Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ
к лабораторной работе №2
на тему

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Выполнил: студент гр.253504
Фроленко К.Ю.
Проверил: ассистент кафедры информатики
Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВК ЗАДАЧИ

Цель данной лабораторной работы заключается в освоении принципов лексического анализа и разработке собственного анализатора для подмножества языка программирования, определённого в предыдущей работе. В ходе исследования необходимо изучить существующие подходы к лексическому анализу и определить набор лексических правил, которые позволят преобразовывать поток символов исходного кода в последовательность осмысленных лексем.

Исходный текст программы представляет собой поток символов, в котором могут встречаться имена переменных, арифметические выражения, операторы, скобки и другие элементы синтаксиса. Например, при разборе выражения, задающего переменной значение, элементы вроде имени переменной, арифметических операторов, констант и разделителей должны быть распознаны как отдельные лексемы. При этом идентификаторы представляют собой последовательности букв и цифр, начинающиеся с буквы, а константы могут быть числовыми, строковыми или битовыми.

Разработанный анализатор обязан не только преобразовывать входной поток символов в поток лексем, но и организовывать работу с таблицами символов. В частности, информация об идентификаторах и константах должна сохраняться в специальных таблицах, что позволяет при повторном обнаружении идентификатора ссылаться на уже существующую запись. Кроме того, программа должна уметь обрабатывать ключевые слова и операторы, распознавая их на основе заданных списков, аналогичных тем, что используются в коде (например, *PROCEDURE*, *OPTIONS*, *DECLARE*, *VAR*, *STATIC*, *INITIAL*, *FILE* и другие для ключевых слов, а также набор стандартных операторов и разделителей).

Особое внимание уделяется обработке ошибок. Если в процессе анализа обнаруживается неверная последовательность символов, например, незакрытая строковая константа, неправильное использование скобок или неопознанный символ, анализатор должен генерировать информативное сообщение об ошибке. В отчёте лабораторной работы требуется продемонстрировать примеры обнаружения не менее четырёх различных лексических ошибок.

Реализация данного лексического анализатора осуществляется с помощью языка *R*. Программа использует регулярные выражения для распознавания различных типов лексем и формирует итоговый поток токенов, группируя их по категориям, таким как идентификаторы, константы, ключевые слова, операторы и разделители. Такой подход позволяет не только автоматизировать процесс анализа исходного кода, но и создать эффективный инструмент для последующего синтаксического и семантического разбора программ на языке *PL/1*.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализ – это фундаментальный этап обработки исходного кода, в ходе которого непрерывная последовательность символов преобразуется в поток лексем, или «слов». Этот процесс играет ключевую роль в трансляции программ, так как именно на этом этапе происходит первичная структуризация информации, что в дальнейшем облегчает синтаксический и контекстный анализ. Исходный текст делится на отдельные составляющие: символы, входящие в состав лексем, и символы-разделители, такие как пробелы, знаки препинания и специальные символы, которые помогают определить границы лексем. Важно отметить, что в некоторых языках, например, в *PL/1*, отдельные лексемы могут содержать незначащие символы, такие как пробелы внутри строковых литералов, что требует особого внимания при реализации анализатора [1].

Лексемы, как правило, подразделяются на несколько категорий. Основными классами являются числовые литералы (целые числа, числа с плавающей точкой, а также литералы, представленные в восьмеричной или шестнадцатеричной системах счисления), идентификаторы, строковые литералы, а также ключевые слова и символы пунктуации, которые часто называют ограничителями. Ключевые слова представляют собой фиксированное подмножество идентификаторов, имеющих специальное значение в языке программирования. Такой подход позволяет уже на ранних этапах определить, какие элементы текста являются управляющими конструкциями, а какие – переменными или константами [2].

Процесс лексического анализа начинается с последовательного выделения лексем из входного потока символов. При этом, если выделенная последовательность соответствует символу-разделителю или ограничителю, она немедленно фиксируется в виде отдельного элемента. Если же выделенная последовательность выглядит как идентификатор, происходит дополнительная проверка на предмет её принадлежности к множеству ключевых слов. При обнаружении повторного вхождения идентификатора информация о нем сохраняется в таблице символов, что позволяет использовать уже зарегистрированную запись для дальнейшей обработки. Аналогичным образом обрабатываются и числовые, и строковые литералы, для которых дополнительно сохраняется их конкретное значение для последующего семантического анализа [3].

Современные лексические анализаторы могут работать в двух режимах. Они могут быть реализованы как самостоятельная фаза трансляции, генерирующая отдельный файл с лексемами, или как компонент, который по запросу выдает следующую лексему. В первом случае результатом работы анализатора является уже готовая для синтаксического разбора структура, а во втором – лексема возвращается непосредственно в момент вызова функции, что позволяет динамически управлять потоком анализа. Такой гибкий подход обеспечивает эффективную интеграцию лексического анализа с последующими этапами компиляции, позволяя передавать как

структурированные данные о классах лексем, так и конкретные значения идентификаторов, чисел и строк [4].

Помимо выделения отдельных лексем, анализатор обязан строить таблицы объектов, в которых аккумулируется информация об идентификаторах, числовых и строковых литералах. Это существенно облегчает последующую семантическую обработку кода, так как все повторяющиеся элементы регистрируются только один раз, и их использование в различных частях программы осуществляется посредством ссылок на соответствующие записи в таблицах. Такой механизм не только оптимизирует процесс анализа, но и способствует повышению надежности и эффективности всей системы трансляции.

Кроме того, лексический анализ часто реализуется с использованием мощных инструментов, таких как конечные автоматы, регулярные выражения и праволинейные грамматики. Все эти методы обладают эквивалентной выразительной мощностью, что позволяет, используя заданное регулярное выражение или грамматику, автоматически строить конечный автомат, способный распознавать те же языковые конструкции. Это является важным преимуществом, так как позволяет значительно упростить процесс разработки анализатора, сделать его более модульным и легко расширяемым для поддержки новых языковых конструкций.

Таким образом, лексический анализатор выполняет две ключевые функции: он формирует последовательность лексем, которая служит основой для построения синтаксического дерева, и аккумулирует детальную информацию о каждом элементе исходного кода, что является фундаментом для последующего контекстного анализа. Такой комплексный подход обеспечивает высокую эффективность трансляции программ и является залогом успешного функционирования современных систем компиляции.

3 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

В ходе выполнения лабораторной работы был разработан лексический анализатор для разбора исходного кода на языке *PL/1*. Программа осуществляет выделение лексем из входного файла, распознавая идентификаторы, константы, ключевые слова, операторы и разделители. Анализатор аккумулирует всю полученную информацию в виде таблиц, что позволяет наглядно оценить корректность работы алгоритма.

На рисунке 3.1 представлен анализируемый код, подготовленный для тестирования работы разработанного анализатора.

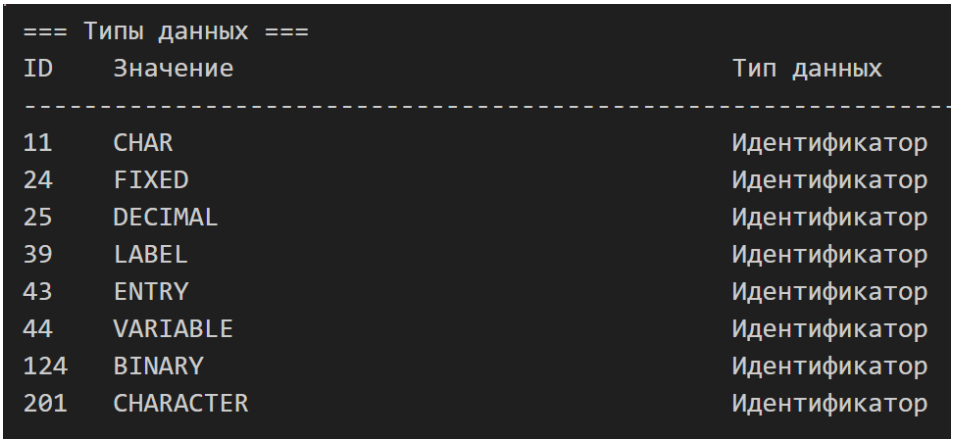
```
test: PROCEDURE OPTIONS(main);

   DECLARE S1 CHAR(*) VAR STATIC INITIAL('yes');
   DECLARE S2 FIXED DECIMAL(10,5) STATIC INITIAL(3674.799);
   DECLARE S8 LABEL;
   DECLARE S9 ENTRY VARIABLE;

   S9 = simple_procedure;

   PUT SKIP LIST('S1 = ', S1);
   PUT SKIP LIST('S2 = ', S2);

   S2 = S2 + 100;
   PUT SKIP LIST('После прибавления 100, S2 = ', S2);

   IF S1 = 'yes' THEN S8 = geometric_mean;
   GOTO S8;

geometric_mean:
   PUT SKIP LIST('Переход в geometric_mean');
   CALL S9;

   simple_procedure: PROCEDURE;

      PUT SKIP LIST('Вызвана simple_procedure');
   END simple_procedure;

   DECLARE I FIXED BINARY(10) STATIC INITIAL(1);

   DO WHILE (I < 5);
      I = I + 1;
      IF MOD(I, 2) = 0 THEN CONTINUE;
      PUT SKIP LIST('Нечётное значение I: ', I);
   END;

   I = 1;
   DO I = 1 TO 5;
      PUT SKIP LIST('Цикл DO TO, I = ', I);
   END I;

   DECLARE A(2,2) CHARACTER(2);
   DECLARE 1 Person,
      2 Name CHARACTER(20),
      2 Age FIXED BINARY;

   A(1,1) = 'A1';
   A(1,2) = 'B1';
   A(2,1) = 'A2';
   A(2,2) = 'B2';

   Person.Name = 'Иван';
   Person.Age  = 25;

   PUT SKIP LIST('Элемент массива A(1,1): ', A(1,1));
   PUT SKIP LIST('Структура Person: ', Person.Name, Person.Age);

END test;
```

Рисунок 3.1 – Анализируемый код

В результате работы программы были сформированы следующие таблицы:

1 Таблица типов данных. На рисунке 3.2 приведена таблица, содержащая информацию об именах типов данных, таких как *CHAR*, *FIXED*,

DECIMAL и других. Данная таблица демонстрирует, как из исходного кода выделяются конструкции, относящиеся к типам данных.

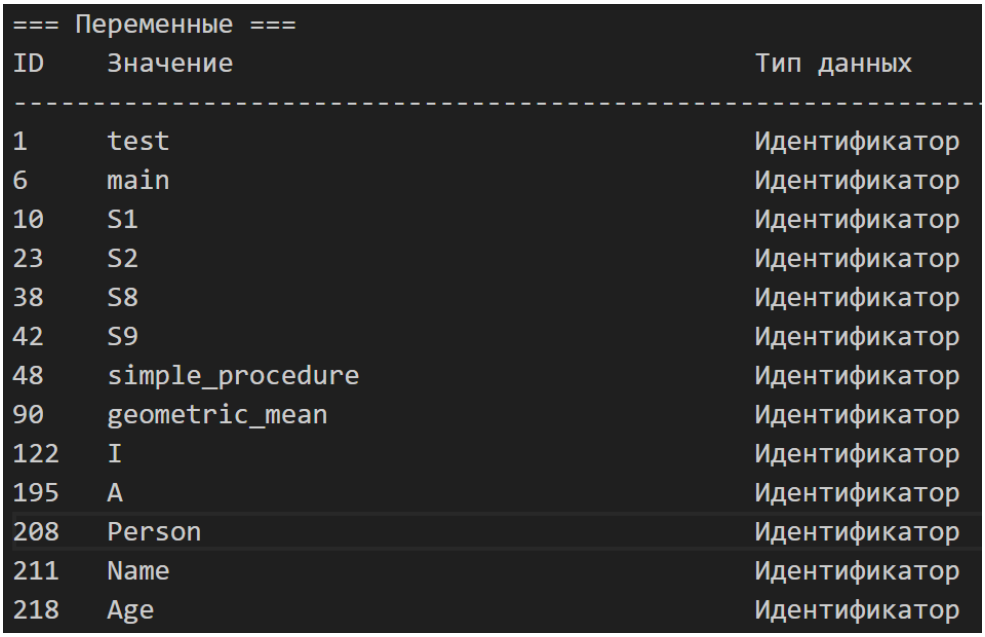


=== Типы данных ===

ID	Значение	Тип данных
11	CHAR	Идентификатор
24	FIXED	Идентификатор
25	DECIMAL	Идентификатор
39	LABEL	Идентификатор
43	ENTRY	Идентификатор
44	VARIABLE	Идентификатор
124	BINARY	Идентификатор
201	CHARACTER	Идентификатор

Рисунок 3.2 – Таблица типов данных

2 Таблица переменных. Рисунок 3.3 иллюстрирует таблицу переменных, в которой отражены все идентификаторы, использованные в анализируемом коде, например: *test*, *main*, *S1*, *S2*, *S8*, *S9*, *simple_procedure*, *geometric_mean*, *I*, *A*, *Person*, *Name*, *Age*. Таблица демонстрирует регистрацию каждого уникального идентификатора.

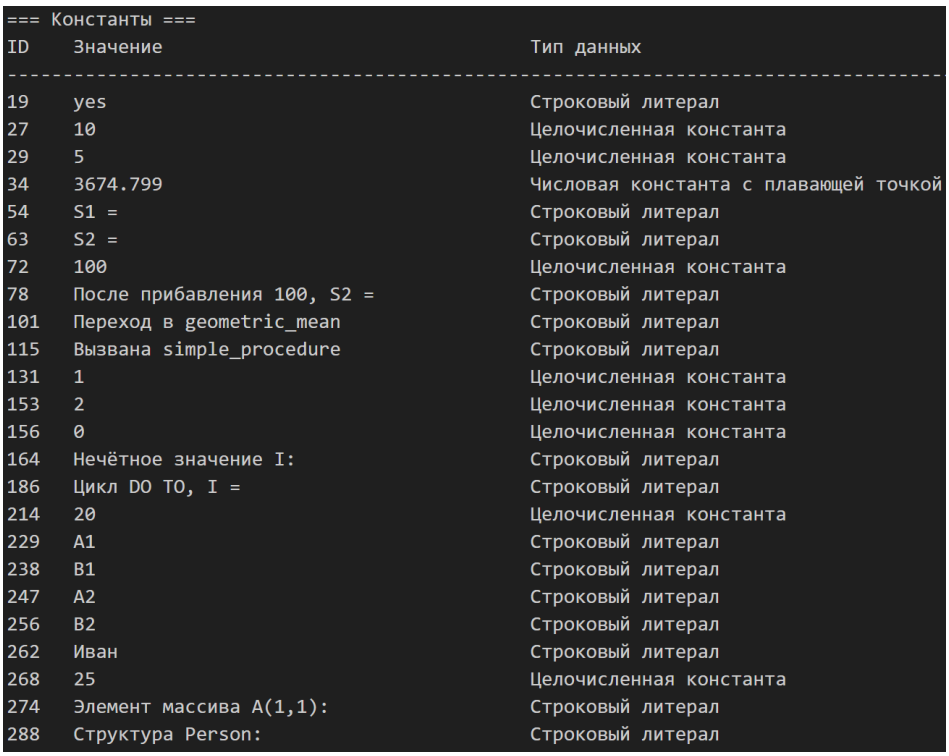


=== Переменные ===

ID	Значение	Тип данных
1	test	Идентификатор
6	main	Идентификатор
10	S1	Идентификатор
23	S2	Идентификатор
38	S8	Идентификатор
42	S9	Идентификатор
48	simple_procedure	Идентификатор
90	geometric_mean	Идентификатор
122	I	Идентификатор
195	A	Идентификатор
208	Person	Идентификатор
211	Name	Идентификатор
218	Age	Идентификатор

Рисунок 3.3 – Таблица переменных

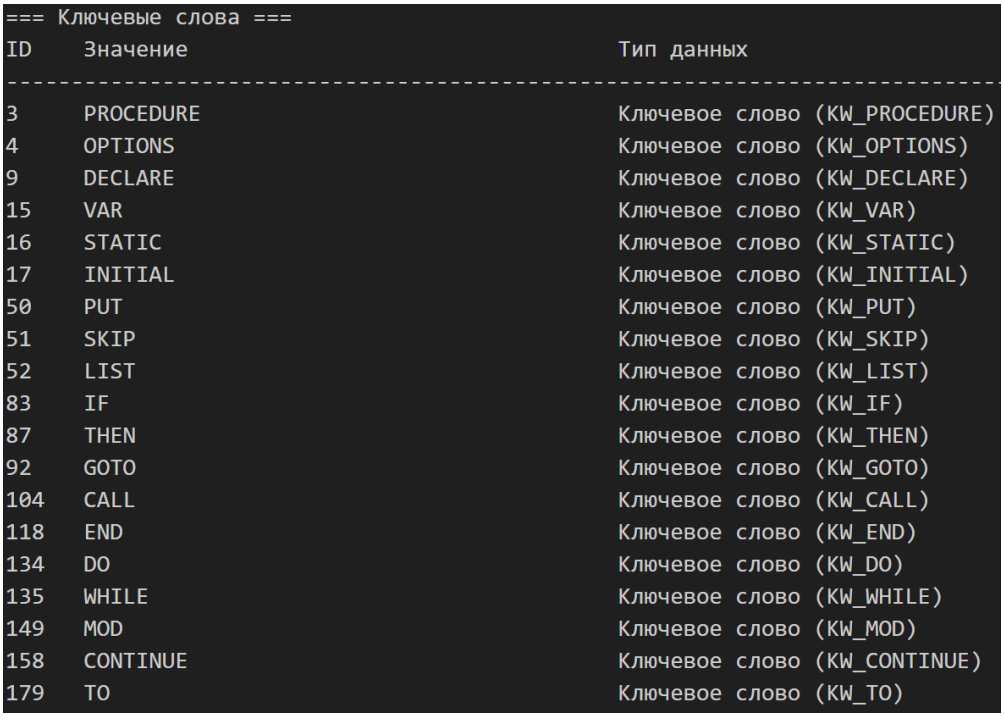
3 Таблица констант**.** На рисунке 3.4 представлена таблица констант, где отображаются числовые и строковые литералы, обнаруженные в исходном коде. Каждая константа сопровождается информацией о её типе (целочисленная, числовая с плавающей точкой или строковый литерал) и уникальным идентификатором.



```
=== Константы ===
ID    Значение                                  Тип данных
-------------------------------------------------------------------------------
19    yes                                       Строковый литерал
27    10                                        Целочисленная константа
29    5                                         Целочисленная константа
34    3674.799                                  Числовая константа с плавающей точкой
54    S1 =                                      Строковый литерал
63    S2 =                                      Строковый литерал
72    100                                       Целочисленная константа
78    После прибавления 100, S2 =               Строковый литерал
101   Переход в geometric_mean                  Строковый литерал
115   Вызвана simple_procedure                  Строковый литерал
131   1                                         Целочисленная константа
153   2                                         Целочисленная константа
156   0                                         Целочисленная константа
164   Нечётное значение I:                      Строковый литерал
186   Цикл DO TO, I =                           Строковый литерал
214   20                                        Целочисленная константа
229   A1                                        Строковый литерал
238   B1                                        Строковый литерал
247   A2                                        Строковый литерал
256   B2                                        Строковый литерал
262   Иван                                      Строковый литерал
268   25                                        Целочисленная константа
274   Элемент массива A(1,1):                   Строковый литерал
288   Структура Person:                         Строковый литерал
```

Рисунок 3.4 – Таблица констант

4 Таблица ключевых слов**.** Рисунок 3.5 демонстрирует таблицу ключевых слов, в которой отражены все зарезервированные слова языка, такие как *PROCEDURE*, *OPTIONS*, *DECLARE*, *VAR*, *STATIC*, *INITIAL*, *PUT*, *SKIP*, *LIST*, *IF*, *THEN*, *GOTO*, *CALL*, *END*, *DO*, *WHILE*, *MOD*, *CONTINUE*, *REPEAT*, *ELSE*, *LEAVE*, *TO* и другие.



```
=== Ключевые слова ===
ID    Значение                                  Тип данных
-------------------------------------------------------------------------------
3     PROCEDURE                                 Ключевое слово (KW_PROCEDURE)
4     OPTIONS                                   Ключевое слово (KW_OPTIONS)
9     DECLARE                                   Ключевое слово (KW_DECLARE)
15    VAR                                       Ключевое слово (KW_VAR)
16    STATIC                                    Ключевое слово (KW_STATIC)
17    INITIAL                                   Ключевое слово (KW_INITIAL)
50    PUT                                       Ключевое слово (KW_PUT)
51    SKIP                                      Ключевое слово (KW_SKIP)
52    LIST                                      Ключевое слово (KW_LIST)
83    IF                                        Ключевое слово (KW_IF)
87    THEN                                      Ключевое слово (KW_THEN)
92    GOTO                                      Ключевое слово (KW_GOTO)
104   CALL                                      Ключевое слово (KW_CALL)
118   END                                       Ключевое слово (KW_END)
134   DO                                        Ключевое слово (KW_DO)
135   WHILE                                     Ключевое слово (KW_WHILE)
149   MOD                                       Ключевое слово (KW_MOD)
158   CONTINUE                                  Ключевое слово (KW_CONTINUE)
179   TO                                        Ключевое слово (KW_TO)
```

Рисунок 3.5 – Таблица ключевых слов

5 Таблица операторов и разделителей**.** На рисунке 3.6 приведена таблица, содержащая символы операторов и разделителей, что позволяет оценить корректность распознавания структурных элементов исходного кода.

```
=== Операторы и разделители ===
ID    Значение                                  Тип данных
-------------------------------------------------------------------------
2     :                                         Оператор (OP_COLON)
5     (                                         Оператор (OP_LPAREN)
7     )                                         Оператор (OP_RPAREN)
8     ;                                         Оператор (OP_SEMICOLON)
13    *                                         Оператор (OP_MULT)
28    ,                                         Оператор (OP_COMMA)
47    =                                         Оператор (OP_ASSIGN)
71    +                                         Оператор (OP_PLUS)
138   <                                         Оператор (OP_LT)
259   .                                         Оператор (OP_DOT)
```

Рисунок 3.6 – Таблица операторов и разделителей

6 В завершение работы лексический анализатор фиксирует четыре основные категории лексических ошибок, каждая из которых проиллюстрирована на рисунках 3.7–3.10. На рисунке 3.7 показана ситуация, когда после ключевого слова *DECLARE* сразу следует лексема, начинающаяся с цифры (например, *123S1*), что противоречит правилу о том, что имя переменной должно начинаться с буквы. В таком случае анализатор выдаёт сообщение об ошибке и прекращает дальнейшую обработку этого участка кода.

```
=== Лексические ошибки ===
Лексическая ошибка (token_id 10): После DECLARE встретилась слитная лексема '123S1' (уровень и идентификатор без пробела).
```

Рисунок 3.7 – Ошибка при объявлении переменной (начало имени с цифры)

На рисунке 3.8 приведён пример ошибки, возникающей при незакрытых открывающих скобках. Если количество открытых скобок в конце анализа превышает количество закрытых, лексер сообщает о том, что остались незакрытые скобки.

```
=== Лексические ошибки ===
Лексическая ошибка: Незакрытых открывающих скобок: 1
```

Рисунок 3.8 – Ошибка незакрытых открывающих скобок

Рисунок 3.9 демонстрирует ошибку незакрытой строковой константы. Она возникает в случае, когда строка начинается одинарной или двойной кавычкой, но не содержит соответствующей закрывающей кавычки до конца строки или до встречи другого разделителя. Анализатор распознаёт такое несоответствие и формирует сообщение вида «Незакрытая строковая константа, начинается с ' ...».

```
=== Лексические ошибки ===
Лексическая ошибка (token_id 19): Незакрытая строковая константа, начинается с '
```

Рисунок 3.9 – Ошибка незакрытой строковой константы

Наконец, на рисунке 3.10 показана ошибка, когда в тексте встречается символ, не принадлежащий ни к допустимым идентификаторам, ни к ключевым словам, ни к операторам или разделителям (например, символ @). В этом случае лексический анализатор отмечает встреченный символ как неопознанный и формирует соответствующее сообщение об ошибке.

```
=== Лексические ошибки ===
Лексическая ошибка (token_id 11): Неопознанный символ: '@'
```

Рисунок 3.10 – Ошибка неопознанного символа

В завершение можно отметить, что выявление и корректное отображение лексических ошибок – важный этап при написании компиляторов и интерпретаторов. Адекватные и наглядные сообщения об ошибках значительно упрощают процесс отладки исходного кода и делают разработку на целевом языке более удобной для программиста.

ЗАКЛЮЧЕНИЕ

В ходе выполненной лабораторной работы была решена задача разработки лексического анализатора для упрощённого подмножества языка *PL/1*. На этапе лексического анализа исходный код превращается в поток лексем, каждая из которых относится к одному из классов: ключевое слово, идентификатор, константа, оператор или разделитель. Особое внимание уделялось обработке лексических ошибок, таких как незакрытые кавычки, незакрытые скобки, недопустимые символы и неправильное объявление переменных (начинающихся с цифры). Анализатор регистрирует обнаруженные несоответствия и выводит информативные сообщения, упрощая процесс отладки и повышая качество программного кода.

Реализованная программа на языке *R* демонстрирует простоту и гибкость подхода к лексическому разбору с использованием регулярных выражений. Структура проекта включает отдельные таблицы для идентификаторов, констант, типов данных и ключевых слов, что позволяет эффективно переиспользовать информацию об уже встречавшихся элементах. Кроме того, модульное построение кода упрощает расширение анализатора новыми конструкциями языка и интеграцию с последующими этапами трансляции, такими как синтаксический и семантический анализ.

Таким образом, проведённое исследование подтвердило важность корректного лексического анализа на ранних этапах трансляции. Предложенный анализатор даёт наглядное представление о структурировании кода и механизмах детектирования типовых лексических ошибок. Полученные результаты могут служить основой для дальнейшего развития инструмента: дополнения поддержки синтаксических конструкций, реализации полноценного парсера и последующего анализа или оптимизации программ, написанных на *PL/1*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] PL/1 – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/ПЛ/1. – Дата доступа: 02.02.2025.

[2] PL1-KT Documentation [Электронный ресурс]. – Режим доступа: https://pl1.su/compiler-pl-1-kt/documentation-load/. – Дата доступа: 02.02.2025.

[3] IBM Enterprise PL/1 for z/OS Language Reference [Электронный ресурс]. – Режим доступа: https://www.ibm.com/support/pages/enterprise-pli-zos-documentation-library. – Дата доступа: 03.02.2025.

[4] Micro Focus Open PL/1 Language Reference Manual [Электронный ресурс]. – Режим доступа: https://www.microfocus.com/documentation/enterprise-developer/ed60/ED-VS2017/BKPFPFPREF.html. – Дата доступа: 03.02.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
keywords <- c(
  "PROCEDURE", "OPTIONS", "DECLARE", "VAR", "STATIC", "INITIAL", "FILE",
  "OPEN", "UPDATE", "RECORD", "TITLE", "PUT", "SKIP", "LIST", "ADDR",
  "IF", "THEN", "GOTO", "CALL", "END", "BASED",
  "DO", "WHILE", "MOD", "CONTINUE", "REPEAT", "ELSE", "LEAVE", "TO",
  "BY", "LIKE"
)

operators_delimiters <- c(
  "(", ")", ";", ":", ",", "=", "+", "-", "*", "/",
  "{", "}", "[", "]", "<", ">", "."
)

data_types_list <- c(
  "CHAR", "FIXED", "DECIMAL", "BIT", "BINARY", "FLOAT", "COMPLEX",
  "LABEL", "ENTRY", "VARIABLE", "POINTER", "FILE", "CHARACTER"
)

identifiers_table <- list()
constants_table <- list()
lex_errors <- list()
token_id_counter <<- 1

get_data_type <- function(tok) {
  if (tok$type == "numeric_constant") {
    if (grepl("[\\.eE]", tok$value)) {
      return("Числовая константа с плавающей точкой")
    } else {
      return("Целочисленная константа")
    }
  } else if (tok$type == "string_constant") {
    if (nchar(tok$value) == 1) {
      return("Символьный литерал")
    } else {
      return("Строковый литерал")
    }
  } else if (tok$type == "bit_constant") {
    return("Битовая константа")
  } else if (tok$type == "identifier") {
    return("Идентификатор")
  } else if (tok$type == "keyword") {
    return(paste("Ключевое слово (KW_", toupper(tok$value), ")", sep = ""))
  } else if (tok$type == "operator") {
    op_map <- list(
      "="  = "OP_ASSIGN",
      "+"  = "OP_PLUS",
      "-"  = "OP_MINUS",
      "*"  = "OP_MULT",
      "/"  = "OP_DIV",
      "("  = "OP_LPAREN",
      ")"  = "OP_RPAREN",
      ":"  = "OP_COLON",
      ";"  = "OP_SEMICOLON",
      ","  = "OP_COMMA",
      "<=" = "OP_LE",
      ">=" = "OP_GE",
      "<>" = "OP_NE",
      "!=" = "OP_NE",
```

```
      "<"  = "OP_LT",
      ">"  = "OP_GT",
      "."  = "OP_DOT"
    )
    if (tok$value %in% names(op_map)) {
      return(paste("Оператор (", op_map[[tok$value]], ")", sep = ""))
    } else {
      return("Оператор/разделитель")
    }
  } else {
    return("Неизвестный тип")
  }
}

tokenize <- function(input) {
  tokens <- list()
  parenthesis_counter <- 0

  remove_leading_spaces <- function(s) {
    sub("^[[:space:]]+", "", s)
  }

  multi_operators <- c("<=", ">=", "<>", "!=")

  while (nchar(input) > 0) {
    input <- remove_leading_spaces(input)
    if (nchar(input) == 0) break

    token <- NULL

    m <- regexpr("^[0-9]+(\\.[0-9]+)?([eE][+-]?[0-9]+)?", input, perl = TRUE)
    if (m[1] != -1 && attr(m, "match.length") > 0) {
      val <- substring(input, 1, attr(m, "match.length"))
      token <- list(
        tokenID = token_id_counter,
        type = "numeric_constant",
        value = val
      )
      if (!(val %in% names(constants_table))) {
        constants_table[[val]] <<- token_id_counter
      }
      token$pointer <- constants_table[[val]]
      token_id_counter <<- token_id_counter + 1
      tokens[[length(tokens) + 1]] <- token
      input <- substring(input, attr(m, "match.length") + 1)
      next
    }

    if (substr(input, 1, 1) %in% c("'", "\"")) {
      quote_char <- substr(input, 1, 1)
      remaining <- substring(input, 2)
      pos_quote <- regexpr(quote_char, remaining, fixed = TRUE)
      pos_newline <- regexpr("\n", remaining, fixed = TRUE)
      if (pos_quote == -1 || (pos_newline != -1 && pos_newline < pos_quote)) {
        err_msg <- sprintf(
          "Лексическая ошибка (token_id %d): Незакрытая
      строковая константа, начинается с %s",
          token_id_counter, quote_char
        )
        lex_errors[[length(lex_errors) + 1]] <<- err_msg
        return(tokens)
      } else {
        match_length <- pos_quote + 1
        raw_val <- substring(input, 1, match_length)
```

```
    m <- regexpr("^('([^']*)'|\"([^\"]*)\")", raw_val, perl = TRUE)
    if (m[1] == -1 || attr(m, "match.length") == 0) {
      err_msg <- sprintf("Лексическая ошибка (token_id %d): Ошибка в
      определении строковой константы.", token_id_counter)
      lex_errors[[length(lex_errors) + 1]] <<- err_msg
      return(tokens)
    }
    val <- sub("^['\"](.*)['\"]$", "\\1", raw_val)
    token <- list(
      tokenID = token_id_counter,
      type = "string_constant",
      value = val
    )
    if (!(val %in% names(constants_table))) {
      constants_table[[val]] <<- token_id_counter
    }
    token$pointer <- constants_table[[val]]
    token_id_counter <<- token_id_counter + 1
    tokens[[length(tokens) + 1]] <- token
    input <- substring(input, match_length + 1)
    next
  }
}

m <- regexpr("^[A-Za-z][A-Za-z0-9_]*", input, perl = TRUE)
if (m[1] != -1 && attr(m, "match.length") > 0) {
  val <- substring(input, 1, attr(m, "match.length"))
  if (val %in% data_types_list) {
    token <- list(
      tokenID = token_id_counter,
      type = "identifier",
      value = val
    )
    if (!(val %in% names(identifiers_table))) {
      identifiers_table[[val]] <<- token_id_counter
    }
    token$pointer <- identifiers_table[[val]]
  } else if (val %in% keywords) {
    token <- list(
      tokenID = token_id_counter,
      type = "keyword",
      value = val
    )
  } else {
    token <- list(
      tokenID = token_id_counter,
      type = "identifier",
      value = val
    )
    if (!(val %in% names(identifiers_table))) {
      identifiers_table[[val]] <<- token_id_counter
    }
    token$pointer <- identifiers_table[[val]]
  }
  token_id_counter <<- token_id_counter + 1
  tokens[[length(tokens) + 1]] <- token
  input <- substring(input, attr(m, "match.length") + 1)
  next
}

if (nchar(input) >= 2) {
  potential_op <- substr(input, 1, 2)
  if (potential_op %in% multi_operators) {
    token <- list(
```

```
        tokenID = token_id_counter,
        type = "operator",
        value = potential_op
      )
      token$pointer <- token_id_counter
      token_id_counter <<- token_id_counter + 1
      tokens[[length(tokens) + 1]] <- token
      if (potential_op == "(") {
        parenthesis_counter <- parenthesis_counter + 1
      } else if (potential_op == ")") {
        if (parenthesis_counter == 0) {
          err_msg <- sprintf("Лексическая ошибка (token_id %d): Избыточная
          закрывающая скобка.", token_id_counter)
          lex_errors[[length(lex_errors) + 1]] <<- err_msg
          return(tokens)
        } else {
          parenthesis_counter <- parenthesis_counter - 1
        }
      }
      input <- substring(input, 3)
      next
    }
  }

  op <- substr(input, 1, 1)
  if (op %in% operators_delimiters) {
    token <- list(
      tokenID = token_id_counter,
      type = "operator",
      value = op
    )
    token$pointer <- token_id_counter
    token_id_counter <<- token_id_counter + 1
    tokens[[length(tokens) + 1]] <- token
    if (op == "(") {
      parenthesis_counter <- parenthesis_counter + 1
    } else if (op == ")") {
      if (parenthesis_counter == 0) {
        err_msg <- sprintf("Лексическая ошибка (token_id %d): Избыточная
        закрывающая скобка.", token_id_counter)
        lex_errors[[length(lex_errors) + 1]] <<- err_msg
        return(tokens)
      } else {
        parenthesis_counter <- parenthesis_counter - 1
      }
    }
    input <- substring(input, 2)
    next
  }

  err_char <- substr(input, 1, 1)
  err_msg <- sprintf("Лексическая ошибка (token_id %d):
   Неопознанный символ: '%s'", token_id_counter, err_char)
  lex_errors[[length(lex_errors) + 1]] <<- err_msg
  return(tokens)
}

if (parenthesis_counter > 0) {
  err_msg <- sprintf("Лексическая ошибка: Незакрытых открывающих
  скобок: %d", parenthesis_counter)
  lex_errors[[length(lex_errors) + 1]] <<- err_msg
  return(tokens)
}
```

```
  return(tokens)
}

group_tokens <- function(tokens) {
  tokens <- lapply(tokens, function(tok) {
    if (tok$type == "identifier" && (tok$value %in% keywords) &&
      !(tok$value %in% data_types_list)) {
      tok$type <- "keyword"
    }
    return(tok)
  })

  groups <- list(
    identifiers = Filter(function(tok) tok$type == "identifier", tokens),
    constants = Filter(function(tok) {
      tok$type %in% c("numeric_constant", "string_constant")
    }, tokens),
    keywords = Filter(function(tok) tok$type == "keyword", tokens),
    operators = Filter(function(tok) tok$type == "operator", tokens)
  )

  groups$types <- Filter(
    function(tok) tok$value %in% data_types_list,
    groups$identifiers
  )
  groups$variables <- Filter(function(tok) {
    !(tok$value %in% data_types_list)
  }, groups$identifiers)

  return(groups)
}

unique_tokens <- function(token_group) {
  keys <- sapply(token_group, function(tok) {
    paste(tok$type, tok$value, sep = "|")
  })
  unique_indices <- which(!duplicated(keys))
  return(token_group[unique_indices])
}

print_token_group <- function(group_name, token_group, con) {
  unique_group <- unique_tokens(token_group)

  id_col_width <- 5
  value_col_width <- 40
  type_col_width <- 40

  writeLines(sprintf("=== %s ===", group_name), con)
  header <- paste(
    pad_string("ID", id_col_width),
    pad_string("Значение", value_col_width),
    pad_string("Тип данных", type_col_width)
  )
  writeLines(header, con)
  writeLines(strrep("-", id_col_width + value_col_width +
    type_col_width + 2), con)

  for (tok in unique_group) {
    value <- tok$value
    dtype <- get_data_type(tok)

    line <- paste(
      pad_string(as.character(tok$tokenID), id_col_width),
      pad_string(value, value_col_width),
```

```
      pad_string(dtype, type_col_width)
    )

    writeLines(line, con)
  }
  writeLines("", con)
}

pad_string <- function(text, width) {
  w <- nchar(text, type = "width")
  if (w >= width) {
    return(paste0(substr(text, 1, width - 3), "..."))
  } else {
    spaces <- strrep(" ", width - w)
    return(paste0(text, spaces))
  }
}

output_results <- function(tokens,
                           output_filename = "LEXICAL_OUTPUT.txt") {
  con <- file(output_filename, open = "w", encoding = "UTF-8")
  groups <- group_tokens(tokens)

  print_token_group("Типы данных", groups$types, con)
  print_token_group("Переменные", groups$variables, con)
  print_token_group("Константы", groups$constants, con)
  print_token_group("Ключевые слова", groups$keywords, con)
  print_token_group("Операторы и разделители", groups$operators, con)

  writeLines("=== Лексические ошибки ===", con)
  if (length(lex_errors) > 0) {
    for (err in lex_errors) {
      writeLines(err, con)
    }
  } else {
    writeLines("Ошибок не обнаружено.", con)
  }
  close(con)
}

if (file.exists("INPUT.TXT")) {
  input_text <- paste(
    readLines("INPUT.TXT", warn = FALSE),
    collapse = "\n"
  )
}

tokens <- tokenize(input_text)
output_results(tokens, "output.txt")
```