

# МИНОБРАЗОВАНИЯ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА  
ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности  
Высшая школа технологий искусственного интеллекта  
Направление 02.03.01 Математика и компьютерные науки

Отчет по дисциплине «Математическая логика и теория автоматов»

Лабораторная работа №4

«Доработка компилятора языка MiLan»

Вариант 13

Обучающийся: \_\_\_\_\_

Шклярова Ксения Алексеевна

Группа: 5130201/20102

Руководитель: \_\_\_\_\_

Востров Алексей Владимирович

«\_\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

Санкт-Петербург, 2025

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Математическое описание</b>	<b>4</b>
1.1 Язык . . . . .	4
1.1.1 Язык Milan . . . . .	4
1.2 Стековая машина . . . . .	4
1.2.1 Цикл выполнения команды . . . . .	5
1.3 Порождающая грамматика Хомского . . . . .	5
1.4 БНФ-нотация (форма Бэкуса-Наура) . . . . .	5
1.5 Грамматика языка Milan . . . . .	6
1.6 Синтаксические диаграммы . . . . .	7
<b>2 Особенности реализации</b>	<b>9</b>
2.1 Добавление токенов . . . . .	9
2.2 Реализация интервального типа . . . . .	15
2.2.1 Вспомогательные методы . . . . .	17
2.3 Модификация синтаксического анализатора . . . . .	21
<b>3 Результаты работы программы</b>	<b>24</b>
<b>Заключение</b>	<b>29</b>
<b>Список литературы</b>	<b>30</b>

## Введение

Данная лабораторная работа представляет собой расширение компилятора языка Milan, реализованного на языке C++. Согласно варианту 13 необходимо добавить поддержку интервального типа в следующих видах:  $[< expression > (".." | expression > "..") < expression >]$ ,  $(< expression > (".." | expression > "..") < expression >)$ ,  $[< expression > (".." | expression > "..") < expression >]$ ,  $(< expression > (".." | expression > "..") < expression >)$ .

# 1 Математическое описание

## 1.1 Язык

Языком над конечным словарем  $\Sigma$  называется произвольное множество конечных цепочек над этим словарем.

- Цепочки языка называются *словами* (предложениями).
- Над конечным непустым словарем можно определить бесконечное количество слов конечной длины (счетное множество).
- Язык — *подмножество* цепочек конечного словаря. Над конечным непустым словарем можно определить бесконечное количество языков, т. е. подмножеств множества всех возможных слов (континуум, как число подмножеств счетного множества).

### 1.1.1 Язык Milan

Язык Milan - паскалеподобный неавтоматный язык программирования, определяемый как:

$$L_{\text{Milan}} \subseteq \Sigma^*, \text{ где:}$$

- $\Sigma$  — конечный словарь допустимых символов (алфавит);

$$\Sigma = \{+, ', 0, \dots, :, =, \dots, \text{begin}, \text{end}, \text{if} \dots\}$$

- $\Sigma^*$  — множество всех возможных цепочек конечной длины над  $\Sigma$ ;
- $L_{\text{Milan}}$  — подмножество  $\Sigma^*$ , содержащее только корректные программы согласно грамматике Milan.

Транслятор (компилятор) языка Milan переводит программу на этом языке в программу на языке стековой машины (промежуточный язык). Программа на языке стековой машины обычно потом интерпретируется.

## 1.2 Стековая машина

Стековая машина — виртуальный однопроцессорный компьютер с простой архитектурой. Содержит следующие компоненты:

- Память данных (ПД) — линейная память, где каждый регистр хранит одно целое число;
- Память программ (ПП) — линейная память, где каждый регистр хранит одну команду (код операции и адрес);

- Стек — линейная память с доступом только к верхнему элементу (LIFO);
- Счетчик команд (СчК) — регистр, хранящий адрес текущей выполняемой команды;
- Регистр команд — хранит декодированную выполняемую команду (одноадресную);
- Регистры А и В — временные регистры для операций с данными.

### 1.2.1 Цикл выполнения команды

Цикл выполнения команды в стековой машине всегда одинаков и состоит из следующих шагов:

- Адрес очередной выполняемой команды находится в счетчике Команд (СчК). Перед выполнением программы СчК устанавливается в 0.
- Из памяти программ (ПП) по адресу, находящемуся в СчК, выбирается код (КодОп, Адрес) и помещается в регистр команд:

$$\text{Регистр Команд} \leftarrow \text{ПП}[\text{СчК}]$$

- Содержимое счетчика команд увеличивается на 1 (подготовка к следующей команде):

$$\text{СчК} \leftarrow \text{СчК} + 1$$

- Поле КодОп регистра команд декодируется, и выполняется соответствующая операция.

## 1.3 Порождающая грамматика Хомского

Порождающая грамматика Хомского формально определяется как четверка  $G = (T, N, S, R)$ , где:

- $T$  — конечное множество терминалов (терминальный словарь);
- $N$  — конечное множество нетерминалов (нетерминальный словарь), причем  $T \cap N = \emptyset$ ;
- $S \in N$  — начальный нетерминал;
- $R$  — конечное множество правил вида  $\alpha \rightarrow \beta$ , где  $\alpha$  и  $\beta$  — цепочки символов из  $T \cup N$ .

Причем  $\alpha$  должна содержать хотя бы один нетерминал.

## 1.4 БНФ-нотация (форма Бэкуса-Наура)

БНФ-нотация — формальное задание порождающих грамматик формальных языков, при котором:

- Нетерминалы помещаются в угловые скобки;

- Символ ::= используется вместо стрелки
- Альтернативы одного и того же нетерминала записываются в правой части одного правила и через знак |, имеющий смысл «либо»

Чтобы пометить элементы, как необязательные, в БНФ можно их поместить в квадратные скобки [ а ] - повторение 0 или 1 раз. А чтобы задать повторяющиеся элементы, нужно использовать фигурные скобки { а } - повторение 0 или произвольное число раз.

## 1.5 Грамматика языка Milan

Грамматика языка Milan с дополнениями в БНФ-нотации приведена ниже:

```

1 <program> ::= "BEGIN" <statement_list> "END"
2
3 <statement_list> ::= <statement> (";" <statement_list>)* |
4
5 <statement> ::= <assignment>
6               | <if_statement>
7               | <while_statement>
8               | <write_statement>
9
10 <assignment> ::= <identifier> ":" <expression>
11               | <identifier> ":" <interval>
12
13 <if_statement> ::= "IF" <relation> "THEN" <statement_list>
14               ("ELSE" <statement_list>)? "FI"
15
16 <while_statement> ::= "WHILE" <relation> "DO" <statement_list> "OD"
17
18 <write_statement> ::= "WRITE" "(" (<expression> | <identifier>) ")"
19
20 <expression> ::= <term> (("+" | "-") <term>)*
21
22 <term> ::= <factor> (("*" | "/" ) <factor>)*
23
24 <factor> ::= <number>
25           | <identifier>
26           | "-" <factor>
27           | "READ"
28           | "(" <expression> ")"
29           | <interval>
30
31 <relation> ::= <expression> ("=" | "!=" | "<" | "<=" | ">" | ">=") <expression>
32
33 <interval> ::= "[" <expression> (".." | "," <expression> "..") <expression> "]"
34           | "(" <expression> (".." | "," <expression> "..") <expression> ")"
35           | "[" <expression> (".." | "," <expression> "..") <expression> "]"
36           | "(" <expression> (".." | "," <expression> "..") <expression> "]"
37

```

```
38 <identifier> ::= <letter> (<letter> | <digit>)*
39
40 <number> ::= <digit>+
41
42 <letter> ::= "a".."z" | "A".."Z"
43
44 <digit> ::= "0".."9"
```

В исходную грамматику языка были добавлены новые лексемы: T\_LBRACKET — '[', T\_RBRACKET — ']', T\_DOTDOT — '..' (диапазон), T\_COMMA — ',' (шаг в интервалах). А так же новые синтаксические инструкции для представления интервалов <interval> и для вывода всех элементов заданного интервала <write\_statement>.

## 1.6 Синтаксические диаграммы

На рис. 1 представлены синтаксические диаграммы для всей грамматики Milan.

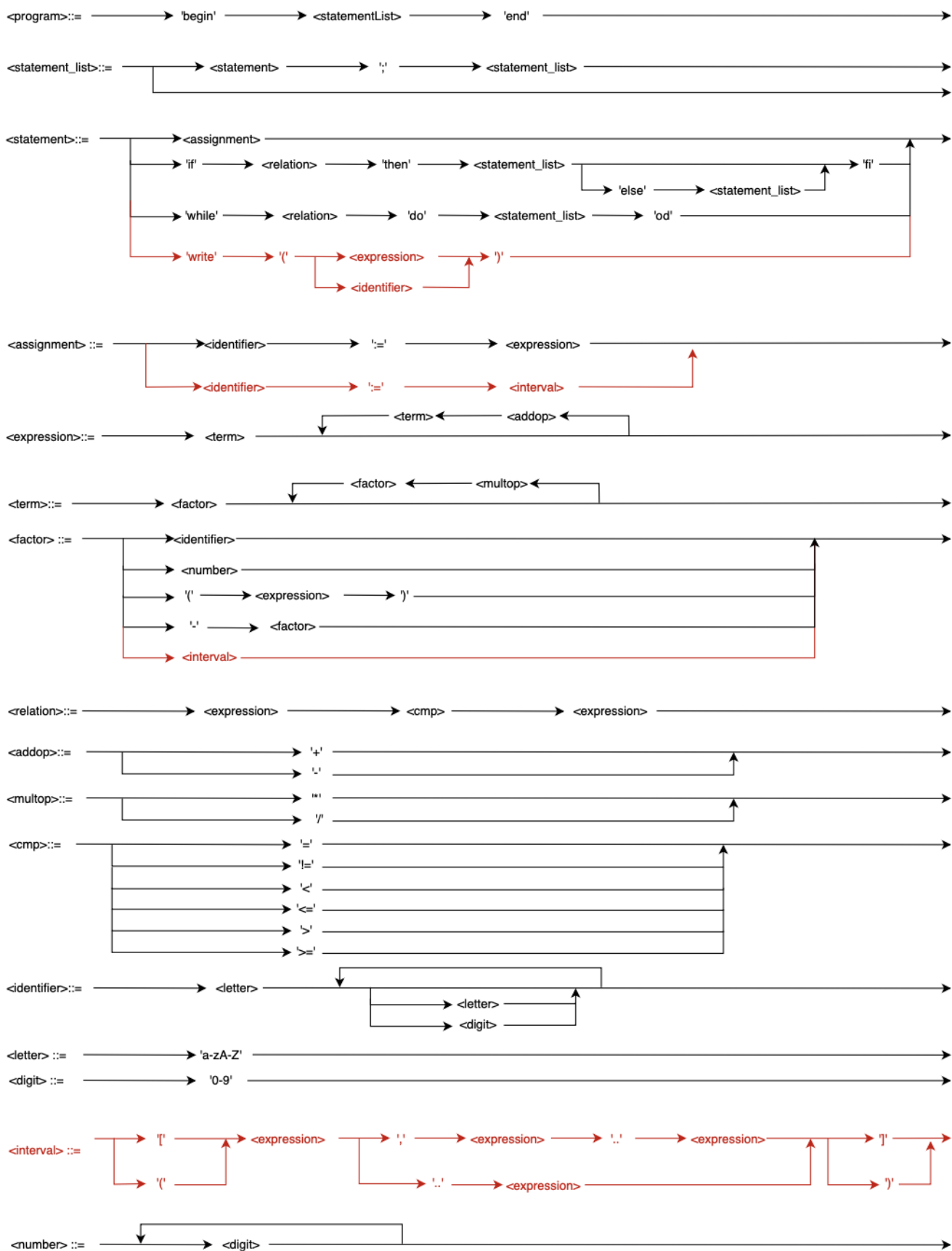


Рис. 1. Синтаксические диаграммы



## 2 Особенности реализации

### 2.1 Добавление токенов

В файл `scanner.h` добавлены новые токены для работы с интервалами:

```
1 enum Token {
2     T_EOF,          // Конец текстового потока
3     T_ILLEGAL,      // Признак недопустимого символа
4     T_IDENTIFIER,   // Идентификатор
5     T_NUMBER,       // Целочисленный литерал
6     T_BEGIN,        // Ключевое слово "begin"
7     T_END,          // Ключевое слово "end"
8     T_IF,           // Ключевое слово "if"
9     T_THEN,         // Ключевое слово "then"
10    T_ELSE,          // Ключевое слово "else"
11    T_FI,            // Ключевое слово "fi"
12    T_WHILE,         // Ключевое слово "while"
13    T_DO,            // Ключевое слово "do"
14    T_OD,            // Ключевое слово "od"
15    T_WRITE,         // Ключевое слово "write"
16    T_READ,          // Ключевое слово "read"
17    T_ASSIGN,        // Оператор ":@"
18    T_ADDOP,
19    T_MULOP,
20    T_CMP,           // Сводная лексема для операторов отношения
21    T_LPAREN,        // Открывающая скобка
22    T_RPAREN,        // Закрывающая скобка
23    T_SEMICOLON,
24    T_LBRACKET,
25    T_RBRACKET,
26    T_DOTDOT,
27    T_COMMA
28 };
```

В массив `tokenNames_[]` в файле `scanner.cpp` были добавлены строковые представления новых токенов. Изменения см. в листинге 1.

Листинг 1. Реализация `tokenNames`

```
1 static const char* tokenNames_[] = {
2     "end of file",
3     "illegal token",
4     "identifier",
5     "number",
6     "'BEGIN'",
7     "'END'",
8     "'IF'",
9     "'THEN'",
10    "'ELSE'",
```

```

11    "'FI'",
12    "'WHILE'",
13    "'DO'",
14    "'OD'",
15    "'WRITE'",
16    "'READ'",
17    "':=' ",
18    "'+' or '-' ",
19    "'*' or '/' ",
20    "comparison operator",
21    "'(' ",
22    "')' ",
23    "';' ",
24    "'[' ",
25    "']' ",
26    "'..' ",
27    "',' ",
28 };

```

В метод `nextToken()` была добавлена обработка этих символов, реализацию метода см. в листинге 2. Данный метод разбивает исходный код на токены (лексемы).

Листинг 2. Реализация `nextToken()`

```

1 void Scanner::nextToken()
2 {
3     skipSpace();
4
5     // Обработка комментариев
6     while (ch_ == '/') {
7         nextChar();
8         if (ch_ == '*') {
9             nextChar();
10            bool inside = true;
11            while (inside) {
12                while (ch_ != '*' && !input_.eof()) {
13                    nextChar();
14                }
15
16                if (input_.eof()) {
17                    token_ = T_EOF;
18                    return;
19                }
20
21                nextChar();

```

```

22         if (ch_ == '/') {
23             inside = false;
24             nextChar();
25         }
26     }
27 }
28 else {
29     token_ = T_MULOP;
30     arithmeticValue_ = A_DIVIDE;
31     return;
32 }
33
34 skipSpace();
35 }
36
37 // Конец файла
38 if (input_.eof()) {
39     token_ = T_EOF;
40     return;
41 }
42
43 // Числовые литералы
44 if (isdigit(static_cast<unsigned char>(ch_))) {
45     int value = 0;
46     while (isdigit(static_cast<unsigned char>(ch_))) {
47         value = value * 10 + (static_cast<int>(ch_) - '0');
48         nextChar();
49     }
50     token_ = T_NUMBER;
51     intValue_ = value;
52 }
53 // Идентификаторы и ключевые слова
54 else if (isIdentifierStart(ch_)) {
55     string buffer;
56     while (isIdentifierBody(ch_)) {
57         buffer += static_cast<char>(ch_);
58         nextChar();
59     }
60
61     // Безопасное преобразование в нижний регистр
62     for (char& c : buffer) {
63         c = static_cast<char> (::tolower(static_cast<unsigned char>(c)));
64     }
65

```

```

66     auto kwd = keywords_.find(buffer);
67     if (kwd == keywords_.end()) {
68         token_ = T_IDENTIFIER;
69         stringValue_ = buffer;
70     }
71     else {
72         token_ = kwd->second;
73     }
74 }
75 // Специальные символы и операторы
76 else {
77     switch (ch_) {
78         case '(':
79             token_ = T_LPAREN;
80             nextChar();
81             break;
82
83         case ')':
84             token_ = T_RPAREN;
85             nextChar();
86             break;
87
88         case ';':
89             token_ = T_SEMICOLON;
90             nextChar();
91             break;
92
93         case ':':
94             nextChar();
95             if (ch_ == '=') {
96                 token_ = T_ASSIGN;
97                 nextChar();
98             }
99             else {
100                 token_ = T_ILLEGAL;
101             }
102             break;
103
104         case '<':
105             token_ = T_CMP;
106             nextChar();
107             if (ch_ == '=') {
108                 cmpValue_ = C_LE;
109                 nextChar();

```

```

110     }
111     else {
112         cmpValue_ = C_LT;
113     }
114     break;
115
116 case '>':
117     token_ = T_CMP;
118     nextChar();
119     if (ch_ == '=') {
120         cmpValue_ = C_GE;
121         nextChar();
122     }
123     else {
124         cmpValue_ = C_GT;
125     }
126     break;
127
128 case '!':
129     nextChar();
130     if (ch_ == '=') {
131         token_ = T_CMP;
132         cmpValue_ = C_NE;
133         nextChar();
134     }
135     else {
136         token_ = T_ILLEGAL;
137     }
138     break;
139
140 case '=':
141     token_ = T_CMP;
142     cmpValue_ = C_EQ;
143     nextChar();
144     break;
145
146 case '+':
147     token_ = T_ADDOP;
148     arithmeticValue_ = A_PLUS;
149     nextChar();
150     break;
151
152 case '-':
153     token_ = T_ADDOP;

```

```

154         arithmeticValue_ = A_MINUS;
155         nextChar();
156         break;
157
158     case '*':
159         token_ = T_MULOP;
160         arithmeticValue_ = A_MULTIPLY;
161         nextChar();
162         break;
163
164     case '[':
165         token_ = T_LBRACKET;
166         nextChar();
167         break;
168
169     case ']':
170         token_ = T_RBRACKET;
171         nextChar();
172         break;
173
174     case '.':
175         nextChar();
176         if (ch_ == '.') {
177             token_ = T_DOTDOT;
178             nextChar();
179         }
180         else {
181             token_ = T_ILLEGAL;
182         }
183         break;
184
185     case ',':
186         token_ = T_COMMA;
187         nextChar();
188         break;
189
190     default:
191         token_ = T_ILLEGAL;
192         nextChar();
193         break;
194 }
195 }
196 }

```

## 2.2 Реализация интервального типа

Основным методом для реализации интервала является метод `interval()`, реализованный в классе `Parser`. На вход метод не принимает параметров, но использует текущий токен из сканера (`scanner_`). Результатом его выполнения является генерация байт-кода для создания интервала и сохранение его в памяти. Реализация заключается в следующем: определяется тип скобки : `[` или `(`. Считывается начальное значение (`start`), шаг (`step`, если есть) и конечное значение (`end`). Корректируются границы в зависимости от типа скобок (включение/исключение). Значения интервала записываются в память с помощью `PUSH` и `STORE`. Обновляется `lastVar_` для выделения памяти под следующий интервал.

Реализацию данного метода см. в листинге 3.

Листинг 3. Реализация создания интервала

```
1 void Parser::interval() {
2     bool isBracket = false;
3     bool isParen = false;
4     bool closeBracket = false;
5     bool closeParen = false;
6
7     // Определяем тип открывающей скобки
8     if (match(T_LBRACKET)) {
9         isBracket = true;
10    }
11    else if (match(T_LPAREN)) {
12        isParen = true;
13    }
14    else {
15        reportError("expected '[' or '(' at the beginning of interval");
16        return;
17    }
18
19    // Проверка на пустой интервал
20    if (match(T_RBRACKET) || match(T_RPAREN)) {
21        reportError("interval cannot be empty");
22        return;
23    }
24
25    int start = readSignedInt();
26    int step = 1;
27    int end = 1;
28    int size = 1;
29
30    if (match(T_COMMA)) {
```

```

31     step = start;
32     start = readSignedInt();
33     mustBe(T_DOTDOT);
34     end = readSignedInt();
35
36     if (step <= 0) {
37         reportError("step cannot be zero or negative");
38         return;
39     }
40 }
41 else if (match(T_DOTDOT)) {
42     end = readSignedInt();
43 }
44 else {
45     reportError("expected '..' or ',' in interval");
46     return;
47 }
48
49 // Определяем тип открывающей скобки
50 if (match(T_RBRACKET)) {
51     closeBracket = true;
52 }
53 else if (match(T_LPAREN)) {
54     closeParen = true;
55 }
56 else {
57     reportError("expected '[' or '(' at the end of interval");
58     return;
59 }
60
61 size = abs((end - start) / step);
62 if (start <= end) {
63     if (start == end && !isParen && closeParen) {
64         reportError("there must be [] or ()");
65         return;
66     }
67     if (isParen) {
68         if (start + step > end) {
69             reportError("step is too big");
70             return;
71         }
72         start += step;
73     }
74     else size++;

```



```

75     if (closeParen) {
76         end--;
77         size--;
78     }
79 }
80
81 else {
82     if (isParen) {
83         if (start - step < end) {
84             reportError("step is too big");
85             return;
86         }
87         start -= step;
88     }
89     else size++;
90     if (closeParen) {
91         end++;
92         size--;
93     }
94     step = -1 * step;
95     if (size == 0) size++;
96 }
97
98 int baseAddress = lastVar_;
99 for (int i = 0; i < size; i++) {
100     codegen_ ->emit(PUSH, start + i * step);
101     codegen_ ->emit(STORE, lastVar_ + i);
102 }
103
104 if (!currentVarName_.empty()) {
105     arrays_[currentVarName_] = { baseAddress, size };
106 }
107
108 lastVar_ += size;
109 }

```

### 2.2.1 Вспомогательные методы

Метод `readSignedInt()` предназначен для разбора целочисленных выражений, включая унарный минус и простые арифметические операции (+, -, \*, /). Метод не принимает параметров, но использует текущее состояние сканера для чтения токенов. Возвращает вычисленное целое значение выражения. Логика работы: если текущий токен — унарный минус (`A_MINUS`), метод считывает его, затем ожидает число, которое возвращает со знаком минус. В противном слу-

чае, метод вызывает `parseFactor()` для разбора множителя, после чего обрабатывает сложение и вычитание, последовательно вызывая `parseFactor()` для каждого терма.

Метод `parseFactor()` отвечает за разбор и вычисление множителей (factors) в арифметических выражениях, включая операции умножения и деления с учётом их приоритета. Не принимает параметров, работает на основе текущего состояния сканера. Возвращает вычисленное целое число — результат разбора множителя. Работает следующим образом: сначала вызывает `parsePrimary()` для получения начального значения (число, переменная или выражение в скобках). Затем, пока встречаются операторы `*` или `/`, продолжает разбор: запоминает оператор (умножение или деление), получает следующее значение через `parsePrimary()`, выполняет соответствующую операцию над накопленным результатом.

Метод `parseArithmeticExpression()` выполняет разбор арифметического выражения с учетом приоритетов операций. Не принимает параметров, использует текущее состояние сканера для чтения токенов. Возвращает результат вычисления всего выражения. Работа метода: сначала вызывает `parseTerm()` для разбора первого терма (умножение/деление). Далее, если встречаются операции сложения или вычитания, продолжает разбор: сохраняет оператор, разбирает следующий терм через `parseTerm()`, применяет операцию к накопленному результату. Итоговое значение возвращается как результат всего выражения.

Метод `parseTerm()` отвечает за разбор термов — подвыражений, содержащих операции умножения и деления. Принимает данные из текущего состояния сканера. Возвращает вычисленное значение терма. Считывает базовое значение через `parsePrimary()` (число, переменная или выражение в скобках). Пока встречаются операторы `*` или `/`, продолжает разбор: запоминает оператор, считывает новое значение через `parsePrimary()`, Применяет операцию к текущему результату. Возвращает итоговое значение терма.

Метод `parsePrimary()` отвечает за разбор базовых элементов выражения — чисел, переменных и выражений в скобках. Не принимает параметров, использует текущие токены из сканера. Возвращает вычисленное значение соответствующего элемента. Работа метода: если встречено число — возвращает его значение. Если встречена переменная — ищет её значение в контексте (`variableValues_`) и возвращает его. Если встречена открывающая скобка — рекурсивно вызывает `parseArithmeticExpression()` для разбора содержимого скобок. В случае ошибки — генерирует сообщение о некорректном выражении и возвращает 0.

Реализацию данных методов см. в листинге 4.

Листинг 4. Реализация вспомогательных методов

```
1 int Parser::readSignedInt() {
2     if (see(T_ADDOP) && scanner_ ->getArithmeticValue() == A_MINUS) {
3         next();
4         if (!see(T_NUMBER)) {
```

```

5      reportError("number expected");
6      return 0;
7  }
8      int value = scanner_ ->getIntValue();
9      next();
10     return -value;
11 }
12
13 else {
14     int left = parseFactor();
15
16     while (see(T_ADDOP)) {
17         Arithmetic op = scanner_ ->getArithmeticValue();
18         next();
19         int right = parseFactor();
20         left = (op == A_PLUS) ? left + right : left - right;
21     }
22
23     return left;
24 }
25 return 1;
26 }
27
28 int Parser::parseFactor() {
29     int left = parsePrimary();
30
31     while (see(T_MULOP)) {
32         Arithmetic op = scanner_ ->getArithmeticValue();
33         next();
34         int right = parsePrimary();
35
36         if (op == A_MULTIPLY) {
37             left *= right;
38         }
39         else {
40             if (right == 0) {
41                 reportError("Division by zero");
42                 return 0;
43             }
44             left /= right;
45         }
46     }
47
48     return left;

```

```

49 }
50
51 // функция для разбора арифметических выражений с приоритетами
52 int Parser::parseArithmeticExpression() {
53     int left = parseTerm(); // Сначала разбираем умножение/деление
54
55     while (see(T_ADDOP)) {
56         Arithmetic op = scanner_ -> getArithmeticValue();
57         next();
58         int right = parseTerm();
59
60         left = (op == A_PLUS) ? left + right : left - right;
61     }
62
63     return left;
64 }
65
66 int Parser::parsePrimary() {
67     if (see(T_NUMBER)) {
68         int value = scanner_ -> getIntValue();
69         next();
70         return value;
71     }
72     else if (see(T_IDENTIFIER)) {
73         std::string name = scanner_ -> getStringValue();
74         next();
75
76         // Найти переменную в таблице
77         auto it = variableValues_.find(name);
78         if (it == variableValues_.end()) {
79             reportError("Undefined variable: " + name);
80             return 0;
81         }
82
83         return it -> second;
84     }
85     else if (see(T_LPAREN)) {
86         next();
87         int value = parseArithmeticExpression();
88         mustBe(T_RPAREN);
89         return value;
90     }
91     else {
92         reportError("Number, variable, or parenthesized expression expected");

```

```

93     return 0;
94 }
95 }
96
97 // Разбор термов (умножение/деление)
98 int Parser::parseTerm() {
99     int left = readSignedInt(); // Рекурсивно читаем числа/скобки
100
101     while (see(T_MULOP)) {
102         Arithmetic op = scanner_ ->getArithmeticValue();
103         next();
104         int right = readSignedInt();
105
106         if (op == A_MULTIPLY) {
107             left *= right;
108         }
109         else {
110             if (right == 0) {
111                 reportError("Division by zero");
112                 return 0;
113             }
114             left /= right;
115         }
116     }
117
118     return left;
119 }

```

## 2.3 Модификация синтаксического анализатора

Изменения в методе statement():

1. Обработка присваивания интервалов. Добавлена проверка на наличие интервала после оператора присваивания =. Если после = идет [ или (, вызывается метод interval().
2. Поддержка вывода интервалов через WRITE. Добавлена проверка, является ли переменная интервалом при выводе, если является, то выводятся все значения сохраненного массива.

Реализацию данного метода с изменениями см. в листинге 5.

Листинг 5. Реализация statement()

```

1 void Parser::statement()
2 {
3     if(see(T_IDENTIFIER)) {
4         string varName = scanner_ ->getStringValue();
5         currentVarName_ = varName;

```

```

6      int varAddress = findOrAddVariable(varName);
7      next();
8      mustBe(T_ASSIGN);
9
10     if (see(T_LBRACKET) || see(T_LPAREN)) {
11         interval();
12     }
13     else {
14         int value = parseArithmeticExpression();
15         codegen_ ->emit(PUSH, value);
16         codegen_ ->emit(STORE, varAddress);
17
18         variableValues_[varName] = value;
19     }
20     currentVarName_ = "";
21 }
22 else if(match(T_IF)) {
23     relation();
24
25     int jumpNoAddress = codegen_ ->reserve();
26
27     mustBe(T_THEN);
28     statementList();
29     if(match(T_ELSE)) {
30         int jumpAddress = codegen_ ->reserve();
31         codegen_ ->emitAt(jumpNoAddress, JUMP_NO, codegen_ ->getCurrentAddress());
32         statementList();
33         codegen_ ->emitAt(jumpAddress, JUMP, codegen_ ->getCurrentAddress());
34     }
35     else {
36         codegen_ ->emitAt(jumpNoAddress, JUMP_NO, codegen_ ->getCurrentAddress());
37     }
38
39     mustBe(T_FI);
40 }
41
42 else if(match(T_WHILE)) {
43     int conditionAddress = codegen_ ->getCurrentAddress();
44     relation();
45     int jumpNoAddress = codegen_ ->reserve();
46     mustBe(T_DO);
47     statementList();
48     mustBe(T_OD);
49     codegen_ ->emit(JUMP, conditionAddress);

```

```

50     codegen_ ->emitAt(jumpNoAddress, JUMP_NO, codegen_ ->getCurrentAddress());
51 }
52 else if (match(T_WRITE)) {
53     mustBe(T_LPAREN);
54
55     if (see(T_IDENTIFIER)) {
56         string varName = scanner_ ->getStringValue();
57         next();
58
59         if (arrays_.find(varName) != arrays_.end()) {
60             ArrayInfo arr = arrays_[varName];
61             for (int i = 0; i < arr.size; ++i) {
62                 codegen_ ->emit(LOAD, arr.baseAddress + i);
63                 codegen_ ->emit(PRINT);
64             }
65         }
66         else {
67             int addr = findOrAddVariable(varName);
68             codegen_ ->emit(LOAD, addr);
69             codegen_ ->emit(PRINT);
70         }
71     }
72     else {
73         expression();
74         codegen_ ->emit(PRINT);
75     }
76     mustBe(T_RPAREN);
77 }
78 else {
79     reportError("statement expected.");
80 }
81 }

```

### 3 Результаты работы программы

#### Пример №1: полуоткрытый интервал слева

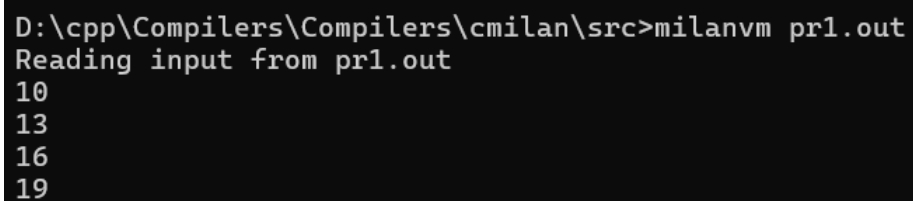
Исходный код программы:

```
1 BEGIN
2   a := 7;
3   b := 20;
4   x := (3, a .. b];
5   write(x)
6 END
```

ОП код:

```
1 0:  PUSH  7
2 1:  STORE 0
3 2:  PUSH  20
4 3:  STORE 1
5 4:  PUSH  10
6 5:  STORE 3
7 6:  PUSH  13
8 7:  STORE 4
9 8:  PUSH  16
10 9:  STORE 5
11 10: PUSH  19
12 11: STORE 6
13 12: LOAD  3
14 13: PRINT
15 14: LOAD  4
16 15: PRINT
17 16: LOAD  5
18 17: PRINT
19 18: LOAD  6
20 19: PRINT
21 20: STOP
```

Результат выполнения программы представлен на рис. 2:



```
D:\cpp\Compilers\Compilers\cmilan\src>milanvm pr1.out
Reading input from pr1.out
10
13
16
19
```

Рис. 2. Результат программы для примера №1



## Пример №2: замкнутый интервал

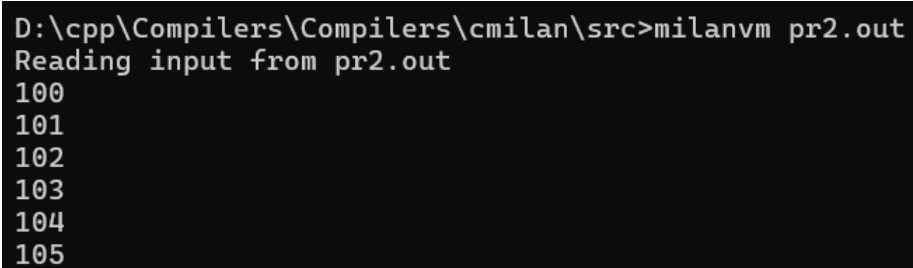
Исходный код программы:

```
1 BEGIN
2   x := [100..105];
3   write(x)
4 END
```

ОП код:

```
1 0:  PUSH  100
2 1:  STORE 1
3 2:  PUSH  101
4 3:  STORE 2
5 4:  PUSH  102
6 5:  STORE 3
7 6:  PUSH  103
8 7:  STORE 4
9 8:  PUSH  104
10 9:  STORE 5
11 10: PUSH  105
12 11: STORE 6
13 12: LOAD  1
14 13: PRINT
15 14: LOAD  2
16 15: PRINT
17 16: LOAD  3
18 17: PRINT
19 18: LOAD  4
20 19: PRINT
21 20: LOAD  5
22 21: PRINT
23 22: LOAD  6
24 23: PRINT
25 24: STOP
```

Результат выполнения программы представлен на рис. 3:



```
D:\cpp\Compilers\Compilers\cmilan\src>milanvm pr2.out
Reading input from pr2.out
100
101
102
103
104
105
```

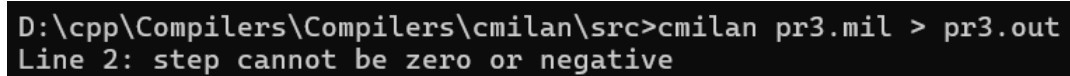
Рис. 3. Результат программы для примера №2

### Пример №3: отрицательный шаг

Исходный код программы:

```
1 BEGIN
2   x := [-2, 109..200];
3   write(x)
4 END
```

Результат выполнения программы представлен на рис. 4:



```
D:\cpp\Compilers\Compilers\cmilan\src>cmilan pr3.mil > pr3.out
Line 2: step cannot be zero or negative
```

Рис. 4. Результат программы для примера №3

### Пример №4: Полуоткрытый интервал справа

Исходный код программы:

```
1 BEGIN
2   x := [4, 20..4);
3   write(x)
4 END
```

ОП код:

```
1 0: PUSH 20
2 1: STORE 1
3 2: PUSH 16
4 3: STORE 2
5 4: PUSH 12
6 5: STORE 3
7 6: PUSH 8
8 7: STORE 4
9 8: LOAD 1
10 9: PRINT
11 10: LOAD 2
12 11: PRINT
13 12: LOAD 3
14 13: PRINT
15 14: LOAD 4
16 15: PRINT
17 16: STOP
```

Результат выполнения программы представлен на рис. 5:

```
D:\cpp\Compilers\Compilers\cmilan\src>milanvm pr4.out
Reading input from pr4.out
20
16
12
8
```

Рис. 5. Результат программы для примера №4

### Пример №5: задание значений интервала с помощью выражений

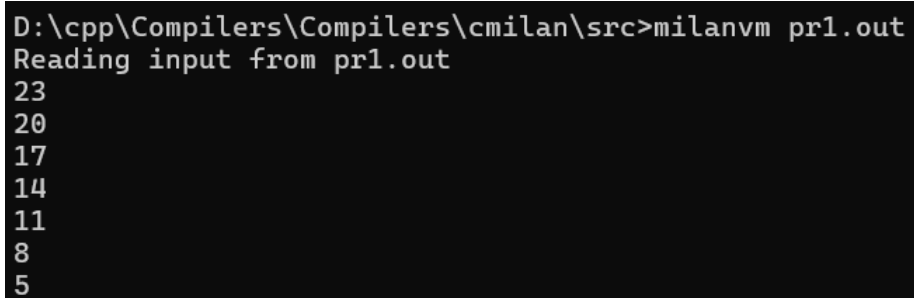
Исходный код программы:

```
1 BEGIN
2   x := [3,10*3-7..20/5];
3   write(x)
4 END
```

ОП код:

```
1 0: PUSH 23
2 1: STORE 1
3 2: PUSH 20
4 3: STORE 2
5 4: PUSH 17
6 5: STORE 3
7 6: PUSH 14
8 7: STORE 4
9 8: PUSH 11
10 9: STORE 5
11 10: PUSH 8
12 11: STORE 6
13 12: PUSH 5
14 13: STORE 7
15 14: LOAD 1
16 15: PRINT
17 16: LOAD 2
18 17: PRINT
19 18: LOAD 3
20 19: PRINT
21 20: LOAD 4
22 21: PRINT
23 22: LOAD 5
24 23: PRINT
25 24: LOAD 6
26 25: PRINT
27 26: LOAD 7
28 27: PRINT
```

Результат выполнения программы представлен на рис. 6:



```
D:\cpp\Compilers\Compilers\cmilan\src>milanvm pr1.out
Reading input from pr1.out
23
20
17
14
11
8
5
```

Рис. 6. Результат программы для примера №5

## Заключение

В ходе выполнения данной лабораторной работы компилятор CMilan был расширен поддержкой интервального типа данных. Milan представляет собой контекстно-свободный язык программирования, грамматика которого описывается формальными правилами в нотации БНФ. Были реализованы новые синтаксические конструкции, позволяющие объявлять и использовать интервалы в различных формах:  $[a..b]$ ,  $(a..b)$ ,  $[a,b..c]$ ,  $(a,b..c)$  и другие.

В язык добавлены новые токены — такие как `T_LBRACKET`, `T_RBRACKET`, `T_DOTDOT` и `T_КОММА`. Также была расширена грамматика языка Milan в БНФ-нотации и построены соответствующие синтаксические диаграммы для описания новых конструкций.

Метод `interval()` является основным и отвечает за корректное вычисление и сохранение элементов интервала в память программы. Для поддержки интервалов были модифицированы класс `Parser` и структура данных, хранящая информацию о массивах. При выводе через `WRITE` реализована автоматическая обработка переменных, представляющих интервалы, с выводом всех их элементов. Кроме того, были доработаны вспомогательные методы разбора арифметических выражений (`readSignedInt()`, `parseFactor()`, `parseTerm()`, `parseArithmeticExpression()` и `parsePrimary()`), обеспечившие корректную работу с выражениями внутри интервалов, включая унарные операции, скобки и переменные.

В компиляторе используется нисходящий синтаксический анализатор типа  $LL(1)$ , реализованный методом рекурсивного спуска, что соответствует структуре грамматики без левой рекурсии и с детерминированным выбором правил по первому токenu. Такая архитектура анализатора реализована в классе `Parser`, где каждому нетерминалу грамматики соответствует отдельный метод (например, `parseExpression()`, `parseStatement()`, `interval()` и другие). Это подтверждает, что расширенный язык Milan по-прежнему относится к классу контекстно-свободных языков.

Достоинства: реализованы все возможные виды интервалов. При использовании `WRITE(x)` автоматически выводятся все элементы переменной  $x$ , если она является интервалом — это упрощает работу с массивами. Логика разбора выражений разделена на модули, что улучшает читаемость кода и облегчает его дальнейшее развитие.

Недостатки: Интервалы реализованы как статические массивы, без возможности их изменения после создания. Интервалы хранятся в виде последовательности значений в памяти, что может быть расточительно при больших диапазонах.

Масштабирование: можно реализовать поддержку арифметических операций над интервальными типами данных (например, сложение, умножение интервалов или применение функций ко всем элементам) и добавить возможность индексации элементов интервала.

Исходный код был доработан на языке C++ с использованием компилятора Visual Studio 2022.

## Список литературы

- [1] Востров, А. В. Математическая логика  
URL:<https://tema.spbstu.ru/compiler> (Дата обращения: 20.05.2025).
- [2] Сети, Р.; Ахо, А. Компиляторы: принципы, технологии и инструменты / Р. Сети, А. Ахо. - М.: Издательство «Наука», 2006. - С. 104.