

## 1 ЦЕЛЬ РАБОТЫ

Получение навыка проектировать и реализовывать собственный формальный язык.

## 2 ЗАДАЧИ РАБОТЫ

Необходимо спроектировать и реализовать язык работы с полиномами и калькулятор полиномов, который был бы удобен в использовании.

Требования:

1. Продуманный синтаксис. Для максимального удобства синтаксис должен быть максимально похож на математический.

2. Обработка полиномов от разных переменных ( $x$ ,  $y$ ,  $z$  и т.п.). Если в выражении участвует одна переменная, то все должно посчитаться без ошибок. Если в выражении появляются разные переменные, например,  $(2x+1)*(y-x)$ , то, вашему желанию, можно или выводить ошибку, или посчитать правильный результат  $(2xy-2x^2+y-x)$ .

3. Наличие переменных, которым можно присваивать полиномы. Например, « $A=x+1$ » означает, что переменной  $A$  присваивается  $x+1$ . Далее, в любом месте, где можно использовать полином в явном виде, можно использовать данную переменную. Также должна быть возможность вывода значений переменных на экран. Синтаксис самих переменных, оператора присваивания и вывода на печать можно придумать, исходя из удобства языка.

4. Возможность задания числовых значений переменным и подсчет значения полинома в этом случае.

5. Программа на вашем языке должна помещаться во входной файл (а не просто в `stdin`), откуда она считывается и выполняется.

6. Развернутые сообщения об ошибках с указанием номера строки, где они произошли. Как минимум, должны появляться по 2-3 сообщения для каждого типа возможных ошибок:

6.1. Лексические. Определяются на этапе лексического анализа (`flex`).

6.2. Синтаксические. Определяются на этапе синтаксического анализа (bison).

6.3. Семантические. Определяются на этапе исполнения.

### **3 ХОД РАБОТЫ**

#### **3.1 Принципы, заложенные в дизайн языка**

Дизайн разработанного языка полиномов основывается на следующих принципах:

1. Приближение к математической нотации. Это означает использование привычных символов для основных операций (+, -, \*, /, ^);
2. Поддержка нескольких переменных полинома (например, x, y, z входят в состав одного полинома  $x^2*y+z$ );
3. Явное описание переменных, для которого используется префикс \$ и присваивание при помощи оператора =, привычные пользователю, знакомому с другими языками программирования;
4. Использование синтаксиса, сходного с языком C (использование символа «;», поддержка однострочных комментариев «//», передача аргументов через символ «,»), способствует снижению порога вхождения и облегчает пользователям освоение разработанного языка.

#### **3.2 Описание грамматики**

Грамматика разработанного языка в форме EBNF выглядит следующим образом:

```
/* Терминалы */
<PRINT>      ::= "print"
<EVAL>       ::= "eval"
<POLY_VAR>   ::= "$" ( [a-z] | [A-Z] | [0-9] )+
<VARIABLE>   ::= [a-z]
<NUMBER>     ::= "-"? [0-9]+

<ADD>        ::= "+"
```

```

<SUB>          ::= "-"
<MUL>          ::= "*"
<DIV>          ::= "/"
<POW>          ::= "^"
<LPAREN>       ::= "("
<RPAREN>       ::= ")"
<COMMA>        ::= ","
<ASSIGN>        ::= "="
<SEMI>         ::= ";"

```

/\* Нетерминалы \*/

```

<program>      ::= <statement>*

```

```

<statement>    ::= <print_stmt>
                  | <assign_stmt>
                  | <eval_stmt>

```

```

<print_stmt>   ::= <PRINT> <LPAREN> <POLY_VAR> <RPAREN> <SEMI>

```

```

<assign_stmt>  ::= <POLY_VAR> <ASSIGN> <expr> <SEMI>

```

```

<eval_stmt>    ::= <EVAL> <LPAREN> <POLY_VAR> ( <COMMA>
<var_assign_list> )? <RPAREN> <SEMI>

```

```

<var_assign_list> ::= <var_assign> ( <COMMA> <var_assign> )*

```

```

<var_assign>   ::= <VARIABLE> <ASSIGN> <NUMBER>

```

```

<expr>         ::= <term> ( ( <ADD> | <SUB> ) <term> )*

```

```

<term>         ::= <factor> ( ( <MUL> | <DIV> ) <factor> )*

```

```

<factor>       ::= <SUB>? ( <VARIABLE>
                          | <POLY_VAR>
                          | <NUMBER>
                          | <LPAREN> <expr> <RPAREN> )

```

( <POW> <NUMBER> ) ?

Программа состоит из последовательности инструкций (<statement>), каждая из которых заканчивается точкой с запятой.

Инструкции:

1. Вывод полинома по имени (print(\$A));
2. Присваивание полинома переменной (\$A = x^2 + 3\*y);
3. Вычисление значения полинома при заданных переменных (eval(\$A, x=2, y=3));

Выражения поддерживают стандартные арифметические операции со следующими приоритетами (от высшего к низшему):

1. Возведение в степень (^);
2. Умножение и деление (\*, /);
3. Сложение и вычитание (+, -);

Факторы могут быть переменными (x, y), именами полиномов (\$A), числами или выражениями в скобках.

В выражениях допускается унарный минус для отрицательных чисел и отрицательных выражений.

Особенности грамматики:

- Имя полинома начинается с \$ и содержит буквы и цифры;
- Переменные — это строчные буквы (a–z);
- Числа — целые, с возможным знаком минус;
- Грамматика поддерживает списки переменных и полиномов для функции eval;

### 3.3 Структура разработанного решения

Разработанная программа состоит из нескольких ключевых компонентов, реализованных в отдельных исходных файлах. Основу составляют файлы parser.y и lexer.l, в которых описаны грамматика языка и правила лексического анализа соответственно. Эти файлы используются для автоматической генерации парсера и лексера с помощью инструментов bison и flex. Кроме того, в проекте

присутствует файл `polynomial.c`, содержащий реализацию основных функций для работы с полиномами — их хранение, арифметические операции и вычисления.

Процесс сборки программы организован с помощью `Makefile`, который автоматизирует последовательность действий. Сначала из файла грамматики `parser.y` с помощью `bison` создаются исходные файлы парсера — `parser.tab.c` и `parser.tab.h`. Аналогично, из файла лексера `lexer.l` с помощью `flex` генерируется файл `lex.yy.c`. Далее все исходные файлы — сгенерированные и собственные — компилируются в объектные файлы с помощью компилятора `gcc` с включением отладочных и предупреждающих опций. После успешной компиляции объектные файлы связываются в единый исполняемый файл с именем `polycalc`. В процессе линковки подключается библиотека `libfl`, необходимая для работы с лексером. Для удобства и чистоты проекта предусмотрена команда `make clean`, которая удаляет все сгенерированные и промежуточные файлы, включая объектные файлы и исполняемый файл.

### **3.4 Представление полиномов, реализация операций над ними.**

#### **Файлы `polynomial.h`, `polynomial.c`**

Полиномы представлены в виде односвязного списка мономов. Каждый моном хранит коэффициент и массив степеней для всех возможных переменных (по алфавиту, от `a` до `z`). Такой подход позволяет реализовать многочлены с несколькими переменными.

Арифметические операции реализованы как манипуляции со списками мономов: сложение объединяет одинаковые по степеням мономы, умножение перемножает каждый моном с каждым, а деление поддерживается только на константу. Для хранения нескольких именованных полиномов используется таблица на односвязных списках, что позволяет быстро находить и обновлять полиномы по имени.

Вычисление значения полинома при подстановке переменных реализовано следующим образом: для каждого монома коэффициент перемножается на

значения переменных в соответствующих степенях, затем все результаты суммируются.

Файлы `polynomial.h` и `polynomial.c` реализуют это решение. Они содержат объявления и реализации необходимых структур и функций.

`Monomial` – структура для хранения одного монома содержит:

- `int coeff` – коэффициент;
- `int powers[VAR_COUNT]` – массив степеней для всех переменных;
- `struct Monomial *next` – указатель на следующий моном.

`Polynomial` – структура для хранения всего полинома, которая содержит `Monomial *head` – указатель на первый моном (голову списка).

`VarAssign` – пара (переменная, значение) для подстановки при вычислении. `VarAssignList` содержит динамический массив таких пар и их количество. `VarValues` – массив значений переменных (индекс соответствует букве).

`PolyEntry` – элемент хеш-таблицы для хранения именованных полиномов, который содержит:

- `char *name` – имя полинома;
- `Polynomial *poly` – указатель на сам полином;
- `struct PolyEntry *next` – следующий элемент таблицы.

Функции для реализации решения можно разделить на некоторые группы.

Создание полиномов, освобождение ресурсов:

- `*Monomial create_monom(int coeff, int *powers)**`: Создает новый моном с заданным коэффициентом и степенями переменных;
- `*Polynomial create_polynomial(Monomial *m)**`: Создает новый полином, содержащий один моном;
- `*Polynomial polynomial_from_const(int c)**`: Создает полином, представляющий константу;
- `*Polynomial polynomial_from_var(char var)**`: Создает полином, представляющий переменную;
- `void free_polynomial(Polynomial *p)`: Освобождает память, выделенную для полинома;

– `void free_var_assign_list(VarAssignList *list):` Освобождает память, выделенную для списка присваиваний переменных.

Арифметические операции:

– `*Polynomial polynomial_add(Polynomial *p1, Polynomial *p2)**:`

Складывает два полинома;

– `*Polynomial polynomial_mul(Polynomial *p1, Polynomial *p2)**:`

Умножает два полинома;

– `*Polynomial polynomial_pow(Polynomial *p, int n)**:` Возводит полином в заданную степень;

– `*Polynomial polynomial_div(Polynomial *p1, Polynomial *p2)**:` Делит полином на константу.

Вывод, вычисление значения полинома:

– `void print_polynomial(Polynomial *p):` Выводит полином в консоль;

– `int evaluate_polynomial(Polynomial *p, VarValues *vars):` Вычисляет значение полинома при заданных значениях переменных.

Проверки, вспомогательные функции:

– `int is_zero_polynomial(Polynomial *p):` Проверяет, является ли полином нулевым;

– `int is_constant_polynomial(Polynomial *p):` Проверяет, является ли полином константой;

– `int check_vars_in_polynomial(Polynomial *p, VarValues *vars):`

Проверяет, содержит ли полином заданные переменные;

`int get_constant_value(Polynomial *p):` Возвращает значение константы в полиноме (если полином — константа).

Работа с таблицей полиномов:

– `void poly_table_init(void):` Инициализирует таблицу полиномов;

– `void poly_table_set(const char *name, Polynomial *p):` Добавляет полином в таблицу под заданным именем;

- `*Polynomial poly_table_get(const char *name)**`: Возвращает полином из таблицы по его имени;
- `void poly_table_free(void)`: Освобождает память, занятую таблицей полиномов.

### 3.5 Файл `lexer.l`

При написании файла-лексера были использованы опции `noouwrap`, `noinput`, `nooutput` для того, чтобы избежать предупреждений вида «error: ‘funcname’ undeclared» и опция `yylineno` для автоматического подсчета строк в глобальной переменной (для сообщений об ошибках).

Секция С-кода содержит подключение заголовочного файла с необходимыми типами данных и заголовка, сгенерированного Bison, в котором содержатся определения токенов. Для работы с памятью и вводом-выводом подключаются стандартные библиотеки. Объявляются глобальная переменная для подсчёта числа строк и отслеживания последнего обработанного символа.

Следующая часть кода содержит правила лексера для регулярных выражений. Пробелы, переносы строк, табуляции, однострочные комментарии игнорируются. Для ключевых слов `print` и `eval` лексер возвращает соответствующие токены. При считывании идентификаторов полиномов и переменных помимо возвращения токена происходит копирование текста в динамическую память для передачи его парсеру. Для последовательности цифр в динамическую память передаётся число. Каждый символ-оператор или разделитель возвращается соответствующим токеном. Если символ не подходит ни под одно из правил, лексер выводит сообщение об ошибке с указанием номера строки и символа, программа завершается.

### 3.6 Файл `parser.y`

В заголовочной части подключаются необходимые заголовочные файлы и стандартные библиотеки, объявляются внешние переменные (аналогично лексеру) и объявляются прототипы используемых функций.



В объединении `%union` определяются типы данных, которые могут храниться в `yyval` для каждого токена или нетерминала (позволяет передавать между правилами значения разного типа). Объявляются токены и типы:

```
%token <str> POLY_VAR VARIABLE
```

```
%token PRINT EVAL
```

```
%token <num> NUMBER
```

```
%token ADD SUB MUL DIV POW LPAREN RPAREN SEMI ASSIGN COMMA
```

```
%type <poly> expr term factor
```

```
%type <var_assign> var_assign
```

```
%type <var_assign_list> var_assign_list
```

Определяются приоритеты операторов: левосторонние умножение и деление с высшим приоритетом, левосторонние сложение и вычитание, правосторонний унарный минус с высшим приоритетом, правостороннее возведение в степень.

Следующая часть парсера содержит правила грамматики. Программа – последовательность нуля или более `statement` (инструкций), допустима пустая программа.

Для инструкций предусмотрено четыре варианта:

1. Вывод полинома;
2. Присваивание выражения полиному;
3. Вычисление полинома без подстановки значений переменных;
4. Вычисление полинома с подстановкой значений переменных.

Для каждой из инструкций написан код обработки такого случая. В обработку включаются вывод некоторых сообщений об ошибках и вызов функций из `polynomial.c` для выполнения действий над полиномами.

Для `print` необходимо получить полином из таблицы по имени. Если полином не найден – выводится ошибка о том, что он не определен. Если полином найден – распечатывается, после чего освобождается память, где находился этот полином (из таблицы извлекается копия в переменную \$3).

Для присваивания проверяется, что парсинг выражения завершен успешно, после чего полином сохраняется в таблице. Если помещение в таблицу завершается неудачей, то выводится ошибка о том, что присваивание завершилось неудачей (в текущей реализации такая ошибка возможна только при недопустимом делении на нуль, так как остальные ошибки – синтаксические и будут выводиться во время парсинга выражения).

При вычислении значения полинома без подстановки значений переменных операция завершится успешно, когда полином является константой, в ином случае пользователь увидит ошибку о недостатке значений переменных для вычисления.

При вычислении значения полинома с переменными формируется список полученных переменных и их значений, осуществляется проверка, что список соответствует требуемым для полинома переменным, после чего происходит вычисление значения и вывод его пользователю. В случае, если переменных, необходимых для полинома недостаточно – пользователь видит соответствующее сообщение об ошибке.

Списки присваиваний переменных (для вычислений выше) обрабатываются следующим образом: первый элемент создает новый список с одним элементом, последующие – добавляют элемент в существующий список, расширяя память. Присваивание переменной создаёт структуру с переменной и её значением.

Выражения `expr`, `term`, `factor` описывают арифметические операции с полиномами. Для реализации унарного минуса используется модификатор `%prec UMINUS`, чтобы избежать конфликта приоритетов (отличие бинарного минуса от унарного).

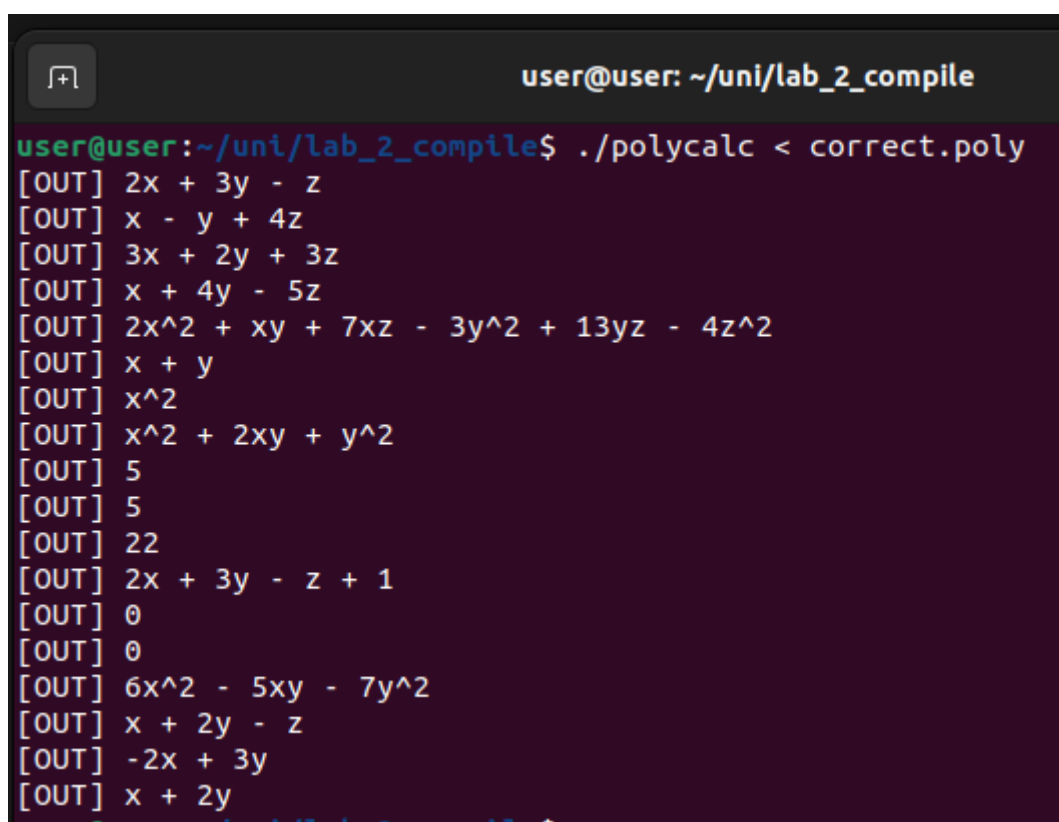
Функция обработки синтаксических ошибок `uerror` выводит подробное сообщение об ошибке с номером строки и символом, возле которого произошла ошибка (если такой символ получен при помощи `last_char`).

Файл завершается основной функцией, которая задаёт общий алгоритм работы программы – инициализация таблицы полиномов, запуск парсера, освобождение ресурсов по завершении его работы.

## 4 ТЕСТИРОВАНИЕ И РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

### 4.1 Примеры обработки корректного кода

Был составлен корректный тестовый файл, в котором отражены возможности работы программы. Результат работы программы для этого файла показан на Рисунок 1.



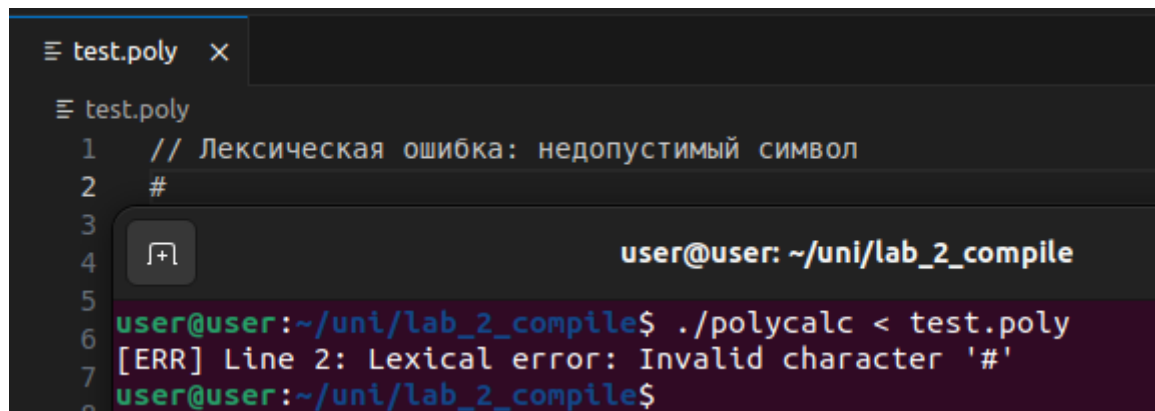
```
user@user: ~/uni/lab_2_compile
user@user:~/uni/lab_2_compile$ ./polycalc < correct.poly
[OUT] 2x + 3y - z
[OUT] x - y + 4z
[OUT] 3x + 2y + 3z
[OUT] x + 4y - 5z
[OUT] 2x^2 + xy + 7xz - 3y^2 + 13yz - 4z^2
[OUT] x + y
[OUT] x^2
[OUT] x^2 + 2xy + y^2
[OUT] 5
[OUT] 5
[OUT] 22
[OUT] 2x + 3y - z + 1
[OUT] 0
[OUT] 0
[OUT] 6x^2 - 5xy - 7y^2
[OUT] x + 2y - z
[OUT] -2x + 3y
[OUT] x + 2y
```

Рисунок 1 - Результат работы программы в случае корректного входного файла

### 4.2 Примеры обработки ошибок

Программа имеет возможность обрабатывать лексические, синтаксические и семантические ошибки. В случае лексических или синтаксических ошибок, программа сразу завершает свою работу. В случае семантической ошибки, действие, которое её породило, игнорируется и программа продолжает работу.

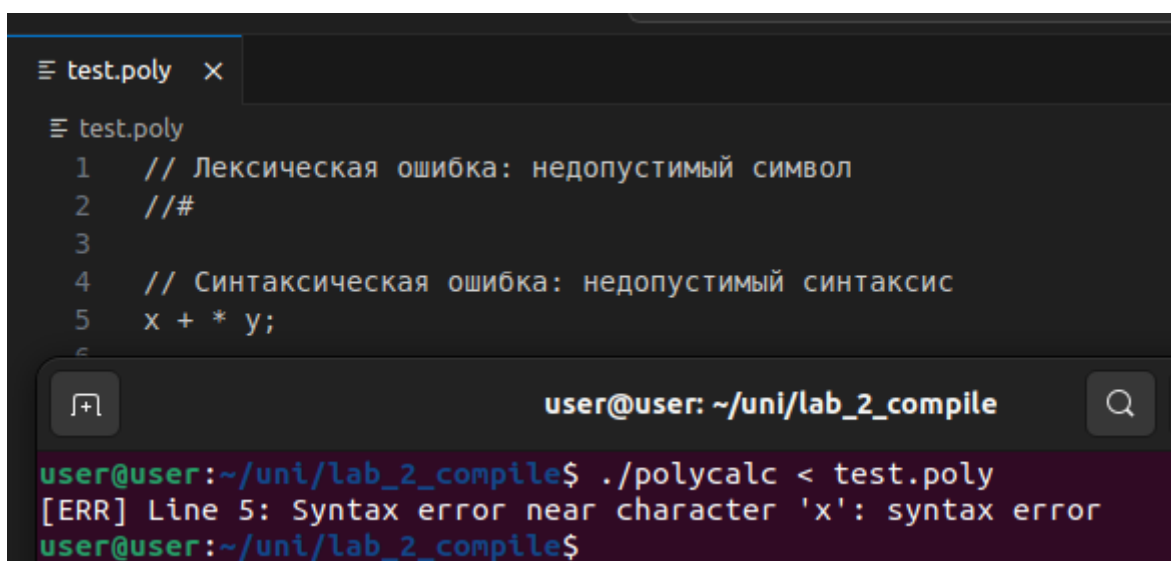
Для получения лексической ошибки на вход необходимо передать цепочку с символом, который не соответствует ни одному из правил лексера, как показано на Рисунок 2.



```
test.poly x
test.poly
1 // Лексическая ошибка: недопустимый символ
2 #
3
4 user@user: ~/uni/lab_2_compile
5
6 user@user:~/uni/lab_2_compile$ ./polycalc < test.poly
7 [ERR] Line 2: Lexical error: Invalid character '#'
8 user@user:~/uni/lab_2_compile$
```

Рисунок 2 - Результат работы программы при обнаружении лексической ошибки

Для получения синтаксической ошибки необходимо передать цепочку, нарушающую синтаксические правила языка (например, пропуск символа, неверный порядок элементов и т.д.), как показано на Рисунок 3.



```
test.poly x
test.poly
1 // Лексическая ошибка: недопустимый символ
2 // #
3
4 // Синтаксическая ошибка: недопустимый синтаксис
5 x + * y;
6
7 user@user: ~/uni/lab_2_compile
8
9 user@user:~/uni/lab_2_compile$ ./polycalc < test.poly
10 [ERR] Line 5: Syntax error near character 'x': syntax error
11 user@user:~/uni/lab_2_compile$
```

Рисунок 3 - Результат работы программы в случае обнаружения синтаксической ошибки

Семантические ошибки могут быть получены в нескольких случаях. Например, при попытке использовать полином, который не был объявлен. Другой случай – попытка выполнить деление на ноль, также невозможно выполнение

деления на полином, не являющийся константой. Сообщения об ошибке также можно увидеть, если попытаться вычислить значение полинома с переменными, не передав значений этих переменных, или передав не все значения переменных. Все эти случаи показаны на Рисунок 4.

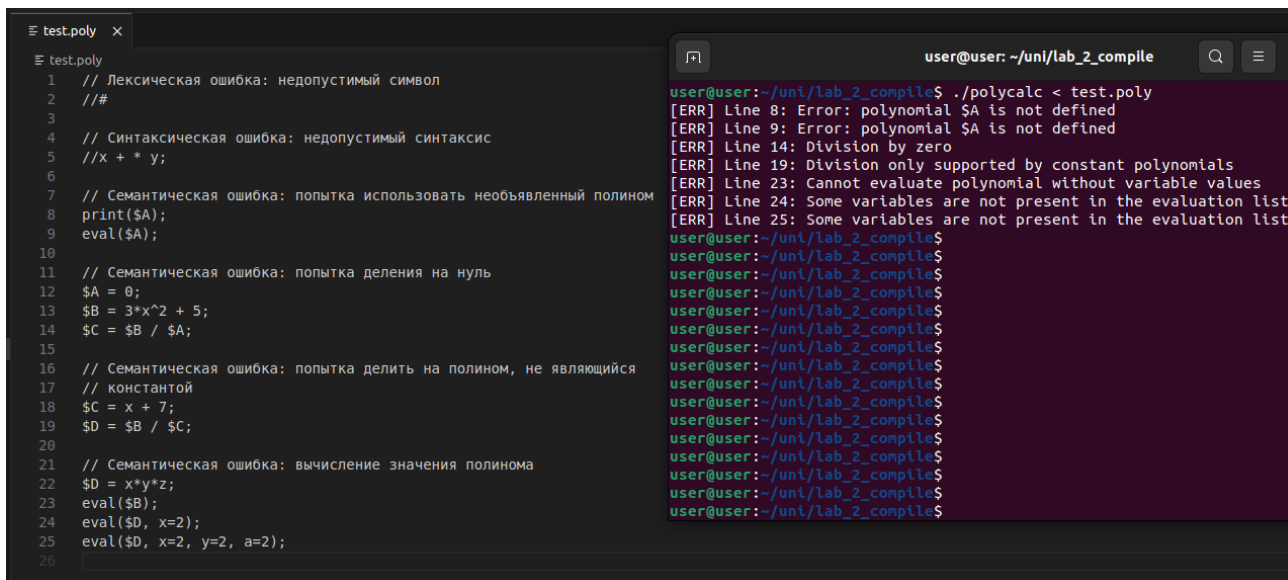


Рисунок 4 - Результат работы программы в случае обнаружения семантических ошибок

## 5 ВЫВОД

В ходе выполнения лабораторной работы был разработан формальный язык работы с полиномами и разработана программа для обработки файлов, написанных на этом языке. Язык позволяет объявлять полиномы, выполнять над ними арифметические операции (сложение, вычитание, умножение, деление на константу и возведение в степень), вычислять значения полиномов при заданных значениях переменных, а также выводить их в консоль. Программа создана с использованием инструментов flex и bison для лексического и синтаксического анализа, а также языка C для реализации логики работы с полиномами.

## ПРИЛОЖЕНИЕ А

### Листинг программы «Makefile»

```
TARGET = polycalc

CC = gcc
CFLAGS = -g -Wall -Wextra

LEX_SRC = lex.yy.c
YACC_SRC = parser.tab.c
POLY_SRC = polynomial.c

OBJ = $(LEX_SRC:.c=.o) $(YACC_SRC:.c=.o) $(POLY_SRC:.c=.o)

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $@ $^ -lfl

lex.yy.c: lexer.l parser.tab.h
    flex lexer.l

parser.tab.c parser.tab.h: parser.y
    bison -d parser.y

clean:
    rm -f $(TARGET) $(OBJ) parser.tab.c parser.tab.h lex.yy.c

.PHONY: all clean
```

## ПРИЛОЖЕНИЕ В

### Листинг программы «lexer.l»

```
%option noyywrap
%option yylineno
%option noinput
%option nounput

%{
#include "polynomial.h"
#include "parser.tab.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern int yylineno;
char last_char = '\0';
%}

%%

[ \t]+          {}
\n              {}

"print"         { return PRINT; }
"eval"          { return EVAL; }

[$@][A-Za-z_][A-Za-z0-9_]* { last_char = *yytext; yylval.str =
strdup(yytext); return POLY_VAR; }

[a-zA-Z_][a-zA-Z0-9_]* { last_char = *yytext; yylval.str =
strdup(yytext); return VARIABLE; }

[0-9]+          { last_char = *yytext; yylval.num =
atoi(yytext); return NUMBER; }

", "            { last_char = *yytext; return COMMA; }
```

```

"+"      { last_char = *yytext; return ADD; }
"_"      { last_char = *yytext; return SUB; }
"*"      { last_char = *yytext; return MUL; }
"/"      { last_char = *yytext; return DIV; }
"^"      { last_char = *yytext; return POW; }
"("      { last_char = *yytext; return LPAREN; }
")"      { last_char = *yytext; return RPAREN; }
";"      { last_char = *yytext; return SEMI; }
"="      { last_char = *yytext; return ASSIGN; }

"//".*   {}

.        {
            last_char = *yytext;
            fprintf(stderr, "[ERR] Line %d: Lexical
error: Invalid character '%c'\n", yylineno, *yytext);
            exit(1);
        }

```



## ПРИЛОЖЕНИЕ С

### Листинг программы «parser.y»

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "polynomial.h"

extern int yylineno;
extern char last_char;

int yylex(void);
void yyerror(const char *s);
int check_vars_in_polynomial(Polynomial *p, VarValues *vars);
%}

%union {
    char *str;
    int num;
    Polynomial *poly;
    VarAssign var_assign;
    VarAssignList var_assign_list;
}

%token <str> POLY_VAR VARIABLE
%token PRINT EVAL
%token <num> NUMBER
%token ADD SUB MUL DIV POW LPAREN RPAREN SEMI ASSIGN COMMA

%type <poly> expr term factor
%type <var_assign> var_assign
%type <var_assign_list> var_assign_list

%left ADD SUB
```

```
%left MUL DIV
%right POW
%right UMINUS
```

```
%%
```

```
program:
```

```
    | program statement
    ;
```

```
statement:
```

```
    PRINT LPAREN POLY_VAR RPAREN SEMI {
        Polynomial *p = poly_table_get($3);
        if (!p) {
            fprintf(stderr, "[ERR] Line %d: Error: polynomial %s
is not defined\n", yylineno, $3);
        } else {
            printf("[OUT] ");
            print_polynomial(p);
        }
        free($3);
    }
    | POLY_VAR ASSIGN expr SEMI {
        if ($3 != NULL) {
            poly_table_set($1, $3);
        }
        free($1);
    }
    | EVAL LPAREN POLY_VAR RPAREN SEMI {
        Polynomial *p = poly_table_get($3);
        if (!p) {
            fprintf(stderr, "[ERR] Line %d: Error: polynomial %s is
not defined\n", yylineno, $3);
        } else {
            if (is_constant_polynomial(p)) {
                printf("[OUT] %d\n", p->head->coeff);
            }
        }
    }
```

```

        } else {
            fprintf(stderr, "[ERR] Line %d: Cannot evaluate
polynomial without variable values\n", yylineno);
        }
    }
    free($3);
}
| EVAL LPAREN POLY_VAR COMMA var_assign_list RPAREN SEMI {
Polynomial *p = poly_table_get($3);
if (!p) {
    fprintf(stderr, "[ERR] Line %d: Error: polynomial %s is not
defined\n", yylineno, $3);
} else {
    VarValues vars = {0};
    for (int i = 0; i < $5.count; i++) {
        char v = $5.vars[i].var;
        int val = $5.vars[i].value;
        if (v >= 'a' && v <= 'z') {
            vars.values[v - 'a'] = val;
        }
    }
    if (check_vars_in_polynomial(p, &vars)) {
        int res = evaluate_polynomial(p, &vars);
        printf("[OUT] %d\n", res);
    } else {
        fprintf(stderr, "[ERR] Line %d: Some variables are not
present in the evaluation list\n", yylineno);
    }
}
free($3);
free_var_assign_list(&$5);
}
;

```

```

var_assign_list:
    var_assign {

```

```

        $$ = (VarAssignList){malloc(sizeof(VarAssign)), 1};
        $$ .vars[0] = $1;
    }
| var_assign_list COMMA var_assign {
    $$ = $1;
    $$ .vars = realloc($$ .vars, sizeof(VarAssign) * ($$
.count + 1));
    $$ .vars[$$ .count] = $3;
    $$ .count++;
}
;

var_assign:
    VARIABLE ASSIGN NUMBER {
        $$ = (VarAssign){$1[0], $3};
        free($1);
    }
;

expr:
    expr ADD term {
        Polynomial *res = polynomial_add($1, $3);
        free_polynomial($1);
        free_polynomial($3);
        $$ = res;
    }
| expr SUB term {
        Polynomial *neg = polynomial_mul(polynomial_from_const(-
1), $3);
        Polynomial *res = polynomial_add($1, neg);
        free_polynomial($1);
        free_polynomial($3);
        free_polynomial(neg);
        $$ = res;
    }
| expr DIV term {

```

```

    Polynomial *res = polynomial_div($1, $3);
    if (!res) {
        free_polynomial($1);
        free_polynomial($3);
        $$ = NULL;
    } else {
        free_polynomial($1);
        free_polynomial($3);
        $$ = res;
    }
}
| term { $$ = $1; }
;

```

term:

```

    term MUL factor {
        Polynomial *res = polynomial_mul($1, $3);
        free_polynomial($1);
        free_polynomial($3);
        $$ = res;
    }
| factor { $$ = $1; }
;

```

factor:

```

    SUB factor %prec UMINUS {
        Polynomial *neg = polynomial_mul(polynomial_from_const(-
1), $2);
        free_polynomial($2);
        $$ = neg;
    }
| VARIABLE {
        Polynomial *p = polynomial_from_var($1[0]);
        free($1);
        $$ = p;
    }

```

```

| POLY_VAR {
    Polynomial *p = poly_table_get($1);
    if (!p) {
        fprintf(stderr, "[ERR] Line %d: Error: polynomial %s
is not defined\n", yylineno, $1);
        p = polynomial_from_const(0);
    }
    Polynomial *copy = polynomial_add(p,
polynomial_from_const(0));
    free($1);
    $$ = copy;
}
| NUMBER {
    Polynomial *p = polynomial_from_const($1);
    $$ = p;
}
| LPAREN expr RPAREN { $$ = $2; }
| factor POW NUMBER {
    Polynomial *res = polynomial_pow($1, $3);
    free_polynomial($1);
    $$ = res;
}
;

```

%%

```

void yyerror(const char *s) {
    if (last_char != '\0') {
        fprintf(stderr, "[ERR] Line %d: Syntax error near character
'%c': %s\n", yylineno, last_char, s);
        last_char = '\0';
    } else {
        fprintf(stderr, "[ERR] Line %d: Syntax error: %s\n",
yylineno, s);
    }
}

```

```
int main() {  
    poly_table_init();  
    yyparse();  
    poly_table_free();  
    return 0;  
}
```

## ПРИЛОЖЕНИЕ D

### Листинг программы «polynomial.h»

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#define VAR_COUNT 26

typedef struct Monomial {
    int coeff;
    int powers[VAR_COUNT];
    struct Monomial *next;
} Monomial;

typedef struct Polynomial {
    Monomial *head;
} Polynomial;

typedef struct {
    char var;
    int value;
} VarAssign;

typedef struct {
    VarAssign *vars;
    int count;
} VarAssignList;

typedef struct {
    int values[VAR_COUNT];
} VarValues;

typedef struct PolyEntry {
    char *name;
    Polynomial *poly;
```



```

    struct PolyEntry *next;
} PolyEntry;

Monomial* create_monom(int coeff, int *powers);
Polynomial* create_polynomial(Monomial *m);
Polynomial* polynomial_from_const(int c);
Polynomial* polynomial_from_var(char var);
void free_polynomial(Polynomial *p);

Polynomial* polynomial_add(Polynomial *p1, Polynomial *p2);
Polynomial* polynomial_mul(Polynomial *p1, Polynomial *p2);
Polynomial* polynomial_pow(Polynomial *p, int n);
Polynomial* polynomial_div(Polynomial *p1, Polynomial *p2);

void print_polynomial(Polynomial *p);
int evaluate_polynomial(Polynomial *p, VarValues *vars);

int is_zero_polynomial(Polynomial *p);
int is_constant_polynomial(Polynomial *p);
int check_vars_in_polynomial(Polynomial *p, VarValues *vars);
int get_constant_value(Polynomial *p);

void poly_table_init(void);
void poly_table_set(const char *name, Polynomial *p);
Polynomial* poly_table_get(const char *name);
void poly_table_free(void);
void free_var_assign_list(VarAssignList *list);

#endif // POLYNOMIAL_H

```

## ПРИЛОЖЕНИЕ Е

### Листинг программы «polynomial.c»

```
#include "polynomial.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>

extern int yylineno;

static PolyEntry *poly_table = NULL;

Monomial* create_monom(int coeff, int *powers) {
    Monomial *m = malloc(sizeof(Monomial));
    m->coeff = coeff;
    if (powers)
        memcpy(m->powers, powers, sizeof(int) * VAR_COUNT);
    else
        memset(m->powers, 0, sizeof(int) * VAR_COUNT);
    m->next = NULL;
    return m;
}

Polynomial* create_polynomial(Monomial *m) {
    Polynomial *p = malloc(sizeof(Polynomial));
    p->head = m;
    return p;
}

Polynomial* polynomial_from_const(int c) {
    Monomial *m = create_monom(c, NULL);
    return create_polynomial(m);
}
```

```

Polynomial* polynomial_from_var(char var) {
    int powers[VAR_COUNT] = {0};
    if (var >= 'A' && var <= 'Z') var = var - 'A' + 'a';
    if (var >= 'a' && var <= 'z') {
        powers[var - 'a'] = 1;
    }
    Monomial *m = create_monom(1, powers);
    return create_polynomial(m);
}

void free_polynomial(Polynomial *p) {
    Monomial *m = p->head;
    while (m) {
        Monomial *next = m->next;
        free(m);
        m = next;
    }
    free(p);
}

int monom_equal(Monomial *a, Monomial *b) { // auxiliary, not in the
header
    for (int i = 0; i < VAR_COUNT; i++) {
        if (a->powers[i] != b->powers[i])
            return 0;
    }
    return 1;
}

void add_monom_to_polynomial(Polynomial *p, Monomial *m) { //
auxiliary, not in the header
    if (m->coeff == 0) {
        free(m);
        return;
    }
    Monomial *cur = p->head;

```

```

Monomial *prev = NULL;
while (cur) {
    if (monom_equal(cur, m)) {
        cur->coeff += m->coeff;
        free(m);
        return;
    }
    prev = cur;
    cur = cur->next;
}
if (prev)
    prev->next = m;
else
    p->head = m;
m->next = NULL;
}

Polynomial* polynomial_add(Polynomial *p1, Polynomial *p2) {
    Polynomial *res = create_polynomial(NULL);
    for (Monomial *m = p1->head; m != NULL; m = m->next) {
        int powers[VAR_COUNT];
        memcpy(powers, m->powers, sizeof(int) * VAR_COUNT);
        Monomial *copy = create_monom(m->coeff, powers);
        add_monom_to_polynomial(res, copy);
    }
    for (Monomial *m = p2->head; m != NULL; m = m->next) {
        int powers[VAR_COUNT];
        memcpy(powers, m->powers, sizeof(int) * VAR_COUNT);
        Monomial *copy = create_monom(m->coeff, powers);
        add_monom_to_polynomial(res, copy);
    }
    return res;
}

```

```

Polynomial* polynomial_mul(Polynomial *p1, Polynomial *p2) {

```

```

Polynomial *res = create_polynomial(NULL);
for (Monomial *m1 = p1->head; m1 != NULL; m1 = m1->next) {
    for (Monomial *m2 = p2->head; m2 != NULL; m2 = m2->next) {
        int powers[VAR_COUNT];
        for (int i = 0; i < VAR_COUNT; i++)
            powers[i] = m1->powers[i] + m2->powers[i];
        Monomial *prod = create_monom(m1->coeff * m2->coeff,
powers);
        add_monom_to_polynomial(res, prod);
    }
}
return res;
}

```

```

Polynomial* polynomial_pow(Polynomial *p, int n) {
    if (n == 0) {
        return polynomial_from_const(1);
    }
    Polynomial *res = polynomial_from_const(1);
    for (int i = 0; i < n; i++) {
        Polynomial *tmp = polynomial_mul(res, p);
        free_polynomial(res);
        res = tmp;
    }
    return res;
}

```

```

Polynomial* polynomial_div(Polynomial *p1, Polynomial *p2) {
    if (!is_constant_polynomial(p2)) {
        fprintf(stderr, "[ERR] Line %d: Division only supported by
constant polynomials\n", yylineno);
        return NULL;
    }
    int divisor = get_constant_value(p2);
    if (divisor == 0) {

```

```

        fprintf(stderr, "[ERR] Line %d: Division by zero\n",
yylineno);
        return NULL;
    }
    Polynomial *result = create_polynomial(NULL);
    for (Monomial *m = p1->head; m != NULL; m = m->next) {
        Monomial *new_m = malloc(sizeof(Monomial));
        new_m->coeff = m->coeff / divisor;
        memcpy(new_m->powers, m->powers, sizeof(int) * VAR_COUNT);
        new_m->next = NULL;
        add_monom_to_polynomial(result, new_m);
    }
    return result;
}

```

```

void print_polynomial(Polynomial *p) {
    Monomial *m = p->head;
    int first = 1;
    while (m) {
        if (m->coeff == 0) {
            m = m->next;
            continue;
        }
        if (!first) {
            if (m->coeff > 0)
                printf(" + ");
            else
                printf(" - ");
        } else {
            if (m->coeff < 0)
                printf("-");
        }
        int abs_coeff = m->coeff < 0 ? -m->coeff : m->coeff;
        int printed_coeff = 0;
        if (abs_coeff != 1) {

```

```

        printf("%d", abs_coeff);
        printed_coeff = 1;
    }
    for (int i = 0; i < VAR_COUNT; i++) {
        if (m->powers[i] > 0) {
            printf("%c", 'a' + i);
            if (m->powers[i] > 1)
                printf("^%d", m->powers[i]);
            printed_coeff = 1;
        }
    }
    if (!printed_coeff)
        printf("%d", abs_coeff);
    first = 0;
    m = m->next;
}

if (first)
    printf("0");
printf("\n");
}

int evaluate_polynomial(Polynomial *p, VarValues *vars) {
    int sum = 0;
    for (Monomial *m = p->head; m != NULL; m = m->next) {
        int term = m->coeff;
        for (int i = 0; i < VAR_COUNT; i++) {
            if (m->powers[i] > 0) {
                if (vars->values[i] == 0) return INT_MIN;
                int base = vars->values[i];
                int exp = m->powers[i];
                int power = 1;
                for (int j = 0; j < exp; j++) power *= base;
                term *= power;
            }
        }
        sum += term;
    }
}

```

```

    }
    return sum;
}

int is_zero_polynomial(Polynomial *p) {
    if (!p || !p->head) return 1;
    Monomial *m = p->head;
    while (m) {
        if (m->coeff != 0) return 0;
        m = m->next;
    }
    return 1;
}

int is_constant_polynomial(Polynomial *p) {
    if (!p || !p->head) return 1;
    Monomial *m = p->head;
    if (m->next != NULL) return 0;
    for (int i = 0; i < VAR_COUNT; i++) {
        if (m->powers[i] != 0) return 0;
    }
    return 1;
}

int check_vars_in_polynomial(Polynomial *p, VarValues *vars) {
    int has_vars = 0;
    for (int i = 0; i < VAR_COUNT; i++) {
        int var_present = 0;
        for (Monomial *m = p->head; m != NULL; m = m->next) {
            if (m->powers[i] > 0) {
                has_vars = 1;
                var_present = 1;
                break;
            }
        }
    }
}

```



```

        if (var_present && vars->values[i] == 0) {
            return 0;
        }
    }
    return has_vars;
}

int get_constant_value(Polynomial *p) {
    if (!p || !p->head) return 0;
    return p->head->coeff;
}

void poly_table_init(void) {
    poly_table_free();
    poly_table = NULL;
}

void poly_table_set(const char *name, Polynomial *p) {
    PolyEntry *cur = poly_table;
    while (cur) {
        if (strcmp(cur->name, name) == 0) {
            free_polynomial(cur->poly);
            cur->poly = p;
            return;
        }
        cur = cur->next;
    }
    PolyEntry *new_entry = malloc(sizeof(PolyEntry));
    new_entry->name = strdup(name);
    new_entry->poly = p;
    new_entry->next = poly_table;
    poly_table = new_entry;
}

```

```

Polynomial* poly_table_get(const char *name) {
    PolyEntry *cur = poly_table;
    while (cur) {
        if (strcmp(cur->name, name) == 0) {
            return cur->poly;
        }
        cur = cur->next;
    }
    return NULL;
}

void poly_table_free(void) {
    PolyEntry *cur = poly_table;
    while (cur) {
        PolyEntry *next = cur->next;
        free(cur->name);
        free_polynomial(cur->poly);
        free(cur);
        cur = next;
    }
    poly_table = NULL;
}

void free_var_assign_list(VarAssignList *list) {
    if (list->vars) {
        free(list->vars);
        list->vars = NULL;
    }
    list->count = 0;
}

```

## ПРИЛОЖЕНИЕ F

### Листинг программы на языке полиномов для тестирования

```
// 1. Базовые арифметические операции
$A = 2*x + 3*y - z;
$B = x - y + 4*z;

// Вывод полиномов
print($A);    // Вывод: 2x + 3y - z
print($B);    // Вывод: x - y + 4z

// Сложение
$C = $A + $B;
print($C);    // Вывод: 3x + 2y + 3z

// Вычитание
$D = $A - $B;
print($D);    // Вывод: x + 4y - 5z

// Умножение
$E = $A * $B;
print($E);    // Вывод: 2x^2 + xy + 7xz + 3xy - 3y^2 + 11yz - xz + yz
               - 4z^2

//Деление (на константу)
$F = $A / 2;
print($F);    // Вывод: x + y

// 2. Степени
$G = x^2;
print($G);    // Вывод: x^2

$H = (x+y)^2;
print($H);    // Вывод: x^2 + 2xy + y^2

// 3. Вычисление
```

```

// Вычисление константного полинома
$K = 5;
eval($K);    // Вывод: 5

// Вычисление полинома с переменными
$L = x + y;
eval($L, x=2, y=3); // Вывод: 5

// Вычисление более сложного полинома
$M = x*y + z^2;
eval($M, x=2, y=3, z=4); // Вывод: 22

// 4. Переназначение полиномов
$A = $A + 1;
print($A);    // Вывод: 2x + 3y - z + 1

// 5. Нулевой полином
$ZERO = 0;
print($ZERO); // Вывод: 0

$N = $A - $A;
print($N); // Вывод: 0

// 6. Полином с несколькими операциями
$P = (2*x + y) * (x - 3*y) + 4*(x^2 - y^2);
print($P);    // Вывод: 6x^2 - 5xy - 4y^2

// 7. Комментарии
// Это комментарий, он должен быть проигнорирован

//8. Проверка со скобками
$Q = (x + (2*y - z));
print($Q); //Вывод: x + 2y - z

//9. Проверка с отрицательными числами
$R = -2*x + 3*y;

```

```
print($R); //Вывод:  $-2x + 3y$ 
```

```
$S = x - ( -2 * y);
```

```
print($S); //Вывод:  $x + 2y$ 
```

## ПРИЛОЖЕНИЕ F

### Листинг программы на языке полиномов для репрезентации ошибок

```
// Лексическая ошибка: недопустимый символ
//#

// Синтаксическая ошибка: недопустимый синтаксис
//x + * y;

// Семантическая ошибка: попытка использовать необъявленный полином
print($A);
eval($A);

// Семантическая ошибка: попытка деления на нуль
$A = 0;
$B = 3*x^2 + 5;
$C = $B / $A;

// Семантическая ошибка: попытка делить на полином, не являющийся
// константой
$C = x + 7;
$D = $B / $C;

// Семантическая ошибка: вычисление значения полинома
$D = x*y*z;
eval($B);
eval($D, x=2);
eval($D, x=2, y=2, a=2);
```