Содержание

[Формулировка задачи 2](#_Toc107065733)

[Ход работы 4](#_Toc107065734)

[Общие положения 4](#_Toc107065735)

[Представление полиномов, реализация математических операций, особые правила 6](#_Toc107065736)

[Ошибки грамматики 8](#_Toc107065737)

[Ошибки и предупреждения 9](#_Toc107065738)

[Вывод 11](#_Toc107065739)

[Приложения 12](#_Toc107065740)

# Формулировка задачи

Необходимо улучшить и дополнить язык полиномов, созданный в первой лабораторной работе.

Требования:

1) Продуманный синтаксис. Для максимального удобства синтаксис должен быть максимально похож на математический. Иначе говоря, было бы очень неплохо, если некоторые операции умножения можно опускать: не 2x\*(x+1), а 2x(x+1). Это необязательное требование, но, если вы хотите спроектировать действительно удобный язык, то лучше применять общепринятые правила записи операций с полиномами

2) Появляются полиномы 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 то вашему желанию можно или выводить ошибку, или же посчитать правильный результат (2xy-2x^2+y-x).

3) Появляются переменные, 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 переменную типа полином.

Также, переменную можно выводить на экран. Синтаксис самих переменных, оператора присваивания и вывода на печать придумайте исходя из удобства языка.

4) Теперь программа на вашем языке должна помещаться во входной файл (а не просто в stdin), откуда она считывается и исполняется.

5) Появляются развернутые сообщения об ошибках с указанием номера строки, где они произошли. Как минимум, должны появиться по 2-3 сообщения для каждого типа возможных ошибок

а) лексические. Определяются на этапе лексического анализа (lex)

б) синтаксические. Определяются на этапе синтаксического анализа (yacc)

в) семантические. Определяются на этапе исполнения

# Ход работы

## Общие положения

В ходе создания более улучшенной версии языка полиномов в первую очередь были произведены изменения, касающиеся понимания термина «полином» и «переменная». Ранее под «переменная» понималась отдельная единичная конструкция полинома, а «полином» же являлся множеством данных переменных с обязательными математическими знаками между ними. Ниже представлена грамматика языка первой работы и структуры используемых данных, которые наглядно демонстрируют данный недостаток:

expression:

t\_Obracket expression t\_Cbracket

| expression t\_SignPlus expression

| expression t\_SignMinus expression

| expression t\_SignMulti expression

| variable

;

struct variableStruct

{

int coeff;

int degree;

};

struct expressionStruct

{

int isActive[32];

int count;

int coeffsArray[64];

int degreesArray[64];

};

Как можно видеть, каждый отдельно взятый операнд (будь то n, n\*x или n\*x^m, где n и m – целые числа) – это «переменная». Во второй работе этот принцип был изменён на более гибкий и, пожалуй, более корректный с математической точки зрения – теперь каждая отдельная единица полинома и есть полином. В математике любое выражение (2, 2\*x и тому подобные) являются полиномами, записанными в сокращённом виде: «2» эквивалентно выражению «2\*x^0», например.

Соответственно, имя «переменная» или «variable» теперь может быть использовано для единицы языка, больше подходящей по смыслу в контексте данной работы. В данном случае – это новый отдельный тип, способный хранить в себе значение произвольного полинома.

## Представление полиномов, реализация математических операций, особые правила

Представление полиномов сохранилось тем же. Полином представляет собой структуру данных, содержащую в себе набор коэффициентов и степеней. Так же в структуре хранится информация о размере полинома.

Реализация операций над полиномами не отличается чем-то уникальным и представляет собой просто набор последовательных операций, целью которых является перебор элементов двух полиномов и выполнение необходимой операции между ними.

По факту грамматика элементарна и представляет собой корректное разложение подаваемых строк (предположительно полиномов, равенств или вызовов функции «print») на более мелкие токены с последующей проверкой корректности с точки зрения математических выражений.

В новую грамматику было добавлено правило, позволяющее опускать знак умножения между двумя полиномами в скобках:

polynom:

…

| t\_Obracket polynom t\_Cbracket nextO\_bracket

{

…

}

, где правило nextO\_bracket представимо в следующем виде:

nextO\_bracket:

nextO\_bracket t\_Obracket polynom t\_Cbracket

| t\_Obracket polynom t\_Cbracket

;

Также стоит отметить ряд особых правил. Первое – это правила, возводящие число или полином в ряд степеней. Во-первых, встречается конфликт сдвиг/свёртки, который не представляет какой-либо ошибки с математической точки зрения – рано или поздно числа, возводимые в степень других чисел (последовательность x^x^x^…, где x – число), свернутся до одного числа.

Во-вторых, новой особенностью данной грамматики является работа с переменными. Для реализации этого не было задействовано каких-либо особых правил, но была произведена работа над обработкой ошибок, связанных с переменными. Например, нужно учитывать то, инициализировалась ли переменная в случаях её использования.

Также стоит отметить правило, позволяющее возводить переменные и численные значения в степень числового значения, а не единственного числа. Добавленные правила приводят к появлению ещё одного конфликта «сдвиг/свёртка», но, по опыту проверенных примеров, данный конфликт не влияет на работу программы. Ряд добавленных правил следующий:

digitBrace:

digit

{

$$ = $1;

}

|

digitBrace t\_SignPlus digitBrace

{

$$ = $1 + $3;

}

|

digitBrace t\_SignMinus digitBrace

{

$$ = $1 - $3;

}

|

digitBrace t\_SignMulti digitBrace

{

$$ = $1 \* $3;

}

;

Это основное фигурирующее правило. Оно добавляется ко всем правилом со степенями с применением скобок следующим образом:

| t\_LETTER '^' digit

| t\_LETTER '^' t\_Obracket digitBrace t\_Cbracket

## Ошибки грамматики

Ошибки, выводимые yacc при компиляции были только в первой версии первой лабораторной работы. Ошибки заключались в неоднозначности грамматики. После получения ряда данных ошибок грамматика была практически полностью переписана, а ошибки были исключены. Информация о тех ошибках не была сохранена. В конечной версии второй работы имеется три ошибки типа «сдвиг/свёртка», но эти элементы грамматики являются ошибками только с общей точки зрения и, соответственно, с точки зрения компилятора yacc. Ошибка заключается в несовместимости правил:

Polynom:

…

polynom '^' digit

…

И

digit:

…

digit '^' digit

…

На самом деле совокупность данных правил хоть и является «тавтологией», ни к какой ошибке она не приведёт. Поскольку до тех пор, пока мы работаем с одним числом – мы находимся в рамках второго правила. Но если на входе мы будем иметь не число типа «int», а полином, то грамматика к этому моменту уже свернёт рассматриваемую конструкцию до полинома, и операция(операции) возведения в степень будет производиться в рамках первого правила.

Единственно возможный ошибочный случай был рассмотрен ранее.

Также после добавления правила, позволяющего возводить в степень не одного числа, а числового выражения, появился ещё один конфликт «сдивг/свёртка». Он имеет ту же природу, что и два конфликта, рассмотренных ранее.

## Ошибки и предупреждения

Ошибки, согласно спецификации задания, разделяются на три категории: лексические, синтаксические и семантические.

Лексические ошибки необходимо замечать и анализировать в функции лексического анализа yylex(). Множество данных ошибок ограничивается обработкой команды «print». Если реализация команды не отвечает спецификации – будет выведена ошибка и программа будет остановлена.

Синтаксические ошибки выявляются на стадии разбора последовательности токенов при условии корректности самих токенов. Грубо говоря – при правильных минимальных единицах языка должна соблюдаться корректная их последовательность. Последовательность токенов должна отвечать любой из возможных последовательностей, выводимых из данной грамматики. Ошибки такого рода, зачастую, являются общими, поэтому используется общая функция, уведомляющая о наличии ошибки, указывающая номер строки и место ошибки в ней.

Множество семантических ошибок ограничивается обработкой случая возведения нуля в нулевую степень. Обработка данной ошибки, вообще, происходит на этапе синтаксической обработки с тем лишь отличием, что во время синтаксического разбора в общем случае, значения терминалов и нетерминалов не читаются и не учитываются. Семантическая же ошибка учитывает значения единиц грамматики данного правила.

Ниже приведены выводы всех ошибок:

Text

Description automatically generated

Рисунок 1. Использование переменной в полиноме, отличающейся от ‘x’.

Graphical user interface, text

Description automatically generated

Рисунок 2. Некорректная команда «print», один вариант ошибки (если длина слова, похожего на «print» меньше 5-ти).

Text

Description automatically generated

Рисунок 3. Некорректная команда «print» при неправильном её написании (неправильная буква).

Text

Description automatically generated

Рисунок 4. Некорректное математическое действие (возведение 0-ля в 0-ую степень).

Text

Description automatically generated

Рисунок 5. Ошибка общего рода (или синтаксическая ошибка).

# Вывод

В процессе выполнения данной лабораторной работы был улучшен компилятор для языка полиномов, созданный в результате выполнения первой работы. Улучшения касались многих аспектов и усовершенствованная версия гораздо функциональнее старой. Присутствует анализ поступаемого кода на предмет ошибок и вывод информации о них. Теперь в файле может быть множество полиномов – для работы предоставляется множество строк, а не одна. Полиномы могут записываться «в память» (в переменные) и, как следствие, могут использоваться в дальнейшем для выполнения операций над ними или же для вывода на экран.

Также были усовершенствованы ранее имеющиеся функции, была произведена оптимизация кода.

# Приложения

Исходный код грамматики без действий

main:

| expression

| main '\n' expression

| main '\n'

;

expression:

polynom

| t\_PRINT variable

| variable '=' polynom

;

variable:

'$' t\_LETTER

;

polynom:

t\_Obracket t\_SignMinus polynom t\_Cbracket

| t\_Obracket polynom t\_Cbracket

| t\_Obracket polynom t\_Cbracket t\_Obracket polynom t\_Cbracket

| polynom t\_SignPlus polynom

| polynom t\_SignMinus polynom

| polynom t\_SignMulti polynom

| polynom '^' digit

| polynom '^' t\_Obracket digitBrace t\_Cbracket

| t\_NUMBER

| t\_LETTER

| t\_LETTER '^' digit

| t\_LETTER '^' t\_Obracket digitBrace t\_Cbracket

| t\_NUMBER t\_LETTER

| t\_NUMBER t\_LETTER '^' digit

| t\_NUMBER t\_LETTER '^' t\_Obracket digitBrace t\_Cbracket

|

variable

// pseudo-rules for error in grammatic

| polynom t\_SignPlus t\_SignPlus polynom

| polynom t\_SignPlus t\_SignMinus polynom

| polynom t\_SignPlus t\_SignMulti polynom

| polynom t\_SignMinus t\_SignMinus polynom

| polynom t\_SignMinus t\_SignPlus polynom

| polynom t\_SignMinus t\_SignMulti polynom

| polynom t\_SignMulti t\_SignMulti polynom

| polynom t\_SignMulti t\_SignPlus polynom

| polynom t\_SignMulti t\_SignMinus polynom

;

digitBrace:

digit

|

digitBrace t\_SignPlus digitBrace

|

digitBrace t\_SignMinus digitBrace

|

digitBrace t\_SignMulti digitBrace

;

digit:

t\_NUMBER

| t\_Obracket digit t\_Cbracket

| digit '^' digit

;