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 Т. П. Власенко

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Целью выполнения данной лабораторной работы является 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.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Данный код представляет собой лексический анализатор для языка программирования *Golang*. Лексический анализатор – это программа или часть программы, выполняющая лексический анализ [1]. Она разбивает исходный код на токены и находит лексические ошибки.

Программа начинается с определения набора ключевых слов, операторов и типов литералов, которые используются в языке *Golang*. Далее описаны функции, участвующие в разбиении кода на токены.

Функция *tokenize* управляет главным циклом анализа кода. Она запускает цикл, который выполняется до тех пор, пока не будет обработан весь исходный код.

Для обработки следующего токена используется функция scan_*token*. В зависимости от текущего символа она определяет какого типа токен необходимо сейчас обработать: литерал, оператор, идентификатор, комментарий, нелегальный символ.

Для получения разных типов токенов были разработаны соответствующие функции: *scan_number, scan_identifier, scan_string_literal, scan_char_literal* и другие.

Также в разработанный лексический анализатор добавлена функция, которая при считывании идентификатора проверит, не хотел ли разработчик использовать подходящее ключевое слово вместо написанного идентификатора. Она проверяет, можно ли из идентификатора получить ключевое слово, совершив не более N изменений (удаление или замена символа). Таким образом реализован механизм подсказок.

Остальная часть кода содержит вспомогательные функции, которые необходимы для проведения анализа.

В результате работы программы на выходе лексический анализатор предоставляет массив токенов, каждый из которых содержит информацию о типе, лексеме и позиции токена в коде. Также на выходе получается массив ошибок, которые содержат в себе позицию ошибки и сообщение, и массив предупреждений, которые содержат в себе позицию возможной ошибки и сообщение. Эти массивы можно использовать для дальнейшего анализа и обработки программного кода на языке *Golang*.

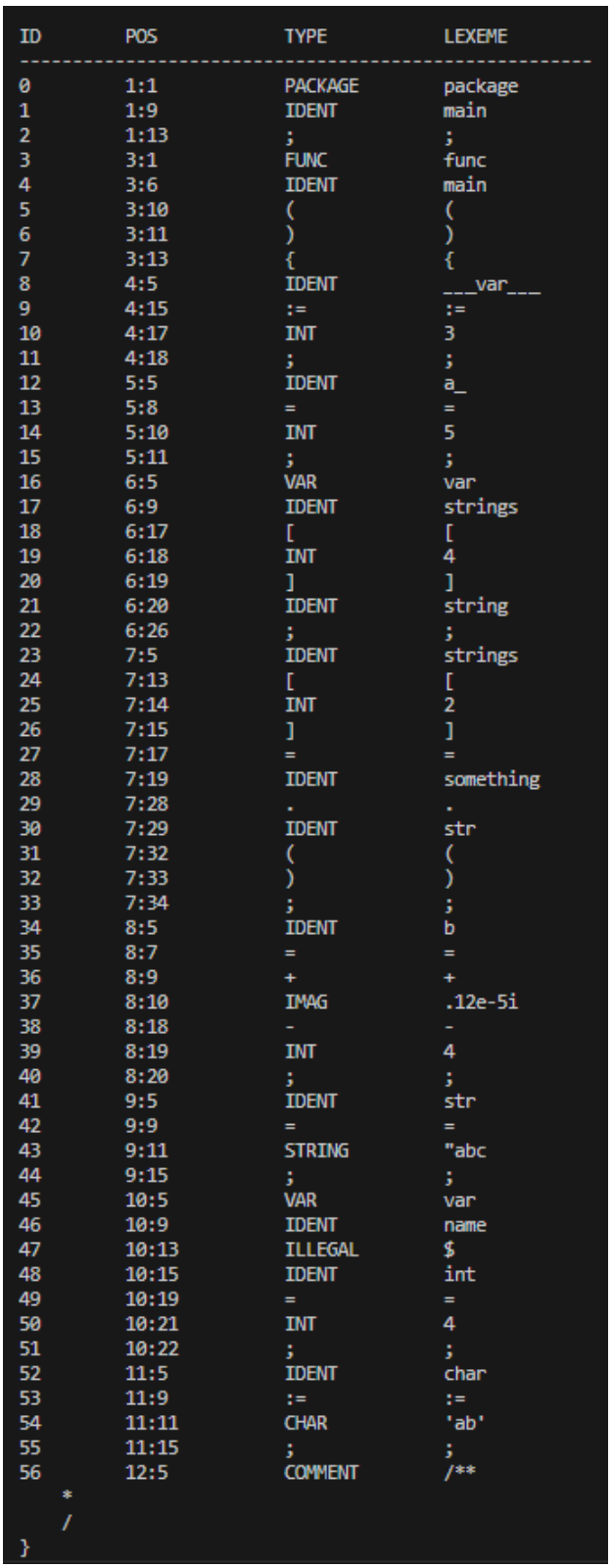
3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы был разработан лексический анализатор подмножества языка *Golang*. На рисунке 1 представлен тестовый код, который использовался для анализа разработанной программой:

```
1   package main
2
3   func main() {
4       ___var___ :=3
5       a_ = 5
6       var strings [4]string
7       strings [2] = something.str()
8       b = +.12e-5i -4
9       str = "abc
10      var name$ int = 4
11      char:='ab'
12      /**
13      *
14      /
15  }
```

Рисунок 1 – Тестовый код программы

На рисунке 2 можно увидеть результат работы анализатора, который разбивает исходный код на токены, находит все лексические ошибки, находит потенциальные ошибки и предупреждает о них, затем выводит полученный массив токенов в виде таблицы.



```
ID      POS         TYPE        LEXEME
-------------------------------------------------
0       1:1         PACKAGE     package
1       1:9         IDENT       main
2       1:13        ;           ;
3       3:1         FUNC        func
4       3:6         IDENT       main
5       3:10        (           (
6       3:11        )           )
7       3:13        {           {
8       4:5         IDENT       ___var___
9       4:15        :=          :=
10      4:17        INT         3
11      4:18        ;           ;
12      5:5         IDENT       a_
13      5:8         =           =
14      5:10        INT         5
15      5:11        ;           ;
16      6:5         VAR         var
17      6:9         IDENT       strings
18      6:17        [           [
19      6:18        INT         4
20      6:19        ]           ]
21      6:20        IDENT       string
22      6:26        ;           ;
23      7:5         IDENT       strings
24      7:13        [           [
25      7:14        INT         2
26      7:15        ]           ]
27      7:17        =           =
28      7:19        IDENT       something
29      7:28        .           .
30      7:29        IDENT       str
31      7:32        (           (
32      7:33        )           )
33      7:34        ;           ;
34      8:5         IDENT       b
35      8:7         =           =
36      8:9         +           +
37      8:10        IMAG        .12e-5i
38      8:18        -           -
39      8:19        INT         4
40      8:20        ;           ;
41      9:5         IDENT       str
42      9:9         =           =
43      9:11        STRING      "abc
44      9:15        ;           ;
45      10:5        VAR         var
46      10:9        IDENT       name
47      10:13       ILLEGAL     $
48      10:15       IDENT       int
49      10:19       =           =
50      10:21       INT         4
51      10:22       ;           ;
52      11:5        IDENT       char
53      11:9        :=          :=
54      11:11       CHAR        'ab'
55      11:15       ;           ;
56      12:5        COMMENT     /**
    *
    /
}
```

Рисунок 2 – Результат разбиения тестового кода на токены

На рисунке 3 представлены ошибки, который выявил лексический анализатор в процессе анализа кода.

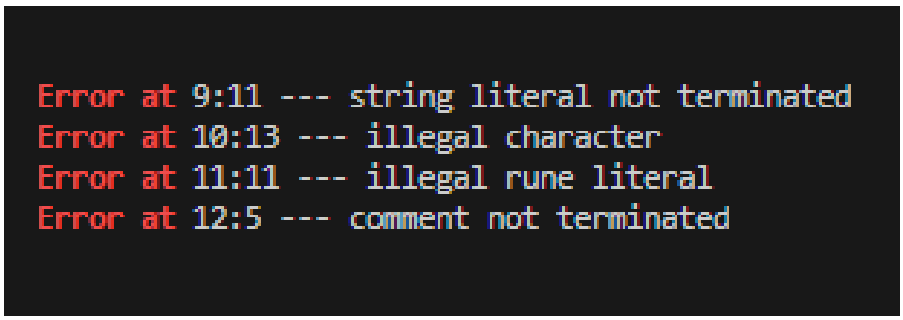


Рисунок 3 – Лексические ошибки, найденные во время анализа тестового кода

Первая ошибка означает, что в коде был обнаружен строковый литерал, который не имеет закрывающей кавычки. Вторая ошибка сигнализирует о том, что лексический анализатор обнаружил нелегальный символ. Третья ошибка сообщает, что символьный литерал в тестовом коде является некорректным. Четвертая ошибка указывает на незакрытый комментарий.

На рисунке 4 результат работы программы представлен в ином формате: идентификаторы в коде программа заменила на *ID* соответствующего им токена.

```
1   package <1> ;
2   func <4> ( ) {
3       <8> := 3 ;
4       <12> = 5 ;
5       var <17> [ 4 ] <21> ;
6       <23> [ 2 ] = <28> . <30> ( ) ;
7       <34> = + .12e-5i - 4 ;
8       <41> = "abc ;
9       var <46> $ <48> = 4 ;
10      <52> := 'ab' ;
11      /**
12      *
13      /
14  }
15
16
```

Рисунок 4 – Тестовый код, в котором идентификаторы представлены в виде своего *ID*

В данной главе была рассмотрена работа разработанного лексического анализатора языка программирования *Golang*.

ВЫВОДЫ

В ходе выполнения данной лабораторной работы был разработан лексический анализатор подмножества языка программирования *Golang*, который в результате своей работы получает массив токенов, ошибок и предупреждений. Был продемонстрирован результат работы программы при обнаружении некорректных лексем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Лексический анализ [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Лексический_анализ#Лексический_анализатор – Дата доступа: 22.02.2024

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

```
void lexer::scan_token() {
    char c = next();
    switch (c) {
        // 1
        case '(':
            add_token(token_type::LPAREN);
            break;
        case ')':
            add_token(token_type::RPAREN);
            break;
        case '{':
            add_token(token_type::LBRACE);
            break;
        case '}':
            add_token(token_type::RBRACE);
            break;
        case '[':
            add_token(token_type::LBRACK);
            break;
        case ']':
            add_token(token_type::RBRACK);
            break;
        case ',':
            add_token(token_type::COMMA);
            break;
        case ';':
            add_token(token_type::SEMICOLON);
            break;
        // 2
        case '.':  // .[0-9]
            if (isdigit(source_[cur_])) {
                scan_number();
            } else {
                add_token(token_type::PERIOD);
            }
            break;
        case ':':  // : :=
            add_token(match('=') ? token_type::DEFINE : token_type::COLON);
            break;
        case '-':  // - -= --
            if (match('-')) {
                add_token(token_type::DEC);
            } else if (match('=')) {
                add_token(token_type::SUB_ASSIGN);
            } else {
                add_token(token_type::SUB);
            }
            break;
        case '+':  // + += ++
            if (match('+')) {
                add_token(token_type::INC);
            } else if (match('=')) {
                add_token(token_type::ADD_ASSIGN);
            } else {
                add_token(token_type::ADD);
            }
```

```
            break;
        case '*':  // * *=
            add_token(match('=') ? token_type::MUL_ASSIGN : token_type::MUL);
            break;
        case '/':  // / /= '//' '/*'
            if (match('=')) {
                add_token(token_type::QUO_ASSIGN);
            } else if (match('/')) {
                while (!eof() && get_cur_char() != '\n') next();
                add_token(token_type::COMMENT);
            } else if (match('*')) {
                size_t prev_line = line_;
                while (!eof() &&
                       (get_cur_char() != '*' ||
                        (get_cur_char() == '*' && get_next_char() != '/'))) {
                    if (get_cur_char() == '\n') line_++;
                    next();
                };
                if (eof()) {
                    add_error({file_path_, prev_line, get_lexem_col()},
                              "comment not terminated");
                }
                if (!eof()) next();
                if (!eof()) next();
                auto lexeme = source_.substr(start_, cur_ - start_);
                tokens_.push_back({token_type::COMMENT,
                                   lexeme,
                                   {file_path_, prev_line,
get_lexem_col()}});
            } else {
                add_token(token_type::QUO);
            }
            break;
        case '%':  // % %=
            add_token(match('=') ? token_type::REM_ASSIGN : token_type::REM);
            break;
        case '=':  // = ==
            add_token(match('=') ? token_type::EQL : token_type::ASSIGN);
            break;
        case '^':  // ^ ^=
            add_token(match('=') ? token_type::XOR_ASSIGN : token_type::XOR);
            break;
        case '<':  // < << <<=
            if (match('<', '=')) {
                add_token(token_type::SHL_ASSIGN);
            } else if (match('<')) {
                add_token(token_type::SHL);
            } else {
                add_token(token_type::LSS);
            }
            break;
        case '>':  // > >> >>=
            if (match('>', '=')) {
                add_token(token_type::SHR_ASSIGN);
            } else if (match('>')) {
                add_token(token_type::SHR);
            } else {
                add_token(token_type::GTR);
            }
            break;
        case '&':  // & &= && &^ &^=
            if (match('^', '=')) {
                add_token(token_type::AND_NOT_ASSIGN);
```

```
            } else if (match('^')) {
                add_token(token_type::AND_NOT);
            } else if (match('&')) {
                add_token(token_type::LAND);
            } else if (match('=')) {
                add_token(token_type::AND_ASSIGN);
            } else {
                add_token(token_type::AND);
            }
            break;
        case '|':  // | |= ||
            if (match('|')) {
                add_token(token_type::LOR);
            } else if (match('=')) {
                add_token(token_type::OR_ASSIGN);
            } else {
                add_token(token_type::OR);
            }
            break;
        case '!':  // ! !=
            add_token(match('=') ? token_type::NEQ : token_type::NOT);
            break;
        case ' ':
        case '\r':
        case '\t':
            // Ignore whitespace chars.
            break;
        case '\n':
            insert_semicolon();
            line_++;
            break;
        case '"':
            scan_string_literal();
            break;
        case '`':
            scan_multiline_string_literal();
            break;
        case '\'':
            scan_char_literal();
            break;
        default:
            if (std::isdigit(c)) {
                scan_number();
            } else if (is_alpha(c)) {
                scan_identifier();
            } else {
                add_error({file_path_, line_, get_lexem_col()},
                          "illegal character");
                add_token(token_type::ILLEGAL);
            }
    }
}
```