

МИНОБРНАУКИ 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

Лабораторная работа №4

По дисциплине:

«Математическая логика и теория автоматов»

Тема работы:

«Доработка комплятора языка Milan»

«Перечислимый тип. Операции над данными перечислимого типа.»

Обучающийся: \_\_\_\_\_

Черепанов Михаил Дмитриевич

Руководитель: \_\_\_\_\_

Востров Алексей Владимирович

«\_\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

Санкт-Петербург 2024

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Описание языка Milan</b>	<b>4</b>
<b>2 Математическое описание</b>	<b>5</b>
2.1 Контекстно-свободные грамматики . . . . .	5
2.2 LL(k) грамматики . . . . .	5
2.3 Доработка грамматики языка Милан . . . . .	5
2.4 Синтаксические диаграммы грамматики языка Milan . . . . .	6
<b>3 Особенности реализации</b>	<b>9</b>
3.1 Доработка лексического анализатора . . . . .	9
3.2 Доработка синтаксического анализатора . . . . .	9
<b>4 Результаты работы</b>	<b>12</b>
<b>Заключение</b>	<b>16</b>
<b>Список использованных источников</b>	<b>17</b>

# Введение

В рамках данной работы необходимо:

Доработать компилятор языка Milan в соответствии с выбранной темой:

Тема:

Перечислимый тип. Операции над данными перечислимого типа.

# 1 Описание языка Milan

Язык Милан - учебный язык программирования. 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[1].

## Описание компилятора `CMilan`

В предоставленную реализацию компилятора `CMilan` включены следующие компоненты[1]:

- генератор кода для виртуальной машины `Milan`;
- лексический анализатор;
- синтаксический анализатор.

Лексический анализатор - посимвольно читает из входного потока текст программы и преобразует группы символов в лексемы (терминальные символы грамматики языка Милан). При этом он отслеживает номер текущей строки, который используется синтаксическим анализатором при формировании сообщений об ошибках.

Синтаксический анализатор - читает сформированную лексическим анализатором последовательность лексем и проверяет ее соответствие грамматике языка Милан. Для этого используется метод рекурсивного спуска. Если в процессе синтаксического анализа обнаруживается ошибка, анализатор формирует сообщение об ошибке, включающее сведения об ошибке и номер строки, в которой она возникла.

Генератор кода - представляет собой служебный компонент, ответственный за формирование внешнего представления генерируемого кода[1].

Грамматика языка `Milan` является грамматикой `LL(1)`, подклассом контекстно-свободных грамматик.

## 2 Математическое описание

### 2.1 Контекстно-свободные грамматики

Грамматикой  $G$  называется множество:  $\{T, N, S, R\}$ , где:

- $T$  - конечное множество символов (терминальный словарь);
- $N$  - конечное множество символов (нетерминальный словарь);
- $S \in N$  - начальный нетерминал;
- $R$  - конечное множество правил (продукций) вида  $\alpha \rightarrow \beta$ , где  $\alpha$  и  $\beta$  цепочки

над словарём  $T \cup N$

Контекстно-свободной грамматикой называется грамматика, у которой левые части всех продукций являются одиночными нетерминалами. То есть все правила грамматики имеют вид:

$$A \rightarrow \lambda, \text{ где } A \in T, \lambda \in N$$

### 2.2 LL(k) грамматики

LL(k) грамматики - подкласс контекстно-свободных грамматик, в котором можно выполнить нисходящий синтаксический анализ, просматривая входную цепочку слева при восстановлении левого вывода данной терминальной цепочки, заглядывая вперед на каждом шаге не более чем на  $k$  символов.

Для обозначения LL(K) грамматик введем функции FIRST и FOLLOW:

FIRST(A) - множество терминальных символов, с которых могут начинаться цепочки, выводимые из A.

FOLLOW(A) - множество терминальных символов, с которых могут начинаться цепочки, следующие за A в любых сентенциальных формах.

Формально грамматика является LL(k) если:

$$S \rightarrow *wAx \rightarrow w\alpha x \rightarrow *wu$$

$$S \rightarrow *wAx \rightarrow w\beta x \rightarrow *wv, \text{ где:}$$

$S, A$  - нетерминалы грамматики;

$w$  - цепочка из терминалов;

$\alpha, \beta, x$  - цепочки из терминалов и нетерминалов.

$A$  — нетерминал грамматики, для которого есть правила  $A \rightarrow \alpha$  и  $A \rightarrow \beta$ ;

Из условия  $FIRST_k(u)$  следует условие  $FIRST_k(v)$ .

LL(1) - подмножество LL(K) грамматик, в которых можно выполнить нисходящий синтаксический анализ, просматривая не более 1 символа вперед на каждом шаге.

### 2.3 Доработка грамматики языка Милан

Грамматика языка Милан является контекстно-свободной LL(1) грамматикой.

Для реализации перечислимого типа были добавлены следующие символы:

Нетерминалы:

$\langle \text{enumeration} \rangle$  - обозначает общий блок перечисления

<item> - обозначает один элемент перечисления.

Терминалы:

«enum» - ключевое слово для обозначения блока перечисления.

В качестве обозначения начала и конца перечисления используются круглые скобки, уже присутствующие в грамматике.

Дополненная грамматика приведена на рис.1. Добавленные строки выделены цветом.

```
<program> ::= 'begin' <statementList> 'end'

<statementList> ::= <statement> ';' <statementList> | ε

<statement> ::= <ident> ':' <expression>
               | 'if' <relation> 'then' <statementList> 'else' <statementList> 'fi'
               | 'while' <relation> 'do' <statementList> 'od'
               | 'write' '(' <expression> ')'
               | 'enum' '(' <enumeration> ')'

<enumeration> ::= <item> , <enumeration> | <item> | ε

<item> ::= <ident> | <ident> ':' <number>

<expression> ::= <term> { <addop> <term> }

<term> ::= <factor> { <mulop> <factor> }

<factor> ::= <ident> | <number> | '(' <expression> ')'

<relation> ::= <expression> <cmp> <expression>

<addop> ::= '+' | '-'

<mulop> ::= '*' | '/'

<cmp> ::= '=' | '!=' | '<' | '<=' | '>' | '>='

<ident> ::= <letter> { <letter> | <digit> }

<letter> ::= 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | 'C' | ... | 'Z'

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Рис. 1: Доработанная грамматика языка Милан в БНФ.

## 2.4 Синтаксические диаграммы грамматики языка Milan

На рис. 2 - рис. 3 изображены синтаксические диаграммы грамматики языка Milan с дополнениями

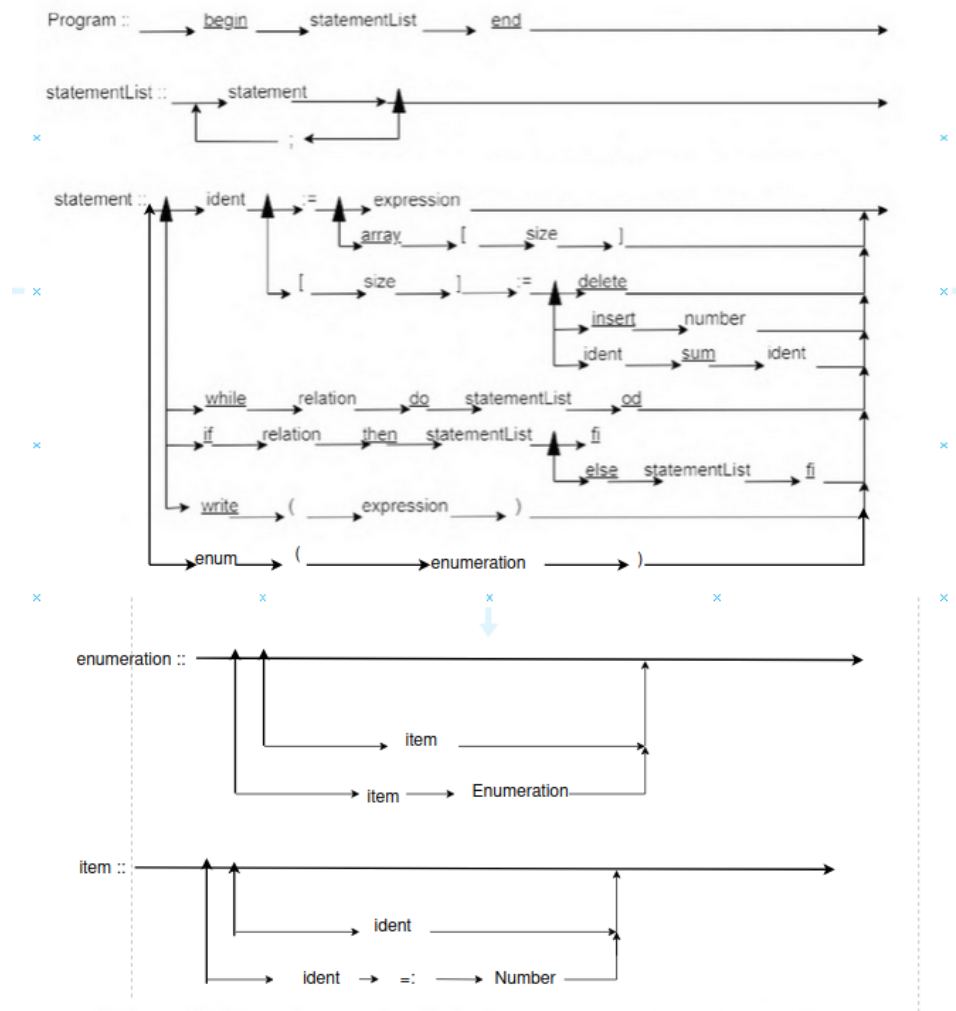


Рис. 2: Синтаксические диаграммы. Элемент №1.

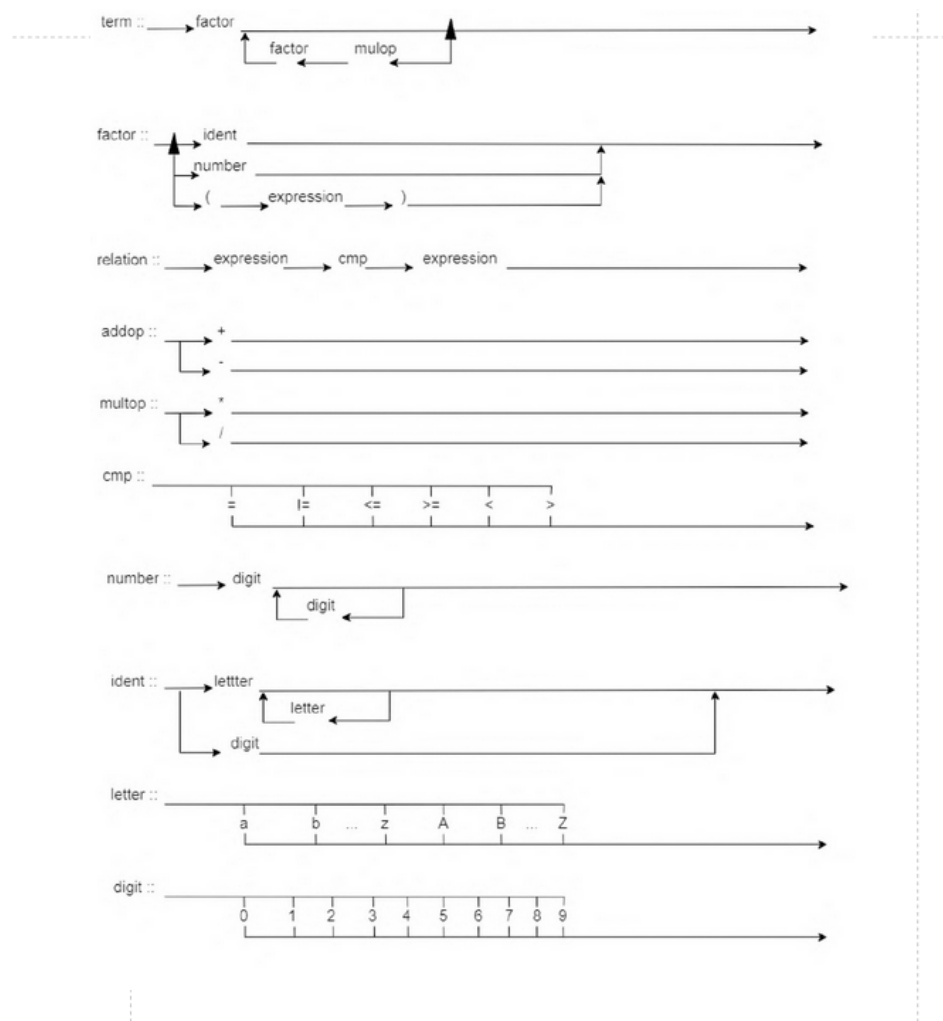


Рис. 3: Синтаксические диаграммы. Элемент №2.



## 3 Особенности реализации

Для доработки компилятора для возможности использовать перечислимый тип, потребовалось дополнить классы scanner(Лексический анализатор) и parser(Синтаксический анализатор).

### 3.1 Доработка лексического анализатора

Доработка лексического анализатора заключалась в добавлении ключевого слова "ENUM" в список лексем в массиве строк tokenNames.

Итоговый список токенов представлен ниже:

```
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     "ENUM",
18     "':=' ",
19     "'+' or '-' ",
20     "'*' or '/' ",
21     "comparison operator",
22     "'(' ",
23     "')' ",
24     "';' ",
25     "',' ",
26 };
```

### 3.2 Доработка синтаксического анализатора

Доработка синтаксического анализатора заключалась в добавлении функции enumeration() для обработки блока перечисления.

**Вход функции:** последовательность токенов;

**Выход функции:** команды Р-кода или сообщения об ошибке, в случае некорректной последовательности токенов.

В теле функции реализован метод рекурсивного спуска в соответствии с грамматикой, показанной на рис.1 для продукции enumeration.

Эффектами функции является запись команд Р-кода в буфер вывода класса codegen. В случае некорректности последовательности, сообщение об ошибке записывается в буфер вывода ошибок.

Исходный код функции enumeration:

```
1 void Parser::enumeration() {
2     int currentValEnum=0;
3     //codegen_->emit(PUSH, currentValEnum);
4
5     bool more = 1;
6     bool flagFirstIteration=1;
7     while (more) {
8         if(see(T_IDENTIFIER)){
9             string varName=scanner_->getStringValue();
10            if(checkAlreadyInitialized(varName)) {
11                reportError("Variable already exist.");
12            }
13            ↪ // если такой переменной еще нет
14            int varAddress = AddVariableWithoutChecking(varName);
15            ↪ // сохранили переменную, получили адрес
16            next();
17            if (match(T_ASSIGN)) {
18                expression();
19                codegen_->emit(STORE, varAddress);
20            }
21            else{
22                if(flagFirstIteration){
23                    codegen_->emit(PUSH, currentValEnum);
24                    codegen_->emit(STORE, varAddress);
25                }
26                else{
27                    codegen_->emit(LOAD, varAddress-1);
28                    codegen_->emit(PUSH, 1);
29                    codegen_->emit(ADD);
30                    codegen_->emit(STORE, varAddress);
31                }
32            }
33        }
34        else{
35            reportError("Identifier expected.");
36            while(!see(T_COMMA) && !see(T_RPAREN) && !see(T_END))
37                next();
38        }
39    }
40 }
```

```
37         more = match(T_COMMA);
38         flagFirstIteration=0;
39     }
40
41
42 }
43
```

Функция `enumeration` вызывается из функции `statement` в соответствии с доработанной грамматикой.

## 4 Результаты работы

### Пример№1

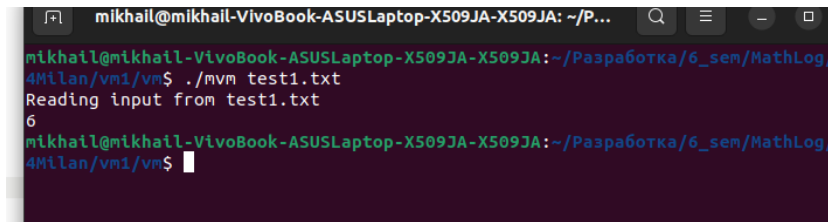
Код на языке Milan:

```
1 begin
2  enum(
3    a:=1,
4    b,
5    c);
6  res := a+b+c;
7  write(res)
8  end
```

Сгенерированный Р-код и результат выполнения на виртуальной машине представлен на рис. 4 и 5 соответственно.

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

Рис. 4: Р-code примера №1.



```
mikhail@mikhail-VivoBook-ASUSLaptop-X509JA-X509JA: ~/P...
mikhail@mikhail-VivoBook-ASUSLaptop-X509JA-X509JA:~/Разработка/6_sem/MathLog/
4Milan/vm1/vm$ ./mvm test1.txt
Reading input from test1.txt
6
mikhail@mikhail-VivoBook-ASUSLaptop-X509JA-X509JA:~/Разработка/6_sem/MathLog/
4Milan/vm1/vm$
```

Рис. 5: Результат выполнения программы на виртуальной машине.

## Пример№2

Код на языке Milan:

```
1 begin
2 enum(
3 a1,
4 a2,
5 a3
6 );
7
8 enum(
9 b1,
10 b2,
11 b3
12 );
13 b2:= (a2+a3+b3)*b2;
14 write(b2)
15 end
```

Сгенерированный P-код и результат выполнения на виртуальной машине представлен на рис. 6 и 7 соответственно.

```

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

```

Рис. 6: P-code примера №2.

```

mikhail@mikhail-VivoBook-ASUSLaptop-X509JA-X509JA:~/Pa
4Milan/vm1/vm$ ./mvm test1.txt
Reading input from test1.txt
5
mikhail@mikhail-VivoBook-ASUSLaptop-X509JA-X509JA:~/Pa

```

Рис. 7: Результат выполнения программы на виртуальной машине.

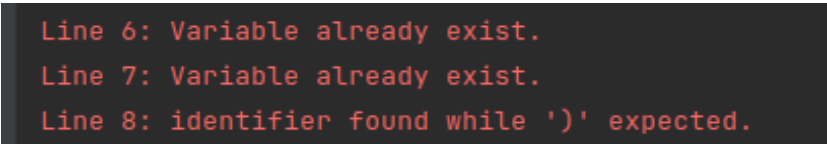
### Пример№3 (Ошибка компиляции)

Код на языке Milan:

```
1 begin
2  a:=1;
3  a1:=2;
4  enum(
5  b:=1,
6  a:=2,
7  a1
8  a2
9  );
10 write(a)
11 end
```

В данном примере в блоке `enum` присутствуют повторно объявленные переменные и отсутствует запятая между идентификаторами.

Пример сообщения об ошибках представлен на рис.8.



```
Line 6: Variable already exist.
Line 7: Variable already exist.
Line 8: identifier found while ')' expected.
```

Рис. 8: Сообщение об ошибках.

## Заключение

В результате работы был доработан компилятор SMilan языка Milan добавлением поддержки перечислимого типа и арифметических операций над ним.

Перечислимый тип задается с помощью ключевого слова `enum` и ограничивается круглыми скобками. Элементы внутри перечислимого типа отделяются запятыми.

Для каждого из элементов перечислимого типа есть возможность задать значение. Если значение не задано, элементу присвоится значение предыдущего идентификатора, увеличенного на один (Если элемент - первый в списке, присвоится значение 1).

Над данными перечислимого типа возможны все арифметические операции, предусмотренные базовой реализацией языка Milan (+, -, /, \*).

### **Преимущества реализации**

1. Использование метода рекурсивного спуска;
2. Использование круглых скобок, уже являющихся частью языка Milan в качестве обозначения начала и конца блока перечислений.

### **Недостатки реализации**

1. Высокая вложенность в функции `enumerate`.

### **Масштабирование**

Программу можно масштабировать путем добавления классов перечислений (`enum class`), что позволяет отделить переменные перечисляемого типа от других переменных в коде программы.



## **Список использованных источников**

- [1] Аллахвердиев Э.Ф., Тимофеев Д.А. Компилятор CМlап.
- [2] А.В. Востров. Лекция №12 по дисциплине «Математическая логика и теория автоматов». 2024г. 55 сл.