1.1. ПОСТАНОВКА ЗАДАЧИ

1.1. Цель

Целью данной работы является:

- освоение основных методов разработки распознавателей формальных языков на примере модельного языка программирования;
- приобретение практических навыков написания транслятора языка программирования; закрепление практических навыков самостоятельного решения инженерных задач, умения пользоваться справочной литературой и технической документацией.

1.2. Требования

Разработать распознаватель модельного языка программирования согласно заданной формальной грамматике.

Распознаватель это специальных алгоритм, позволяющий вынести решение и принадлежности символов некоторому языку.

Распознаватель можно схематично представить в виде совокупности входной ленты, читающей головки, которая указывает на очередной символ на ленте, устройства управления (УУ) и дополнительной памяти (стек).

Конфигурацией распознавателя является:

- состояние УУ; - содержимое входной ленты;
- положение читающей головки;
- содержимое дополнительной памяти (стека).

Трансляция исходного текста программы происходит в несколько этапов. Основными этапами являются следующие:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация целевого кода.

Лексический анализ является наиболее простым этапом и выполняется с помощью регулярной грамматики. Регулярным грамматикам соответствуют конечные автоматы, следовательно, разработка и написание программы лексического анализатора эквивалентна разработке конечного автомата и его диаграммы состояний (ДС).

Синтаксический анализатор строится на базе контекстно-свободных (КС) грамматик. Задача синтаксического анализатора – провести разбор программы и сопоставить его с формальным описание языка.

Семантический анализ позволяет учесть особенности языка программирования, которые не могут быть описаны правилами КС-грамматики. К таким особенностям относятся:

- обработка описаний;
- анализ выражений;
- проверка правильности операторов.

Обработки описаний помогают убедиться, что каждая переменная в программе описана и при этом только один раз.

Анализ выражений заключается в том, чтобы проверить описаны ли переменные, участвующие в выражении, и соответствуют ли типы операндов друг другу и типу операции.

Этапы синтаксического и семантического анализа обычно можно объединить.

1.3. Порядок выполнения

1. В соответствии с номером варианта составить описание модельного языка программирования в виде правил вывода формальной грамматики;

2. Составить таблицу лексем и нарисовать диаграмму состояний для распознавания и формирования лексем языка;

3. Разработать процедуру лексического анализа исходного текста программы на языке высокого уровня;

4. Разработать процедуру синтаксического анализа исходного текста методом рекурсивного спуска на языке высокого уровня;

5. Построить программный продукт, читающий текст программы, написанной на модельном языке, в виде консольного приложения;

6. Протестировать работу программного продукта с помощи серии тестов, демонстрирующих все основные особенности модельного языка программирования, включая возможные лексические и синтаксические ошибки.

2. ГРАММАТИКА ЯЗЫКА

Согласно индивидуальному варианту задания на курсовую работу грамматика языка включает следующие синтаксические конструкции:

1. <операции_группы_отношения>::= NE | EQ | LT | LE | GT | GE
2. <операции_группы_сложения>::= plus | min | or
3. <операции_группы_умножения>::= mult | div | and
4. <унарная_операция>::= ~
5. <программа> = {/ (<описание> | <оператор>)(:|переход строки) /} end
6. <описание>::= {<идентификатор> {, <идентификатор> } :<тип> ;}
7. <тип>::= % | ! | $
8. <оператор>::= <составной>|<присваивания> |<условный> |<цикла> |<ввода> |<вывода>
9. <составной>::= «{» <оператор> { ; <оператор> } «}»
10. <присваивания> ::= [let] <идентификатор> = <выражение>
11. <условный>::= if <выражение> then <оператор> [else <оператор>] end_else
12. <фиксированного_цикла>::= for «(»[<выражение>] ; [<выражение>] ; [<выражение>] «)»<оператор>
13. <условного_цикла>::= do while <выражение> <оператор> loop
14. <ввода>::= input «(»<идентификаторр> {пробел <идентификатор>} «)»
15. <вывода>::= output «(»<выражение> { пробел <выражение> } «)»
16. <выражение> ::= {<операции_группы_отношения> <операнд>}
17. <операнд> ::= <слагаемое> {<операции_группы_сложенияя> <слагаемое>}
18. <слагаемое> ::= <множитель> {<операции_группы_умножения> <множитель>}
19. <множитель> ::= <идентификатор> | <число> | <логическая_константа> | <унарная_операция> <множитель> | «(» <выражение> «)»
20. <логическая_константа> ::= true | false
21. <идентификатор> ::=< буква>>{<буква>| <цифра>}

22.<числоо> ::= <цифра>{<цифра>}

23.Признак начала комментария (* Признак конца комментария *)

24.<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

25.<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

26.<действительное> ::= <цифра>{<цифра>}.<цифра>{<цифра>}

Правила, определяющие константные выражения в программе:

<ключеные_слова>::=true | false | NE | EQ | LT | LE | GT | GE | plus | min | or | mult | div | and | % | ! | $ | if | else | end_else | for | do | while | loop | input | output

<разделители>::= { | } | , | ; | : | (|) | = | (* | *) | ~

Здесь для записи правил грамматики используется форма Бэкуса-Наура (БНФ). В записи БНФ левая и правая части порождения разделяются символом «::=», нетерминалы заключены в угловые скобки, а терминалы – просто символы, используемые в языке. Жирным выделены терминалы, представляющие собой ключевые слова языка.

3. ЭТАПЫ РАЗРАБОТКИ РАСПОЗНАВАТЕЛЯ

3.1. Разработка лексического анализатора

Лексический анализатор – подпрограмма, которая принимает на вход исходный текст программы и выдает последовательность лексем – минимальных элементов программы, несущих смысловую нагрузку.

В модельном языке программирования выделяют следующие типы лексем:

- ключевые слова;
- ограничители;
- числа;
- идентификаторы.

При разработке лексического анализатора, ключевые слова и ограничителя известны заранее, идентификаторы и числовые константы – вычисляются в момент разбора исходного текста.

Для каждого типа лексем предусмотрена отдельная таблица. Таким образом, внутреннее представление лексемы – пара чисел (n, k), где n – номер таблицы лексем, k – номер лексемы в таблице.

Кроме того, в исходном коде программы кроме ключевых слов, идентификаторов и числовых констант может находиться произвольное число пробельных символов («пробел», «табуляция», «перенос строки», «возврат каретки») и комментариев, заключенных в фигурные скобки.

Лексический анализ текста проводится по регулярной грамматике. Известно, что регулярная грамматика эквивалентна конченому автомату, следовательно, для написания лексического анализатора необходимо построить диаграмму состояний, соответствующего конечного автомата (рис. 1).

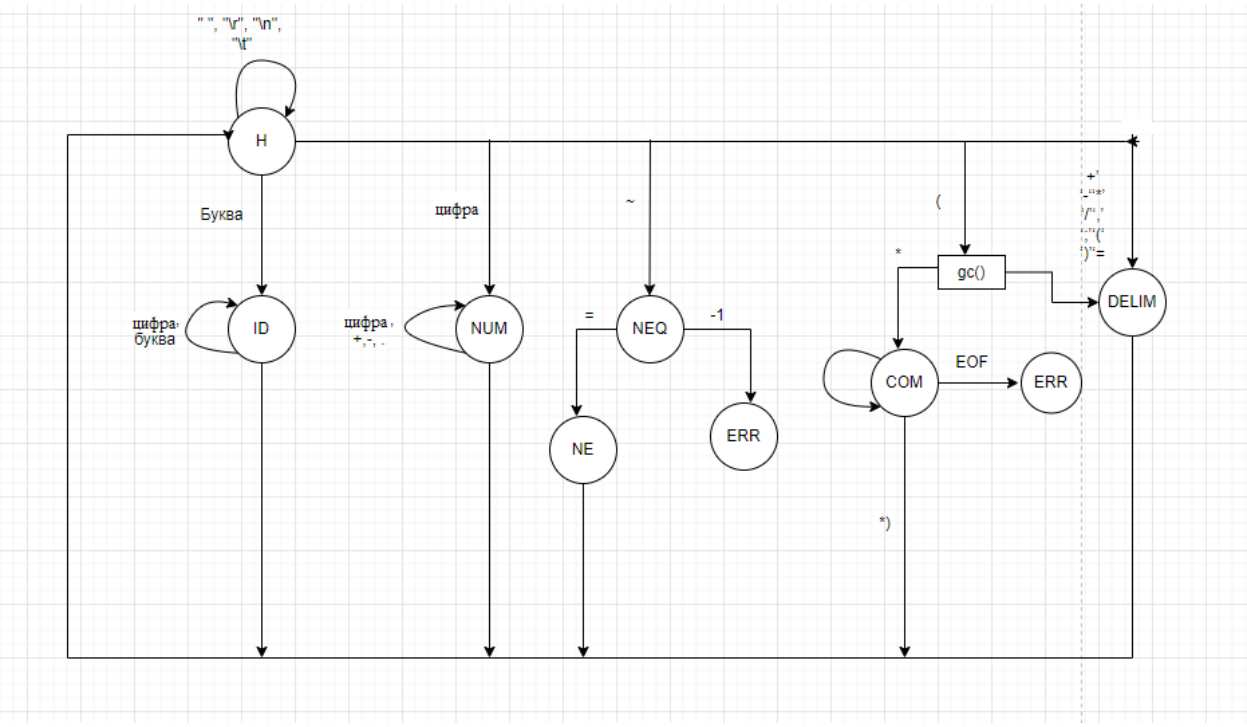

Рис. 1. Диаграмма состояний лексического анализатора

3.2. Разработка синтаксического анализатора

Будем считать, что лексический и синтаксической анализаторы взаимодействуют следующим образом. Если синтаксическому анализатору для анализа требуется очередная лексема, он запрашивает ее у лексического анализатора. Таким образом, разбор исходного текста программы идет под управлением подпрограммы синтаксического анализатора (Parser).

Разработку синтаксического анализатора проведем с помощью метода рекурсивного спуска (РС). В основе метода лежит тот факт, что каждому нетерминалу ставится в соответствие рекурсивная функция. Для того, чтобы в явном виде представить множество рекурсивных функций, перепишем грамматические правила следующим образом:

Выражения языка задаются правилами:

COMPARE -> ADD | COMPARE NE ADD | COMPARE EQ ADD |

COMPARE LT ADD | COMPARE LE ADD |

COMPARE GT ADD | COMPARE GE ADD

ADD -> MULT | ADD plus MULT | ADD min MULT | ADD or MULT

MULT -> FACT | MULT div FACT | MULT mult FACT | MULT and FACT

FACT -> ID | NUM | LOG | ~FACT | (COMPARE)

NUM -> INT | REAL

LOG -> true | false

Правила, определяющие структуру программы:

RP -> BODY end

BODY -> DESCR: | OPER: | BODY OPER: | BODY DESCR:

Правила, определяющие раздел описания и типы переменных:

DESCR -> ID1 : % | ID1 : ! | ID1 : $

ID1 -> ID | ID1, ID

OPER -> { OPER1 } | ID1 = COMPARE | if COMPARE then OPER end_else |

| if COMPARE then OPER else OPER end_else | do while COMPARE OPER loop

| for (COMPARE_FOR) OPER | input(ID1) | output(EXPR)

EXPR -> COMPARE | EXPR, COMPARE

COMPARE_FOR -> ;; | ; COMPARE ; | COMPARE ;; | ;; COMPARE | ; COMPARE ; COMPARE | COMPARE ; ; COMPARE | COMPARE; COMPARE; | COMPARE ; COMPARE ; COMAPRE ;

Правило определяющее составной оператор программы:

OPER1 -> OPER | OPER1 ; OPER

$P \rightarrow$ S|D1 (: | \n) {S|D1}

$D1 \rightarrow I$ {,I} : (**%|!|$**);

$B \rightarrow S$ {S :} **end**

$S \rightarrow I \coloneqq E$| ***if*** E ***then*** S [***else*** S] **end_else** | **do *while*** E S **loop**| **for (**[E];[E];[E] S**)** |B | D1 | **input**(I) | **output**(E)

$E \rightarrow E1${[**NE** | **EQ** | **LT** | **LE** | **GT** | **GE**]$E1$}

$E1 \rightarrow T${[**plus** | **min** | ***or***] T}

$T \rightarrow F${[**mult** | **div** | ***and***] F}

$F \rightarrow I \mid N \mid L \mid \sim F \mid (E)$

$L \rightarrow$ ***true*** | ***false***

$I \rightarrow C \mid IC \mid IR$

$N \rightarrow R \mid NR$

$C \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z$

$R \rightarrow 0 \mid 1 \mid \ldots \mid 9$

Здесь правила для нетерминалов L, I, N, C и R описаны на этапе лексического разбора. Следовательно, остается описать функции для нетерминалов P, D1, D, B, S, E, E1, T, F. Исходный код синтаксического анализатора приведен в Приложении 2

3.3. Семантический анализ

Некоторые особенности модельного языка не могут быть описаны контекстно-свободной грамматикой. К таким правилам относятся:

- любой идентификатор, используемый в теле программы должен быть описан;
- повторное описание одного и того же идентификатора не разрешается;
- в операторе присваивания типы идентификаторов должны совпадать;
- в условном операторе и операторе цикла в качестве условия допустимы только логические выражения;
- операнды операций отношения должны быть целочисленными.

Указанные особенности языка разбираются на этапе семантического анализа. Удобно процедуры семантического анализа совместить с процедурами синтаксического анализа. На практике это означает, что в рекурсивные функции встраиваются дополнительные контекстно-зависимые проверки. Например, на этапе лексического анализа в таблицу TID заносятся данные обо всех лексемах-идентификаторах, которые встречаются в тексте программы. На этапе

синтаксического анализа в ту же таблицу заносятся данные о типе идентификатора (поле type) и о наличии для него описания (поле declared).

С учетом сказанного, правила вывода для нетерминала D (разделе описаний) принимают вид:

D→stack.reset() I stack.push(c_val) {, I stack.push(c_val)} : [% dec(LEX_INT) |

***!** dec(LEX_BOOL) | **$** dec(LEX_BOOL)]*

Здесь stack – структура данных, в которую запоминаются идентификаторы (номера строк в таблице TID), dec – функция, задача которой заключается в занесении информации об идентификаторах (поля type и declared), а также контроль повторного объявления идентификатора.

Описания функций семантических проверок приведены в листинге в Приложении 2.

3.4. Тестирование программы

В качестве программного продукта разработано консольное приложение tfy.exe, Приложение принимает на вход исходный текст программы из текстового файла на модельном языке и выдает в качестве результата сообщение о синтаксической и семантической корректности написанной программы. В случае обнаружения ошибки программа выдает сообщение об ошибке с номером некорректной лексемы.

Рассмотрим примеры.

1. Исходный код программы приведен в Листинге 1.

```
leon : !;
leon = 423
(* комментарий *)
if i ~= true then input(leon) end_else
```

```
if 5 EQ 5 and true then output(2) else input(leon) end_else
do while true input(leon) loop
for(i;g;2) output(test)
end
```

Данная программа синтаксически корректна, поэтому анализатор выдает следующее сообщение (рис. 2).

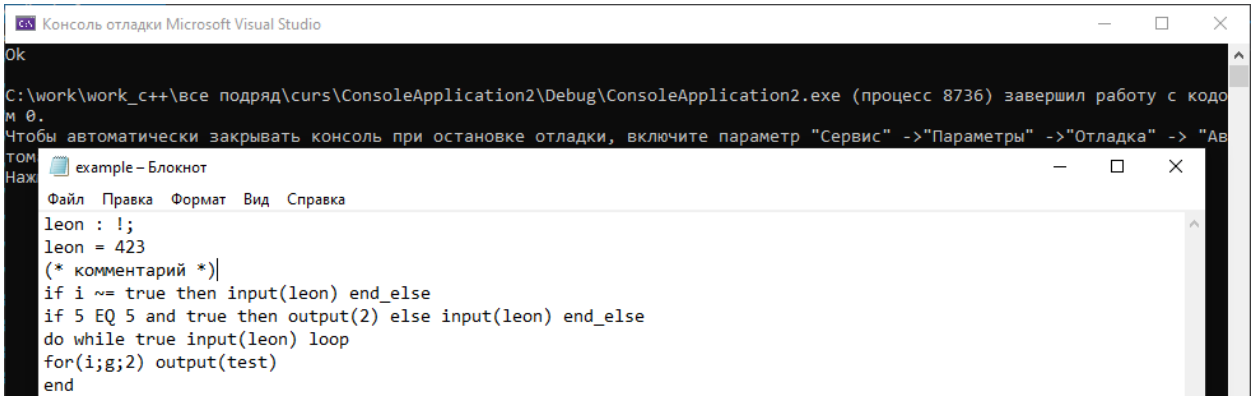

Рис. 2. Пример синтаксически корректной программы

Исходный код программы, содержащий синтаксическую ошибку, приведен на рис. 3 совместно с сообщением об ошибке.

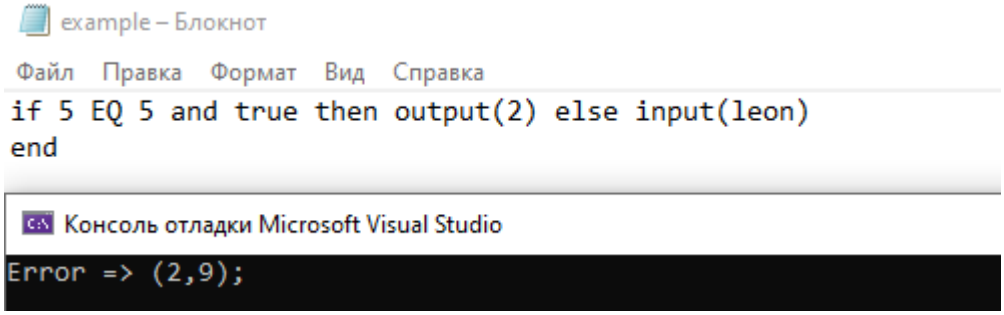

Рис. 3. Пример программы, содержащий ошибку.

Здесь ошибка допущена в строке 1: Отсутствие ключевого слова (end_else). В сообщении об ошибке указана ошибочная лексема (2,9). Тип лексемы, соответствует LEX_END_ELSE, т.е. ключевому слову «end_else»

Исходный код программы, содержащий синтаксическую ошибку, приведен на рис. 4 совместно с сообщением об ошибке.

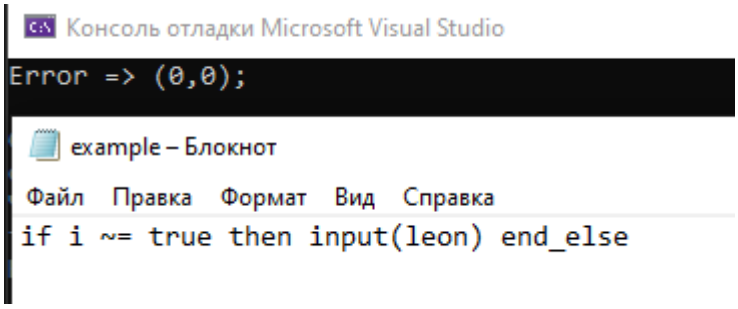

Рис. 4. Пример программы, содержащий ошибку.

Здесь ошибка допущена отсутствие ключевого слова (end). В сообщении об ошибке указана ошибочная лексема (0,0). Тип лексемы, соответствует LEX_END, т.е. ключевому слову «end»

Исходный текст программы, содержащей семантическую проверку, приведен на рис. 5 совместно с сообщением об ошибке.

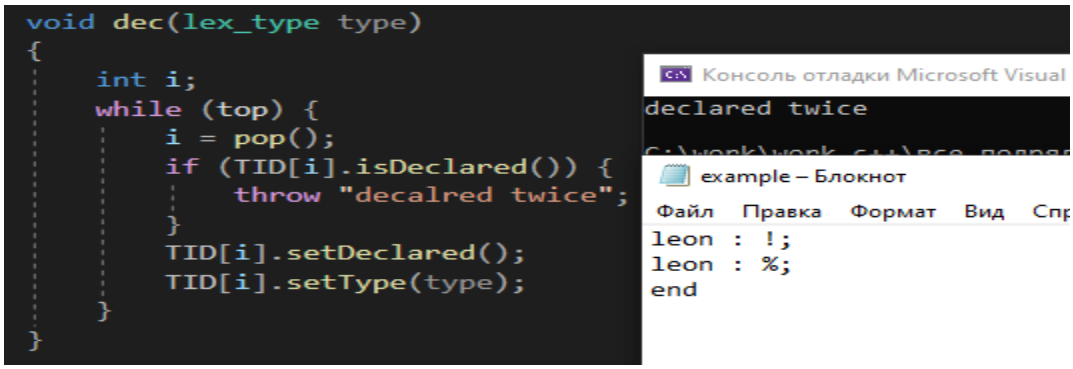

Рис. 5. Пример программы, содержащей семантическую ошибку.

ПРИЛОЖЕНИЯ

Приложение А – Класс лексического анализатора

Приложение Б – Класс синтаксического анализатора (с дополнительными семантическими проверками)

Приложение 1

Класс лексического анализатора

```
    // Типы лексем
    enum lex_type
    {
      LEX_NULL,      // 0
      // КЛЮЧЕВЫЕ СЛОВА
      LEX_AND,       // 1
      LEX_END,       // 2
      LEX_BOOL,      // 3
      LEX_REAL,      // 4
      LEX_ELSE,      // 5
      LEX_END_ELSE,      // 6
      LEX_IF,        // 7
      LEX_FALSE,     // 8
      LEX_INT,       // 9
      LEX_NOT,       // 10
      LEX_OR,        // 11
      LEX_INPUT,     // 12
      LEX_THEN,      // 13
      LEX_TRUE,      // 14
      LEX_DO,     // 15
      LEX_WHILE,     // 16
      LEX_LOOP,     // 17
      LEX_OUTPUT,     // 18
      LEX_LET,     // 19
      LEX_FOR,     // 20
      // МАРКЕР КОНЦА ТЕКСТА ПРОГРАММЫ
      LEX_FIN,       // 21
      // ОПЕРАТОРЫ И РАЗДЕЛИТЕЛИ
      LEX_SEMICOLON,  // 22
      LEX_COMMA,     // 23
      LEX_COLON,     // 24
      LEX_ASSIGN,    // 25
      LEX_LPAREN,    // 26
      LEX_RPAREN,    // 27
      LEX_EQ,        // 28
      LEX_LSS,       // 29
      LEX_GTR,       // 30
      LEX_PLUS,      // 31
      LEX_MINUS,     // 32
      LEX_TIMES,     // 33
      LEX_SLASH,     // 34
      LEX_LEQ,       // 35
      LEX_NEQ,       // 36
      LEX_GEQ,       // 37
      LEX_SCOM,      // 38
      LEX_ECOM,      // 39
      // ЧИСЛОВАЯ КОНСТАНТА
      LEX_NUM,       // 40
      // ИДЕНТИФИКАТОР
      LEX_ID         // 41
};
```

```
//////////////////////////
// ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР //
//////////////////////////

class Lexer
{
    int lencom = 0;
    enum state {
        H,       // новая строка
        ID,      // буква или цифра
        NUM,     // цифра
        COM,     // коментарий
        ALE,     // сравнение
        NEQ,     // не равно
        DELIM    // символы
    };
    state CS;
    static const char* TW[];
    static const char* TD[];
    static lex_type words[];
    static lex_type dlms[];
    FILE* fp;
    char c;
    char buf[80];
    int buf_top;

    void clear()
    {
        buf_top = 0;
        for (int i = 0; i < 80; ++i) {
            buf[i] = '\0';
        }
    }

    void add()
    {
        buf[buf_top++] = c;
    }

    int look(const char* buf, const char** list)
    {
        int i = 0;
        while (list[i]) {
            if (!strcmp(buf, list[i])) {
                return i;
            }
            i++;
        }
        return 0;
    }

    void gc()
    {
        c = (char)fgetc(fp);
    }

public:
    Lexer(const char* filename)
```

```
    {
        fp = fopen(filename, "r");
        CS = H;
        clear();
        gc();
    }

    Lex getLex();
};


// Таблица ключевых слов
const char* Lexer::TW[] = {
    "",
    "and",
    "$",
    "do",
    "while",
    "loop",
    "for",
    "else",
    "end_else",
    "end",
    "if",
    "false",
    "%",
    "!",
    "~",
    "or",
    "input",
    "then",
    "true",
    "output",
    "let",
    NULL
};

// Таблица разделителей
const char* Lexer::TD[] = {
    "",
    ";",
    ",",
    ":",
    "=",
    "(",
    ")",
    "EQ",
    "LT",
    "GT",
    "plus",
    "min",
    "mult",
    "div",
    "LE",
    "NE",
    "GE",
    "(*",
```

```
    "*)",
    NULL
};

// Таблица идентификаторов
TableId TID(100);

// Таблица типов ключевых слов
lex_type Lexer::words[] = {
    LEX_NULL,
    LEX_AND,
    LEX_BOOL,
    LEX_DO,
    LEX_WHILE,
    LEX_LOOP,
    LEX_FOR,
    LEX_ELSE,
    LEX_END_ELSE,
    LEX_END,
    LEX_IF,
    LEX_FALSE,
    LEX_INT,
    LEX_REAL,
    LEX_NOT,
    LEX_OR,
    LEX_INPUT,
    LEX_THEN,
    LEX_TRUE,
    LEX_OUTPUT,
    LEX_LET,
    LEX_NULL
};

// Таблица типов разделителей
lex_type Lexer::dlms[] = {
    LEX_NULL,
    LEX_SEMICOLON,
    LEX_COMMA,
    LEX_COLON,
    LEX_ASSIGN,
    LEX_LPAREN,
    LEX_RPAREN,
    LEX_EQ,
    LEX_LSS,
    LEX_GTR,
    LEX_PLUS,
    LEX_MINUS,
    LEX_TIMES,
    LEX_SLASH,
    LEX_LEQ,
    LEX_NEQ,
    LEX_GEQ,
    LEX_SCOM,
    LEX_ECOM,
    LEX_NULL
};
```

```
// Основная функция лексического разбора
Lex Lexer::getLex()
{
    int d, j;
    CS = H;
    do {
        switch (CS) {
        case H:
            if (c == ' ' || c == '\n' || c == '\r' || c == '\t') {
                gc();
            }


            else if (isalpha(c)) {
                clear();
                add();
                gc();
                CS = ID;
            }
            else if (isdigit(c)) {
                d = c - '0';
                gc();
                CS = NUM;
            }

            else if (c == '~') {
                clear();
                add();
                gc();
                CS = NEQ;
            }

            else {
                CS = DELIM;
            }

            break;


        case ID:
            if (isalpha(c) || isdigit(c) || c == '_') {
                add();
                gc();
            }
            else {

                j = look(buf, TD);
                if (j == 0) {
                    j = look(buf, TW);
                    if (j) {
                        return Lex(words[j], j);
                    }
                    else {
                        j = TID.put(buf);
                        return Lex(LEX_ID, j);
```

```
            }
        }
        else if (j != 0) {
            return Lex(dlms[j], j);
        }

        /*
        j = look(buf, TW);
        if (j) {
            return Lex(words[j], j);
        }
        */
        else {
            j = TID.put(buf);
            return Lex(LEX_ID, j);
        }
    }
    break;


case NUM:
    if (isdigit(c)) {
        d = d * 10 + (c - '0');
        gc();
    }
    else {
        return Lex(LEX_NUM, d);
    }
    break;


case COM:
    gc();
    lencom += 1;
    if (lencom > 50) {
        lencom = 0;
        return Lex(LEX_NULL, 0);
    }
    if (c == '*' && strlen(buf) == 0) {
        add();
    }
    if (c == ')' && strlen(buf) == 1) {
        add();
    }
    if (c == ')' && buf[strlen(buf) - 2] == '*' && strlen(buf) == 2) {
        gc();
        CS = H;
    }
    break;

case ALE:
    j = look(buf, TD);
    return Lex(dlms[j], j);
    break;


case NEQ:
```

```
            if (c == '=') {
                add();
                gc();
                j = look(buf, TD);
                return Lex(LEX_NEQ, j);
            }
            else {
                throw '!';
            }
            /*
            j = look(buf, TW);
            return Lex(LEX_NEQ, j);
                    */
            break;


        case DELIM:
            clear();
            add();


            j = look(buf, TD);
            if (j == 0) {
                j = look(buf, TW);
                gc();
                return Lex(words[j], j);
            }
            if (j) {
                gc();
                return Lex(dlms[j], j);
            }
            else {
                throw c;
            }
            break;


        }
    } while (true);
}
```

Приложение 2

Класс синтаксического анализатора

```
//////////////////////////////
// СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР //
//////////////////////////////
class Parser
{
   Lex curr_lex;       // текущая лексема
   lex_type c_type;    // её тип
   int c_val;          // её значение
   Lexer lexer;

   int stack[100];     // стек переменных для контроля повторного объявления
   int top = 0;




   // рекурсивные функции
   void P();
   void D1();
   void D();
   void B();
   void S();
   void E();
   void E1();
   void T();
   void F();

   // Получить очередную лексему
   void gl()
   {
      curr_lex = lexer.getLex();
      c_type = curr_lex.getType();
      c_val = curr_lex.getValue();
   }

   void reset()
   {
      top = 0;
   }

   void push(int i)
   {
      stack[top] = i;
      ++top;
   }

   int pop()
   {
      --top;
      return stack[top];
   }
```

```
    void dec(lex_type type)
    {
        int i;
        while (top) {
            i = pop();
            if (TID[i].isDeclared()) {
                throw "decalred twice";
            }
            TID[i].setDeclared();
            TID[i].setType(type);
        }
    }

public:

    // Провести синтаксический разбор
    void analyze()
    {
        gl();
        P();
    }

    // Конструктор
    Parser(const char* filename) : lexer(filename), top(0) {}
};


// P->  S|D1 (: | \n) {S|D1}
void Parser::P()
{
    while (true) {

        if (c_type == LEX_ID) {

            /*
            gl();
            while (c_type == LEX_COMMA) {
                gl();
                D1();
            }
            */
            D1();
        }

        else if(c_type == LEX_END){
            break;
        }


        else if (c_type == LEX_NULL) {
            throw curr_lex;
        }

        S();

        if (c_type == LEX_END) {
            break;
```

```
        }

        gl();
    }

}


void Parser::S()
{

    if (c_type == LEX_IF) {
        gl();
        E();
        if (c_type == LEX_THEN) {
            gl();
            S();
            gl();
            if (c_type == LEX_ELSE) {
                gl();
                S();
                gl();
            }
            if (c_type != LEX_END_ELSE) {
                throw curr_lex;
            }
        }
        else {
            throw curr_lex;
        }
    }

    else if (c_type == LEX_DO) {
        gl();
        if (c_type != LEX_WHILE) {
            throw curr_lex;
        }
        while(c_type != LEX_LOOP) {
            gl();
            S();
        }
    }

    else if (c_type == LEX_FOR) {
        gl();
        if (c_type != LEX_LPAREN) {
            throw curr_lex;
        }
        gl();
        while (c_type != LEX_SEMICOLON)
        {
            gl();
        }
        gl();
        while  (c_type != LEX_SEMICOLON) {
            gl();
        }
```

```
        gl();
        while (c_type != LEX_RPAREN) {
            gl();
        }
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
    }

    else if (c_type == LEX_INPUT) {
        gl();
        if (c_type != LEX_LPAREN) {
            throw curr_lex;
        }
        gl();
        if (c_type != LEX_ID) {
            throw curr_lex;
        }
        gl();
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
    }
    else if (c_type == LEX_OUTPUT) {
        gl();
        if (c_type != LEX_LPAREN) {
            throw curr_lex;
        }
        gl();
        E();
        if (c_type != LEX_RPAREN) {
            throw curr_lex;
        }
    }
    else if (c_type == LEX_ID) {
        gl();
        if (c_type != LEX_ASSIGN) {
            throw curr_lex;
        }
        gl();
        E();
    }
}


// D1→ I {,I} : (%|!|$);
void Parser::D1()
{
    reset();
    if (c_type != LEX_ID) {
        throw curr_lex;
    }
    push(c_val);
    gl();
    while (c_type == LEX_COMMA) {
        gl();
        if (c_type != LEX_ID) {
```

```
            throw curr_lex;
        }
        else {
            push(c_val);
            gl();
        }
    }
    if (c_type != LEX_COLON) {
        throw curr_lex;
    }
    gl();


    if (c_type == LEX_INT) {
        this->dec(LEX_INT);
        gl();
        if (c_type != LEX_SEMICOLON) {
            throw curr_lex;
        }
    }
    else if (c_type == LEX_BOOL) {
        this->dec(LEX_BOOL);
        gl();
        if (c_type != LEX_SEMICOLON) {
            throw curr_lex;
        }
    }
    else if (c_type == LEX_REAL) {
        this->dec(LEX_REAL);
        gl();
        if (c_type != LEX_SEMICOLON) {
            throw curr_lex;
        }
    }
    else {
        throw curr_lex;
    }
}



// E→E1{[NE | EQ | LT | LE | GT | GE]E1}
void Parser::E()
{
    E1();
    if (c_type == LEX_EQ ||
        c_type == LEX_LSS ||
        c_type == LEX_GTR ||
        c_type == LEX_LEQ ||
        c_type == LEX_GEQ ||
        c_type == LEX_NEQ)
    {
        gl();
        E1();
    }
}
```

```
// E1→T{[ plus | min | or ] T}
void Parser::E1()
{
   T();
   while (c_type == LEX_PLUS || c_type == LEX_MINUS || c_type == LEX_OR) {
      gl();
      T();
   }
}

// T→F{[ mult  | div | and ] F}
void Parser::T()
{
   F();
   while (c_type == LEX_TIMES || c_type == LEX_SLASH || c_type == LEX_AND) {
      gl();
      F();
   }
}

// F→I | N | L | ~ F | (E)
void Parser::F()
{
   if (c_type == LEX_ID) {
      gl();
   }
   else if (c_type == LEX_NUM) {
      gl();
   }
   else if (c_type == LEX_TRUE || c_type == LEX_FALSE) {
      gl();
   }
   else if (c_type == LEX_NOT) {
      gl();
      F();
   }
   else if (c_type == LEX_LPAREN) {
      gl();
      E();
      if (c_type != LEX_RPAREN) {
         throw curr_lex;
      }
      gl();
   }
   else {
      throw curr_lex;
   }
}
```