenR){\n
      merged [[[ i+j ] = R[ j ];\n
      j++;\n
    }\n
  }\n
}\n
"""
```

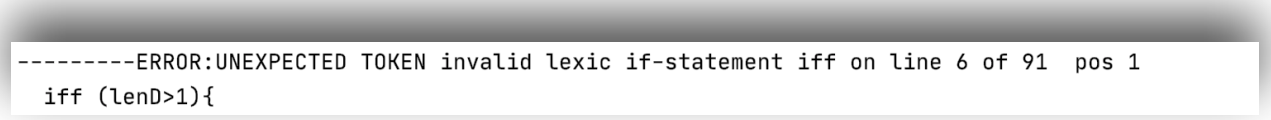

Рис 1. Ошибка неправильный оператор iff вместо if

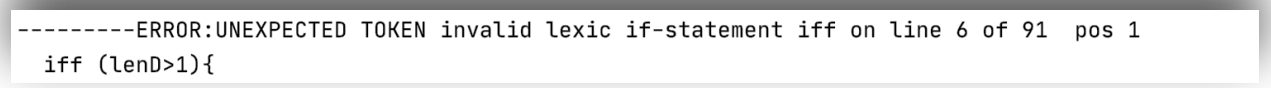

Рис 2. Ошибка неправильный оператор iif вместо if

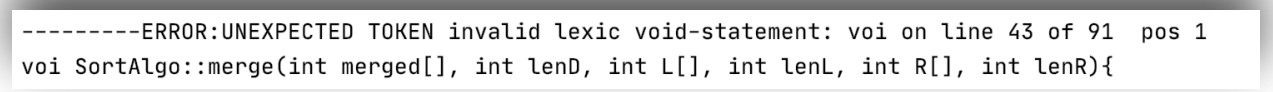

Рис 3. Ошибка неправильное ключевое слово voi вместо void

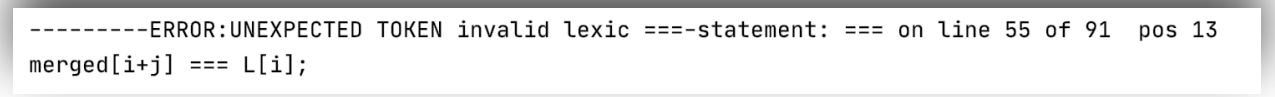

Рис 4. Ошибка неправильный оператор ===

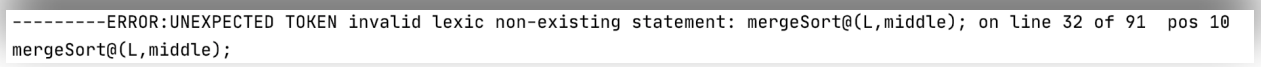

Рис 5. Ошибка несуществующий токен mergeSort@ (лишний символ)

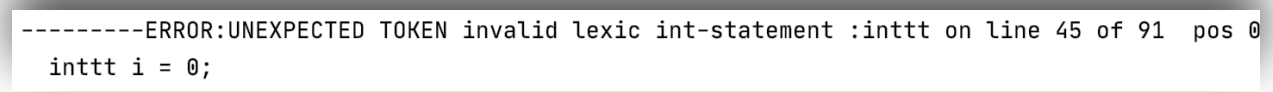

Рис 6. Ошибка неправильное ключевое слово inttt вместо int

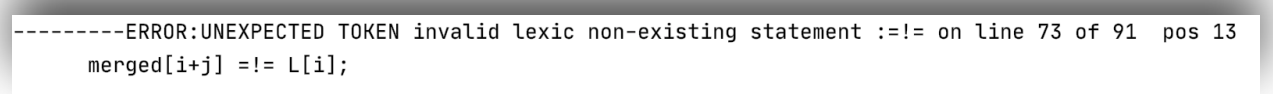

Рис 7. Ошибка несуществующий токен =!= (лишний символ !)

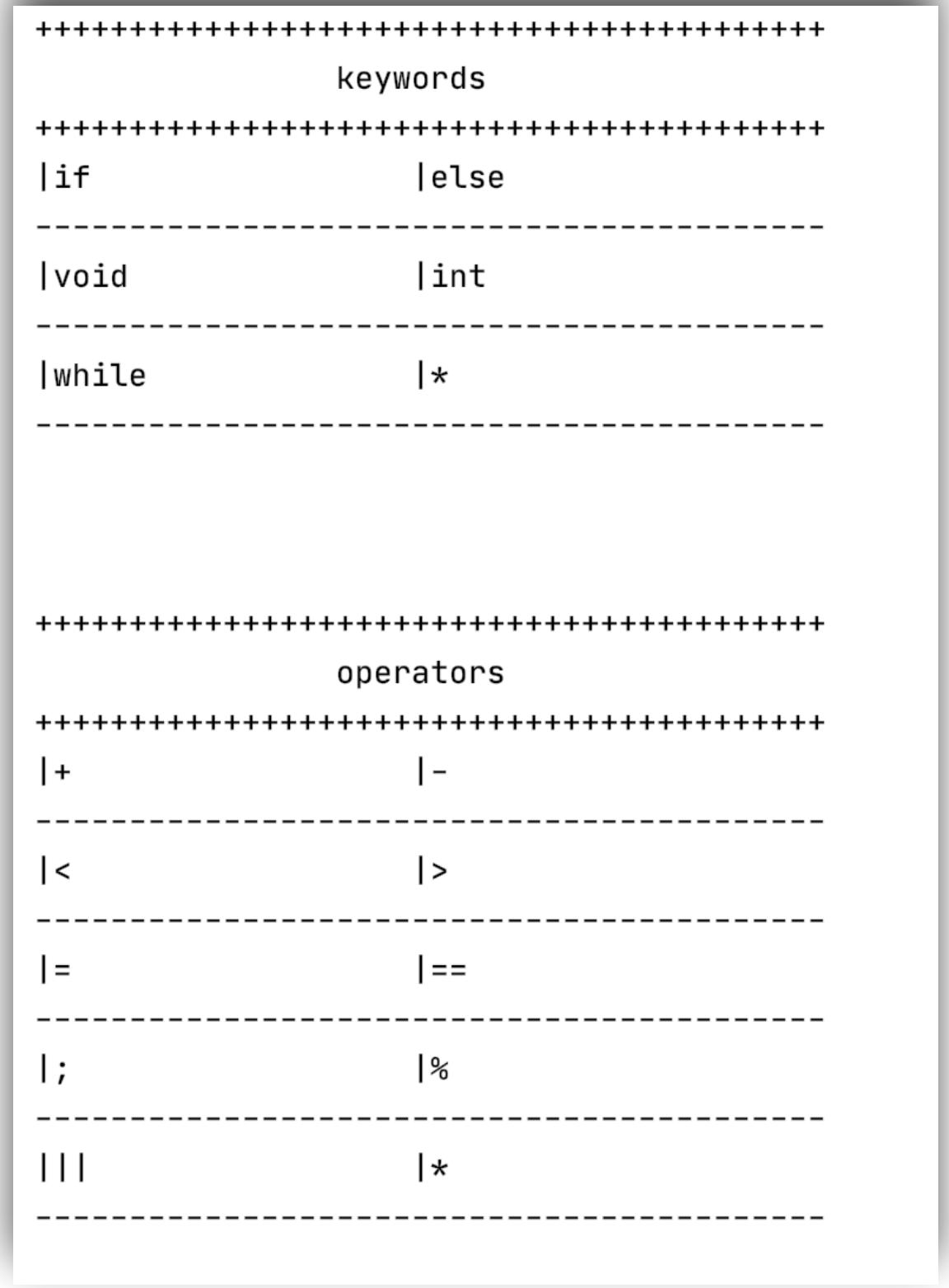

Рис 8. Вывод таблицы имен

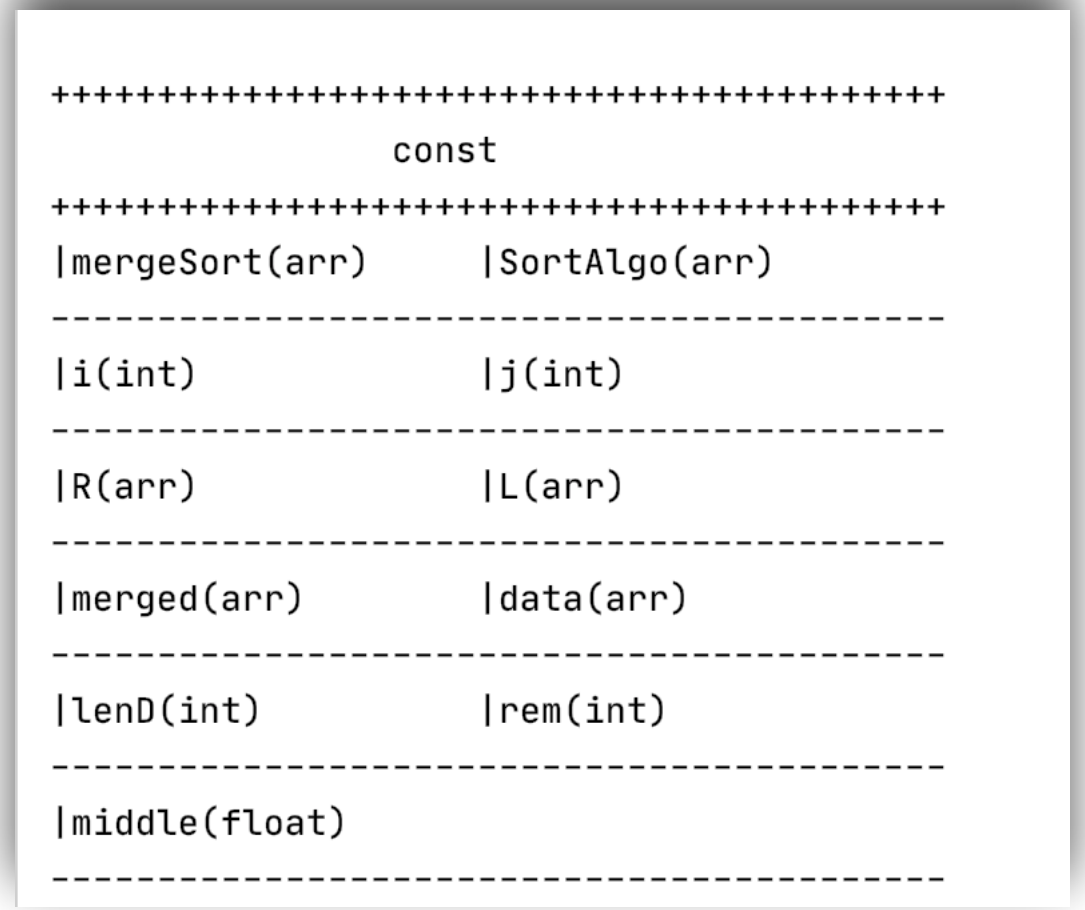

Рис 9. Вывод таблицы имен 2

5 Выводы

Был разработан лексический анализатор подмножества языка C++. Получены знания о принципах работы, обязанностях и особенностях построения лексических анализаторов языков программирования, описанных в отчёте лабораторной работы.

6 Код программы

```
from data import data
import re

pattern = ["void", "int", "while", "if", "else"]
common_mistake = ["==","++","--"]
i = 0
j = 1

n = len(data.split("\n"))
for token in data.split("\n"):
    # print(token,"\n", token.split(),"\n")
    i+=1
    for el in token.split():

        if (len(el) >= 2 and el[0] == "i" and el[1] == "f") or (len(el) >= 3
and el[0] == "i" and el[1] == "i" and el[0] == "f"):
            if len(el) != 2:
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic if-
statement {el} on line {i} of {n}  pos {1}")
                print(token)

                j += 1
                break

        elif el[0] == "v" and el[1] == "o":
            if len(el) != 4 or el[2] != "i" or el[3] != "d":
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic void-
statement: {el} on line {i} of {n}  pos {1}")
                print(token)
                j += 1
                break
        elif el[0] == "=":
            if len(el) > 2 or el[-1] != "=":
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic ===-
statement: {el} on line {i} of {n}  pos {13}")
                print("merged[i+j] === L[i];")

                j += 1
        elif len(el)> 10 and el[9] == "@":
            if len(el) > 2 or el[-1] != "=":
                print(
                    f"---------ERROR:UNEXPECTED TOKEN invalid lexic non-
existing statement: {el} on line {i} of {n}  pos {10}")
                print(el)

                j += 1

        elif len(el) > 2 and el[0] == "i" and el[1] == "n":
            if len(el) != 3 or el[-1] != "t":
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic int-
statement :{el} on line {i} of {n}  pos {0}")
                j += 1
                print(el)
                print(token)

        elif el[0] == "[":
            if len(el) >= 2 and el[1] == "[":
```

```
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic []-
statement :{el} on line {i} of {n}  pos {0}")
                j += 1
                print(token)
        elif el[0] == "]":
            if len(el) >= 2 and el[1] == "]":
                print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic []-
statement :{el} on line {i} of {n}  pos {0}")
                j += 1
                print(token)
        elif el == '=!=':
            print(f"---------ERROR:UNEXPECTED TOKEN invalid lexic non-
existing statement :{el} on line {i} of {n}  pos {13}")
            j += 1
            print(token)
```