МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего 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»

Лабораторная работа №4
«Доработка компилятора языка MiLan»
по дисциплине
«Математическая логика и теория автоматов»
Вариант 8

Студент,
группы 5130201/20102 _____ Тищенко А. А.

Преподаватель _____ Востров А. В.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение

Лабораторная №4 заключается в доработке компилятора языка MiLan согласно варианту работы.

Вариант 8: В компилятор необходимо добавить поддержку операций инкремента и декремента, как постфиксных, так и префиксных: `++i`, `--i`, `i++`, `i--`. С точки зрения действия на операнд i между префиксными и постфиксными операциями разницы нет. Если эти выражения являются частью более сложного выражения, то при префиксной форме сначала изменяется операнд i, и уже измененный результат используется в выражении. При постфиксной форме операторов инкремента или декремента сначала операнд i используется в выражении, а потом уже изменяется.

1 Математическое описание

1.1 Обзор языка MiLan

Язык Милан — учебный язык программирования, описанный в учебнике [3].

Программа на Милане представляет собой последовательность операторов, заключенных между ключевыми словами `begin` и `end`. Операторы отделяются друг от друга точкой с запятой. После последнего оператора в блоке точка с запятой не ставится. Компилятор `CMilan` не учитывает регистр символов в именах переменных и ключевых словах.

В базовую версию языка Милан входят следующие конструкции: константы, идентификаторы, арифметические операции над целыми числами, операторы чтения чисел со стандартного ввода и печати чисел на стандартный вывод, оператор присваивания, условный оператор, оператор цикла с предусловием.

Программа может содержать комментарии, которые могут быть многострочными. Комментарий начинается символами `/*` и заканчивается символами `*/`. Вложенные комментарии не допускаются.

В данной лабораторной рассматривается добавление поддержки операций инкремента и декремента, как постфиксных, так и префиксных: `++i`, `--i`, `i++`, `i--`. С точки зрения действия на операнд i между префиксными и постфиксными операциями разницы нет. Если эти выражения являются частью более сложного выражения, то при префиксной форме сначала изменяется операнд i, и уже измененный результат используется в выражении. При постфиксной форме операторов инкремента или декремента сначала операнд i используется в выражении, а потом уже изменяется.

1.2 Лексический анализ

В реальных трансляторах языков программирования (ЯП) первой фазой является так называемый лексический анализ входной программы — предварительная обработка входного текста с выделением в нем структурно значимых единиц — лексем. На Рис. 1 представлена схема транслятора.

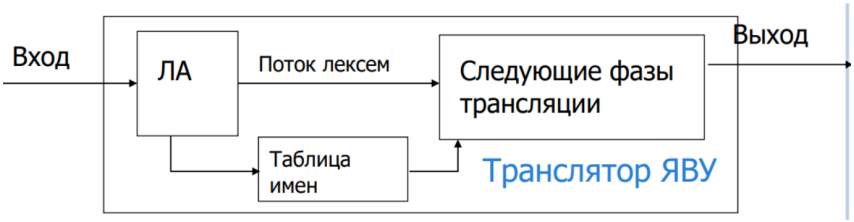

Рис. 1. Схема транслятора.

Лексемы — минимальные единицы языка, которые имеют смысл.

Значение лексемы, определяющее подстроку символов входной цепочки, соответствующих распознанному классу лексемы. В зависимости от класса, значение лексемы может быть преобразовано во внутреннее представление уже на этапе лексического анализа.

Класс лексемы, определяющий общее название для категории элементов, обладающих общими свойствами (идентификатор, целое число, строка символов...).

Лексический анализатор обрабатывает входную цепочку, а на его вход подаются символы, сгруппированные по категориям. Поэтому перед лексическим анализом осуществляется дополнительная обработка, сопоставляющая с каждым символом его класс, что позволяет сканеру манипулировать единым понятием для целой группы символов.

Лексический анализатор (лексер) — это конечный автомат, который преобразует входную строку символов в последовательность токенов. Формально его можно описать следующим образом:

Пусть заданы:

- Σ — входной алфавит (множество допустимых символов)
- T — множество типов токенов
- D — множество допустимых значений токенов

Тогда лексический анализатор реализует отображение:

$$F_{\text{lexer}} : \Sigma^* \to (T \times D)^*$$

где:

- Σ^* — множество всех возможных строк над алфавитом Σ
- $(T \times D)^*$ — множество последовательностей пар (тип токена, значение)

Процесс лексического анализа можно представить как **детерминированный конечный автомат (ДКА)**:

$$M = (Q, \Sigma, \delta, q_0, F),$$

где:

- Q — множество состояний автомата
- $\delta : Q \times \Sigma \to Q$ — функция переходов
- $q_0 \in Q$ — начальное состояние
- $F \subseteq Q$ — множество конечных состояний

Для каждого распознанного токена t_i выполняется:

$$t_i = (\text{type}, \text{value}), \quad \text{где type} \in T, \text{value} \in D$$

1.3 Синтаксический анализ

Синтаксический анализ — процесс сопоставления линейной последовательности лексем естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора. Обычно применяется совместно с лексическим анализом.

Синтаксический анализатор выражений (парсер) — часть программы, выполняющая чтение и анализ выражения.

Существует два типа алгоритмов синтаксического анализа: нисходящий и восходящий:

- Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
- Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.

Грамматика языка MiLan использует нисходящий парсер. При восстановлении синтаксического дерева при нисходящем разборе слева направо последовательно анализирует все поддеревья, принадлежащие самому левому нетерминалу. Когда самым левым становится другой нетерминал, анализируется уже он.

Компилятор CMilan включает три компонента (Рис. 2):

1. лексический анализатор;
2. синтаксический анализатор;
3. генератор команд виртуальной машины Милана.

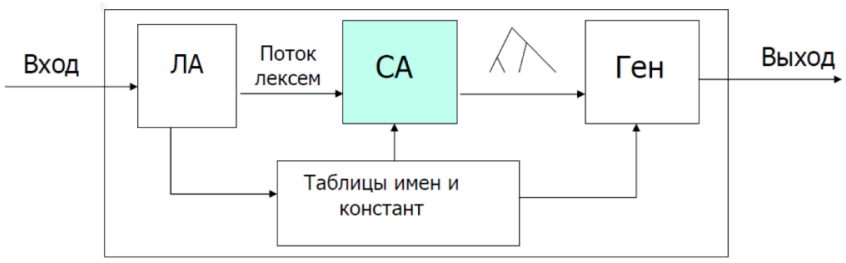

Рис. 2. Компоненты компилятора CMilan.

1.4 Грамматика языка MiLan

Грамматика языка Милан является контекстно-свободной, так как удовлетворяет определению КС-грамматики, т.е. продукции грамматики имеют вид $A \to \beta$, где A – одиночный нетерминал, а β – произвольная цепочка из терминалов и нетерминалов. Более того, грамматика языка Милан является LL(1) грамматикой, так как необходимо просмотреть поток всего на один символ вперед при принятии решения

о том, какое правило грамматики необходимо применить. В данной работе в грамматику были добавлены операции инкремента и декремента, однако это не повлияло на LL(1) свойство грамматики.

Грамматика языка Милан, расширенная операцией инкремента и декремента, в форме Бэкуса-Наура приведена ниже:

```
<program> ::= ‘begin’ <statementList> ‘end’
<statementList> ::= <statement> ‘;’ <statementList>
                  | epsilon
<statement> ::= <ident> ‘:=’ <expression>
              | ‘if’ <relation> ‘then’
                <statementList> [‘else’ <statementList>] ‘fi’
              | ‘while’ <relation> ‘do’ <statementList> ‘od’
              | ‘write’ ‘(’ <expression> ‘)’
<expression> ::= <term> <addop> <term>
<term> ::= <factor> <mulop> <factor>
<factor> ::= <ident>
           | <number>
           | ‘(’ <expression> ‘)’
           | ‘read’
           | ‘-’ <factor>
           | <ident> ‘++’
           | <ident> ‘--’
           | ‘++’ <ident>
           | ‘--’ <ident>
<relation> ::= <expression> <cmpi> <expression>
<addop> ::= ‘+’|‘-’
<mulop> ::= ‘*’|‘/’
<cmpi> ::= ‘=’|‘!=’|‘<’|‘<=’|‘>’|‘>=’
<number> ::= <digit> <digit>
<ident> ::= <letter> <letter> | <digit>
<letter> ::= ‘a’|‘b’|‘c’ | ...| ‘z’|‘A’|‘B’|‘C’ | ...| ‘Z’
<digit> ::= ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’
```

Изменения коснулись только правила `<factor>`, в котором были добавлены 4 новые продукции, описывающие операции постфиксного и префиксного инкремента и декремента.

Синтаксические диаграммы для всех нетерминалов грамматики приведены на Рис. 3 — Рис. 14. Обновлённая синтаксическая диаграмма для нетерминала `<factor>` приведена на Рис. 10.

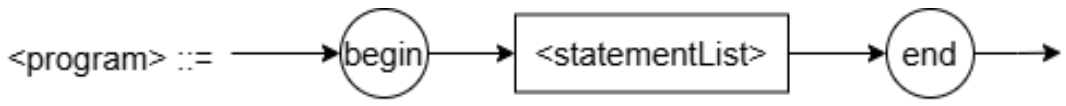

Рис. 3. Синтаксическая диаграмма для нетерминала `<program>`.

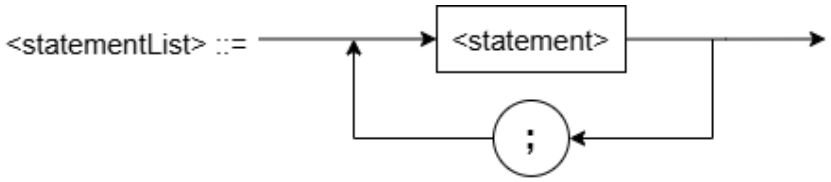

Рис. 4. Синтаксическая диаграмма для нетерминала `<statementList>`.

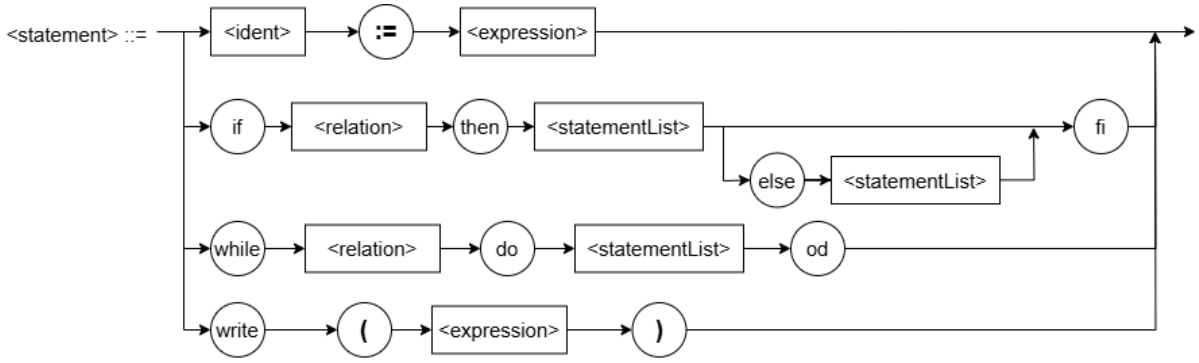

Рис. 5. Синтаксическая диаграмма для нетерминала `<statement>`.

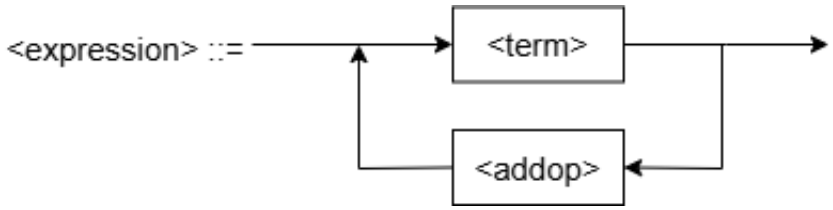

Рис. 6. Синтаксическая диаграмма для нетерминала `<expression>`.

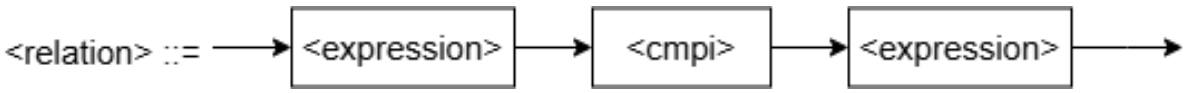

Рис. 7. Синтаксическая диаграмма для нетерминала `<relation>`.

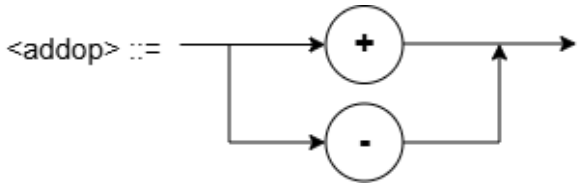

Рис. 8. Синтаксическая диаграмма для нетерминала `<addop>`.

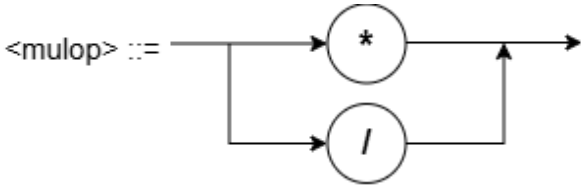

Рис. 9. Синтаксическая диаграмма для нетерминала `<mulop>`.

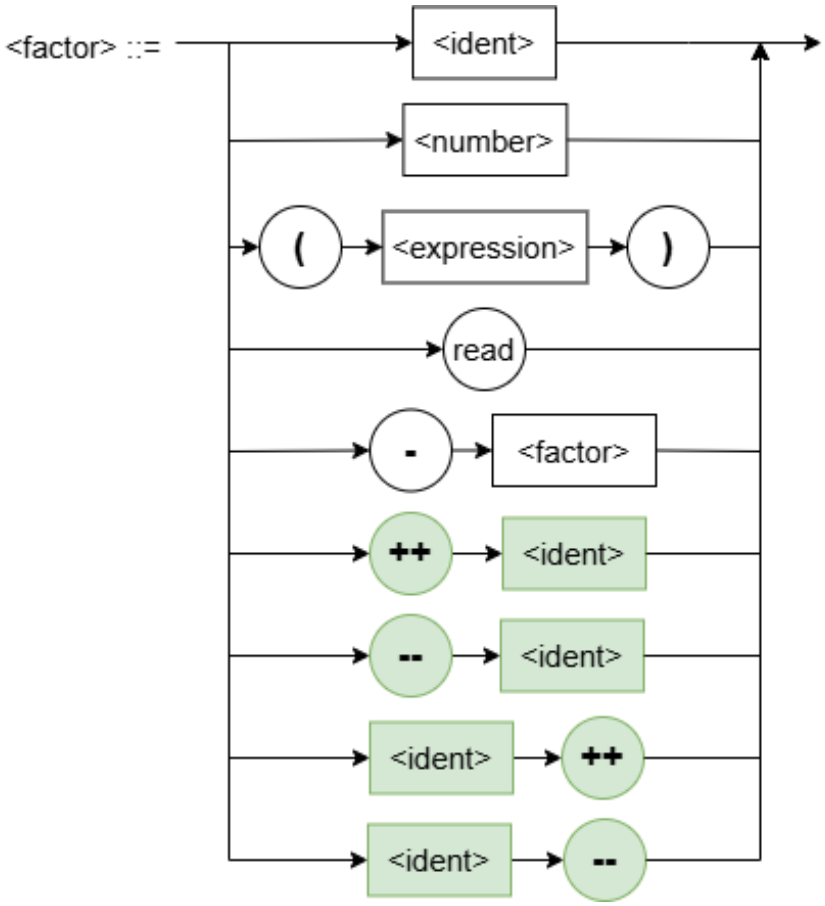

Рис. 10. Синтаксическая диаграмма для нетерминала `<factor>`, дополненная операциями инкремента и декремента (отмечены зеленым цветом).

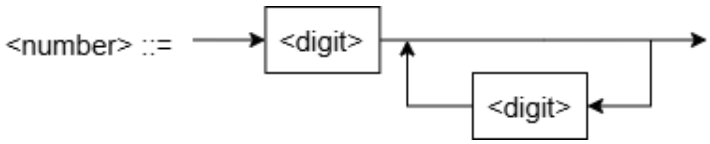

Рис. 11. Синтаксическая диаграмма для нетерминала `<number>`.

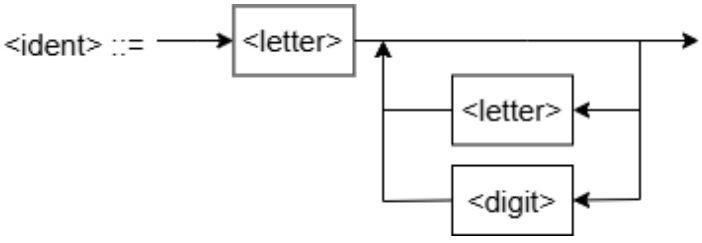

Рис. 12. Синтаксическая диаграмма для нетерминала `<ident>`.

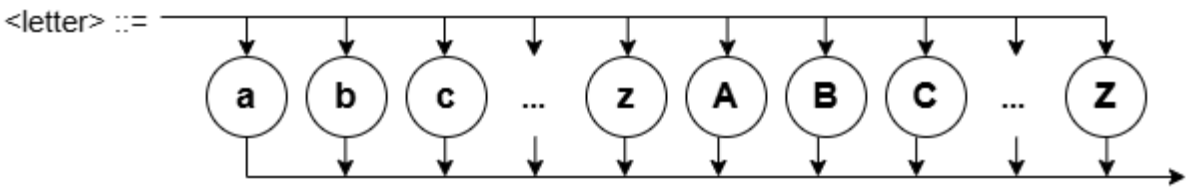

Рис. 13. Синтаксическая диаграмма для нетерминала `<letter>`.

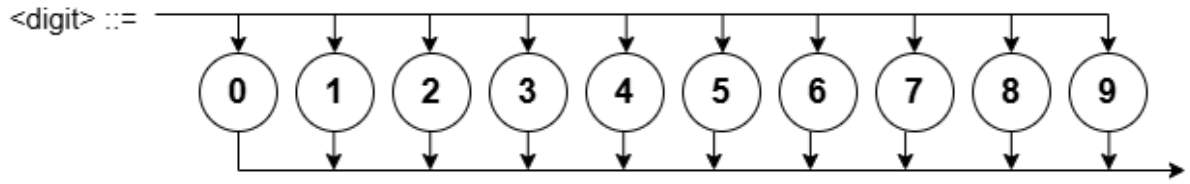

Рис. 14. Синтаксическая диаграмма для нетерминала `<digit>`.

2 Особенности реализации

2.1 Изменения в лексическом анализаторе

2.1.1 Файл `Scanner.h`

В перечисление `Token` в файле `Scanner.h` были добавлены новые токены: `INC`, `DEC`. Код обновлённого перечисления представлен в листинге 1, добавлены строки 24-25.

Листинг 1. Обновлённое перечисление `Token`

```
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,    // Сводная лексема для "+" и "-" операция( типа сложения)
19     T_MULOP,    // Сводная лексема для "*" и "/" операция( типа умножения)
20     T_CMP,      // Сводная лексема для операторов отношения
21     T_LPAREN,   // Открывающая скобка
22     T_RPAREN,   // Закрывающая скобка
23     T_SEMICOLON,  // ";"
24     T_INC,      // Оператор инкремента
25     T_DEC     // Оператор декремента
26 };
```

2.1.2 Файл `Scanner.cpp`

В массив названий токенов `tokenNames_` были добавлены новые строки, соответствующие инкременту и декременту: `"++"` и `"--"`, соответственно. Код обновлённого массива представлен в листинге 2, добавлены строки 24-25.

Листинг 2. Обновлённый массив `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    };
```

Также была обновлена функция `nextToken()`. В ней были добавлены новые условия для распознавания инкремента и декремента. Код обновлённой функции представлен в листинге 3, изменения коснулись строк 158-181.

Листинг 3. Обновлённая функция `nextToken()`

```
1  void Scanner::nextToken()
2  {
3    skipSpace();
4
5    // Пропускаем комментарии
6    // Если встречаем "/", то за ним должна идти "*". Если "*" не встречена, считаем, что
       встретили операцию деления
7    // и лексему - операция типа умножения. Дальше смотрим все символы, пока не находим
       звездочку или символ конца файла.
8    // Если нашли * - проверяем на наличие "/" после нее. Если "/" не найден - ищем
       следующую "*".
9    while(ch_ == '/') {
10     nextChar();
11     if(ch_ == '*') {
12       nextChar();
13       bool inside = true;
14       while(inside) {
15         while(ch_ != '*' && !input_.eof()) {
16           nextChar();
17         }
18
19         if(input_.eof()) {
20           token_ = T_EOF;
21           return;
22         }
23
```

```
24          nextChar();
25          if(ch_ == '/') {
26            inside = false;
27            nextChar();
28          }
29        }
30      }
31      else {
32        token_ = T_MULOP;
33        arithmeticValue_ = A_DIVIDE;
34        return;
35      }
36
37      skipSpace();
38    }
39
40    //Если встречен конец файла, считаем за лексему конца файла.
41    if(input_.eof()) {
42      token_ = T_EOF;
43      return;
44    }
45    //Если встретили цифру, то до тех пока дальше идут цифры – считаем как продолжение
        числа.
46    //Запоминаем полученное целое, а за лексему считаем целочисленный литерал
47
48    if(isdigit(ch_)) {
49      int value = 0;
50      while(isdigit(ch_)) {
51        value = value * 10 + (ch_ - '0'); //поразрядное считывание, преобразуем символьное
          значение к числу.
52        nextChar();
53      }
54      token_ = T_NUMBER;
55      intValue_ = value;
56    }
57    //Если же следующий символ – буква ЛА – тогда считываем до тех пор, пока дальше буквы
        ЛА или цифры.
58    //Как только считали имя переменной, сравниваем ее со списком зарезервированных слов.
        Если не совпадает ни с одним из них,
59    //считаем, что получили переменную, имя которой запоминаем, а за текущую лексему
        считаем лексему идентификатора.
60    //Если совпадает с какимлибо– словом из списка – считаем что получили лексему,
        соответствующую этому слову.
61    else if(isIdentifierStart(ch_)) {
62      string buffer;
63      while(isIdentifierBody(ch_)) {
64        buffer += ch_;
65        nextChar();
66      }
67
68      transform(buffer.begin(), buffer.end(), buffer.begin(), ::tolower);
69
70      map<string, Token>::iterator kwd = keywords_.find(buffer);
71      if(kwd == keywords_.end()) {
72        token_ = T_IDENTIFIER;
73        stringValue_ = buffer;
```

```
74      }
75      else {
76        token_ = kwd->second;
77      }
78    }
79    //Символ не является буквой, цифрой, "/" или признаком конца файла
80    else {
81      switch(ch_) {
82        //Признак лексемы открывающей скобки – встретили "("
83        case '(':
84          token_ = T_LPAREN;
85          nextChar();
86          break;
87        //Признак лексемы закрывающей скобки – встретили ")"
88        case ')':
89          token_ = T_RPAREN;
90          nextChar();
91          break;
92        //Признак лексемы ";" – встретили ";"
93        case ';':
94          token_ = T_SEMICOLON;
95          nextChar();
96          break;
97        //Если встречаем ":", то дальше смотрим наличие символа "=". Если находим, то считаем
          что нашли лексему присваивания
98        //Иначе – лексема ошибки.
99        case ':':
100         nextChar();
101         if(ch_ == '=') {
102           token_ = T_ASSIGN;
103           nextChar();
104
105         }
106         else {
107           token_ = T_ILLEGAL;
108         }
109         break;
110       //Если встретили символ "<", то либо следующий символ "=", тогда лексема нестрогого
          сравнения. Иначе – строгого.
111       case '<':
112         token_ = T_CMP;
113         nextChar();
114         if(ch_ == '=') {
115           cmpValue_ = C_LE;
116           nextChar();
117         }
118         else {
119           cmpValue_ = C_LT;
120         }
121         break;
122       //Аналогично предыдущему случаю
123       case '>':
124         token_ = T_CMP;
125         nextChar();
126         if(ch_ == '=') {
127           cmpValue_ = C_GE;
```

```
128           nextChar();
129         }
130         else {
131           cmpValue_ = C_GT;
132         }
133         break;
134       //Если встретим "!", то дальше должно быть "=", тогда считаем, что получили лексему
          сравнения
135       //и знак "!=" иначе считаем, что у нас лексема ошибки
136       case '!':
137         nextChar();
138         if(ch_ == '=') {
139           nextChar();
140           token_ = T_CMP;
141           cmpValue_ = C_NE;
142         }
143         else {
144           token_ = T_ILLEGAL;
145         }
146         break;
147       //Если встретим "=" - лексема сравнения и знак "="
148       case '=':
149         token_ = T_CMP;
150         cmpValue_ = C_EQ;
151         nextChar();
152         break;
153       //Знаки операций. Для "+"/"-" получим лексему операции типа сложнения, и
          соответствующую операцию.
154       //для "*" - лексему операции типа умножения
155       case '+':
156         nextChar();
157
158               // Ищем оператор инкремента
159         if(ch_ == '+') {
160           token_ = T_INC;
161           nextChar();
162         }
163         else {
164           token_ = T_ADDOP;
165           arithmeticValue_ = A_PLUS;
166         }
167         break;
168
169       case '-':
170         nextChar();
171
172               // Ищем оператор декремента
173         if(ch_ == '-') {
174           token_ = T_DEC;
175           nextChar();
176         }
177         else {
178           token_ = T_ADDOP;
179           arithmeticValue_ = A_MINUS;
180         }
181         break;
```

```
182
183       case '*':
184         token_ = T_MULOP;
185         arithmeticValue_ = A_MULTIPLY;
186         nextChar();
187         break;
188       //Иначе лексема ошибки.
189       default:
190         token_ = T_ILLEGAL;
191         nextChar();
192         break;
193     }
194   }
195 }
```

2.2 Изменения в компиляторе

2.2.1 Файл `Parser.cpp`

В метод `factor()` объекта `Parser` была добавлена логика генерации команд для префиксного и постфиксного инкремента и декремента. Код обновлённого метода представлен в листинге 4, изменения коснулись строк 20-27 (постфиксный инкремент и декремент) и строк 29-41 (префиксный инкремент и декремент).

Для постфиксного инкремента и декремента генерируется следующая последовательность команд виртуальной машины языка MiLan:

- `LOAD <адрес переменной>` - загрузка значения переменной на вершину стека.
- `DUP` - дублирование значения на вершине стека до выполнения операции инкремента или декремента, так как постфиксная операция возвращает старое значение переменной.
- `PUSH 1` - загрузка константы 1 на вершину стека
- `ADD` или `SUB` - сложение или вычитание значения на вершине стека с константой 1.
- `STORE <адрес переменной>` - сохранение результата на вершине стека по адресу переменной.

Для префиксного инкремента и декремента генерируется следующая последовательность команд виртуальной машины языка MiLan:

- `LOAD <адрес переменной>` - загрузка значения переменной на вершину стека.
- `PUSH 1` - загрузка константы 1 на вершину стека
- `ADD` или `SUB` - сложение или вычитание значения на вершине стека с константой 1.

- `DUP` - дублирование значения на вершине стека после выполнения операции инкремента или декремента, так как префиксная операция возвращает новое значение переменной.

- `STORE <адрес переменной>` - сохранение результата на вершине стека по адресу переменной.

Листинг 4. Обновлённый метод `factor()`

```
1  void Parser::factor()
2  {
3      /*
4          Множитель описывается следующими правилами:
5          <factor> -> number | identifier | -<factor> | (<expression>) | READ
6          | ++ identifier | -- identifier | identifier++ | identifier--
7      */
8      if(see(T_NUMBER)) {
9          int value = scanner_->getIntValue();
10         next();
11         codegen_->emit(PUSH, value);
12         //Если встретили число, то преобразуем его в целое и записываем на вершину стека
13     }
14     else if(see(T_IDENTIFIER)) {
15         int varAddress = findOrAddVariable(scanner_->getStringValue());
16         next();
17         codegen_->emit(LOAD, varAddress);
18         //Если встретили переменную, то выгружаем значение, лежащее по ее адресу, на вершину
       стека
19
20         // Постфиксный инкремент или декремент
21         if(see(T_INC) || see(T_DEC)) {
22             codegen_->emit(DUP);
23             codegen_->emit(PUSH, 1);
24             codegen_->emit(see(T_INC) ? ADD : SUB);
25             codegen_->emit(STORE, varAddress);
26             next();
27         }
28     }
29     // Префиксный инкремент или декремент
30     else if(see(T_INC) || see(T_DEC)) {
31         bool isIncrement = see(T_INC);
32         next();
33         mustBe(T_IDENTIFIER);
34         int varAddress = findOrAddVariable(scanner_->getStringValue());
35
36         codegen_->emit(LOAD, varAddress);
37         codegen_->emit(PUSH, 1);
38         codegen_->emit(isIncrement ? ADD : SUB);
39         codegen_->emit(DUP);
40         codegen_->emit(STORE, varAddress);
41     }
42     else if(see(T_ADDOP) && scanner_->getArithmeticValue() == A_MINUS) {
43         next();
44         factor();
45         codegen_->emit(INVERT);
```

```
46          //Если встретили знак "−", и за ним <factor> то инвертируем значение, лежащее на
        вершине стека
47      }
48      else if(match(T_LPAREN)) {
49          expression();
50          mustBe(T_RPAREN);
51          //Если встретили открывающую скобку, тогда следом может идти любое арифметическое
        выражение и обязательно
52          //закрывающая скобка.
53      }
54      else if(match(T_READ)) {
55          codegen_->emit(INPUT);
56          //Если встретили зарезервированное слово READ, то записываем на вершину стека идет
        запись со стандартного ввода
57      }
58      else {
59          reportError("expression expected.");
60      }
61  }
```

3 Результаты работы программы

Программа №1

Исходный код программы представлен в листинге 5.

Листинг 5. Исходный код программы №1

```
1  BEGIN
2      x := 0;
3      write(x++);
4      write(x);
5      write(++x);
6      write(x)
7  END
```

Последовательность команд, сгенерированная компилятором для данной программы, представлена в листинге 6.

Листинг 6. Последовательность команд, сгенерированная компилятором для программы №1.

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

Результаты работы виртуальной машины для программы №1 представлены на Рис. 15.

```
Reading input from compiled.cmil
0
1
2
2
```

Рис. 15. Результаты работы виртуальной машины для программы №1.

Программа №2

Исходный код программы представлен в листинге 7.

Листинг 7. Исходный код программы №2

```
1  BEGIN
2      i := 1;
3      j := 2;
4
5      IF i < --j THEN WRITE(100) ELSE WRITE(-100) FI
6  END
```

Последовательность команд, сгенерированная компилятором для данной программы, представлена в листинге 8.

Листинг 8. Последовательность команд, сгенерированная компилятором для программы №2.

```
0:  PUSH  1
1:  STORE 0
2:  PUSH  2
3:  STORE 1
4:  LOAD  0
5:  LOAD  1
6:  PUSH  1
7:  SUB
8:  DUP
9:  STORE 1
10: COMPARE 2
11: JUMP_NO 15
12: PUSH  100
13: PRINT
14: JUMP  18
15: PUSH  100
16: INVERT
17: PRINT
18: STOP
```

Результаты работы виртуальной машины для программы №2 представлены на Рис. 16.

```
Reading input from compiled.cmil
-100
```

Рис. 16. Результаты работы виртуальной машины для программы №2.

Программа №3

Исходный код программы представлен в листинге 9.

Листинг 9. Исходный код программы №3

```
1      BEGIN
```

```
2     y := x++;
3     write(y);
4     write(x);
5
6     y := 10 - --x;
7     write(y);
8     write(x);
9
10    y := 10 -++x;
11    write(y);
12    write(x)
13  END
```

Последовательность команд, сгенерированная компилятором для данной программы, представлена в листинге 10.

Листинг 10. Последовательность команд, сгенерированная компилятором для программы №3.

```
0:  LOAD  1
1:  DUP
2:  PUSH  1
3:  ADD
4:  STORE 1
5:  STORE 0
6:  LOAD  0
7:  PRINT
8:  LOAD  1
9:  PRINT
10: PUSH  10
11: LOAD  1
12: PUSH  1
13: SUB
14: DUP
15: STORE 1
16: SUB
17: STORE 0
18: LOAD  0
19: PRINT
20: LOAD  1
21: PRINT
22: PUSH  10
23: LOAD  1
24: PUSH  1
25: ADD
26: DUP
27: STORE 1
28: SUB
29: STORE 0
30: LOAD  0
31: PRINT
32: LOAD  1
33: PRINT
34: STOP
```

Результаты работы виртуальной машины для программы №3 представлены на Рис. 17.

```
Reading input from compiled.cmil
0
1
10
0
9
1
```

Рис. 17. Результаты работы виртуальной машины для программы №3.

Программа №4

Исходный код программы представлен в листинге 11.

Листинг 11. Исходный код программы №4

```
1    BEGIN
2    x := 10;
3    y := 10 - --x; /* Корректно: минус и декремент разделены пробелом */
4    y := 10 ---x; /* Некорректно: три минуса подряд */
5    END
```

Результат запуска компилятора для данной программы представлен на Рис. 18.

```
Line 4: '--' found while 'END' expected.
```

Рис. 18. Результат запуска компилятора для программы №4.

Программа №5

Исходный код программы представлен в листинге 12.

Листинг 12. Исходный код программы №5

```
1  BEGIN
2    x := --x++
3  END
```

Результат запуска компилятора для данной программы представлен на Рис. 19.

```
Line 2: '++' found while 'END' expected.
```

Рис. 19. Результат запуска компилятора для программы №5.

Заключение

В ходе выполнения лабораторной работы была успешно реализована поддержка операций инкремента и декремента в компиляторе языка MiLan. Были добавлены как префиксные (`++i`, `--i`), так и постфиксные (`i++`, `i--`) операторы с корректной семантикой их выполнения. Модификации затронули как лексический анализатор (добавление новых токенов `T_INC` и `T_DEC`), так и синтаксический анализатор, использующий метод рекурсивного спуска (расширение метода `factor()` для обработки новых конструкций).

Расширенная грамматика языка MiLan осталась контекстно-свободной и сохранила свойство LL(1), что позволило избежать значительных изменений в существующей архитектуре компилятора. Было добавлено лишь несколько новых правил в определение нетерминала `<factor>`.

Из достоинств реализации можно отметить минимальность вносимых изменений в существующую архитектуру компилятора и сохранение всех ранее реализованных функций. При этом реализация выполняет поставленные задачи, корректно обрабатывая возможные случаи использования операторов инкремента и декремента.

К недостаткам текущей реализации можно отнести отсутствие каких-либо оптимизаций при генерации команд виртуальной машины, что может приводить к избыточному количеству инструкций. Также в коде наблюдается некоторое дублирование логики между обработкой инкремента и декремента.

В качестве направлений масштабирования можно предложить добавление составных операторов присваивания (`+=`, `-=`, `*=`, `/=`), для которых генерируется схожая последовательность низкоуровневых команд. Также возможна реализация оптимизаций генерируемого кода, таких как устранение избыточных команд в простых случаях.

На выполнение лабораторной работы ушло около 6 часов. Работа была выполнена в среде разработки Visual Studio Code.

Список литературы

[1] Востров, А.В. Курс лекций по дисциплине «Математическая логика». URL https://tema.spbstu.ru/compiler/ (дата обращения 01.05.2025 г.)

[2] А. Ахо, М. Лам, Р. Сети, Дж. Ульман, Компиляторы: принципы, технологии и инструментарий, 2-е изд. М.: Вильямс, 2011.

[3] Ю.Г. Карпов, Теория и технология программирования. Основы построения трансляторов. СПб.: БХВ-Петербург, 2005.