МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

ЛЕКСИЧЕСКИЙ И СИНТАКСИЧЕСКИЙ АНАЛИЗ ВЫРАЖЕНИЙ

КУРСОВАЯ РАБОТА

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Рыданова Никиты Сергеевича

Научный руководитель
доцент _____ Г. Г. Наркайтис

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2021

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ДКА — детерменированный конечный автомат.

КС-грамматика — контекстно-свободная грамматика.

ПС-анализатор — анализатор типа «перенос-свертка».

ПС-анализ — анализ типа «перенос-свертка».

АСД — абстрактное синтаксическое дерево.

ВВЕДЕНИЕ

Лексические и синтаксические анализаторы как предмет научного исследования появились в 1950-х годах XX века. Бурный рост интереса был обусловлен в том числе необходимостью создания интерпретаторов и компиляторов для трансляции языков высокого уровня в другие языки или напрямую в машинный код.

Несмотря на то, что сейчас эта область является хорошо изученной, она все еще актуальна и находит свои применения при создании статических и динамических компиляторов, трансляции одних языков программирования в другие, обработке SQL-запросов и некоторых других классов задач [1, 2]. Необходимость быстрого анализа при этом очевидна.

Целью данной работы является изучение работы лексических и синтаксических анализаторов и повышение эффективности анализа за счет различных оптимизаций работы с оперативной памятью.

В ходе написания работы должны быть решены следующие задачи:

1. Изучение понятий лексического и синтаксического анализа.
2. Анализ технических особенностей реализации лексических и синтаксических анализаторов и изучение работы генераторов лексического и синтаксического анализа на примере Flex и GNU Bison.
3. Создание лексического и синтаксического анализатора для анализа математического выражения.
4. Изучение понятия абстрактного синтаксического дерева.
5. Создание нескольких реализаций абстрактного синтаксического дерева для построенной грамматики и сравнение эффективности их работы.

Анализ выражений

1 Конечные автоматы

Основой построения лексических и синтаксических анализаторов являются конечные автоматы [2]. Автомат представляет из себя модель дискретного устройства, имеющего некоторое множество состояний и переходов между ними. Более формально: конечный автомат определяется как пятерка:

$$A = \langle Q, \Sigma, \Delta, I \subseteq Q, F \subseteq Q \rangle$$

где Q — множество состояний автомата,

Σ — алфавит,

I — начальное состояние автомата,

F — множество допускающих состояний,

$\Delta \subseteq Q \times \Sigma^* \times Q$ — множество переходов автомата.

При этом в начальный момент времени $t = 0$ автомат находится в некотором фиксированном начальном состоянии $s_0 \in Q$.

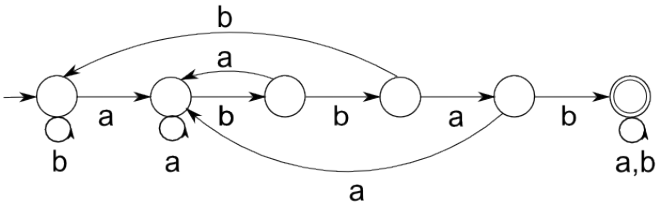

Рисунок 1 – Пример конечного автомата для распознавания строки abbab

У меня есть рублей

Такая модель является простым, удобным и мощным методом для решения достаточно широкого класса задач в программировании [3]. В частности, мы увидим, что она же лежит и в основе анализаторов [4].

2 Лексический анализ

2.1 Понятие лексического анализа

Лексический анализ — в общем случае процесс преобразования потока символов, составляющих исходную последовательность в лексически значащие последовательности.

Лексический анализ является первым этапом разбора некоторого выражения — например, исходного кода программы на некотором языке. Он необходим для упрощения дальнейшего синтаксического анализа программы как с точки зрения логики и удобства для разработчика, так и с точки зрения вычислений.

Действительно, в случае рассмотрения выражения на некотором языке программирования, постоянную величину в этом выражении логически гораздо удобнее бы было считать одной целой лексемой, нежели анализировать это как набор подряд идущих цифр. А применение специализированных методик, предназначенных для решения лексических задач позволяет сделать анализ ещё более эффективным [2].

2.2 Реализация лексического анализа

При реализации лексического анализа оперируют следующими основными терминами:

— **Токен.** Токен представляет из себя пару, состоящую из имени токена и значения атрибута:

$$\langle token\_name, attribute\_value \rangle$$

Имя токена представляет из себя абстрактный символ, необходимый во время синтаксического анализа, а значение атрибута указывает на запись в таблице символов [2]. Токены также могут содержать некоторые дополнительные атрибуты [5].

— **Шаблон.** Некоторое описание вида, которое может принимать лексема токена. Для отдельных ключевых слов или операторов чаще всего представляет из себя их строковое представление, для других же лексем может из себя представлять некоторое множество представлений, заданное регулярным выражением.

— **Лексема.** Последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется анализатором как эк-

земпляр токена [2].

В подавляющем большинстве случаев лексические анализаторы представляют в виде детерменированных конечных автоматов (ДКА). Например, ДКА для распознавания ключевого слова `then` может быть представлен как на рис.2:

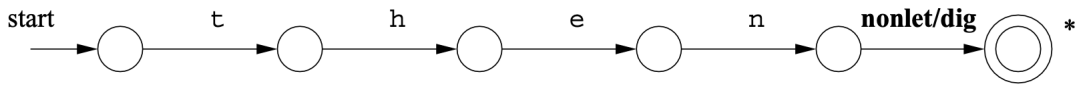

Рисунок 2 – Пример ДКА для ключевого слова then

3 Синтаксический анализ

3.1 Понятие синтаксического анализа

Обычно каждое выражение имеет строгую синтаксическую структуру, позволяющую однозначно его интерпретировать. В случае математического выражения мы получаем численное значение выражения, а в случае, если мы говорим о языках программирования, то набор синтаксических правил необходим для определения корректной программы.

Концептуально синтаксический анализ является логическим продолжением лексического анализа. В качестве набора входных данных он использует полученные ранее в результате лексического анализа токены и строит дерево разбора, которое впоследствии подвергается дальнейшей обработке. Стоит отметить, что в некоторых случаях явное построение дерева разбора является необязательным, поскольку проверки и дополнительные действия над токенами могут проводится и в процессе синтаксического анализа [2]. Здесь же отметим, что синтаксический анализаторы также представляют из себя конечные автоматы.

Вполне естественно звучит следующий вопрос: «Каким образом возможно задать правила, согласно которым будет осуществляться синтаксический разбор выражения?»

3.2 Описание синтаксических конструкций

Синтаксис конструкций может быть описан с помощью контекстно-свободных грамматик или записи формы Бэкуса-Наура. Использование грамматик имеет следующие преимущества:

- Она дает точную и простую для понимания синтаксическую структуру.
- Правильно построенная грамматика способствует облегчению трансляции исходного выражения и выявлению ошибок.
- Грамматика допускает гибкое расширение функциональности за счет добавления дополнительных конструкций [2, 6].

Теперь формально определим, что мы понимаем под контекстно-свободной грамматикой. Контекстно-свободная грамматика — система, представляющая собой четверку:

$$\Gamma = \langle L, N, S \in N, P \subset (N \times (N \cup \Sigma)^*) \rangle$$

где:

— $L \subset \Sigma^*$ — формальный язык над алфавитом Σ, элементы которого называются терминалами.

— N — множество элементов, называемых нетерминалами.

— S — начальный символ грамматики (или аксиома грамматики).

— P — набор правил (или продукций) вывода $\alpha \to \beta$.

При этом L и N — конечные, непересекающиеся множества [4, 7].

В качестве примера контекстно-свободной грамматики можно рассмотреть язык правильных скобочных последовательностей.

В нем $\Sigma = \{(\ ,)\}$, $N = \{S\}$, S является стартовым нетерминалом, а P состоит из правил:

1. $S \to SS$
2. $S \to (S)$
3. $S \to \varepsilon$

Теперь поймем, каким образом можно проверить, возможно ли порождение заданного выражения нашей КС-грамматикой.

3.3 Виды синтаксического анализа

Различают три основных вида синтаксических анализаторов — универсальные, нисходящие и восходящие.

Универсальные методы часто оказываются слишком неэффективны в общем случае, поэтому чаще используют нисходящие и восходящие анализаторы [2, 6].

3.3.1 Нисходящий анализ

В основе нисходящего анализа лежит идея, согласно которой имея начальный нетерминальный символ осуществляется попытка построить заданную строку.

Другими словами, мы пытаемся построить дерево разбора от корня к листьям, последовательно применяя правила к ещё не рассмотренным нетерминалам построенного на текущий момент дерева. В начальный момент времени единственным символов является стартовый нетерминал, являющимся корнем дерева [2, 5].

Если вывод оказывается успешным, то в таком случае говорят, что строка является порождением заданной грамматики.

Очевидно, что над некоторым терминалом возможно применение сразу нескольких продукций, и иногда могут потребоваться дополнительные проверки, чтобы убедиться, что из всех продукций над рассматриваемым нетерминалом была выбрана верная.

Это достаточно неэффективно, поэтому можно, например, воспользоваться предиктивным синтаксическим анализом, суть которого заключается в просмотре на несколько символов вперед, с целью однозначно выбрать верное правило. Если для некоторой грамматики такое возможно, то говорят, что она принадлежит классу LL(k) грамматик, а построенный по этим правилам анализатор — LL(k)-анализатором [2, 7].

3.3.2 Восходящий анализ

Как уже было замечено ранее, восходящий анализ представляет из себя построение дерева разбора от листьев к корню.

Осуществляется восходящий синтаксический анализ с помощью последовательной замены некоторой продукции на ее заголовок с целью получить стартовый символ грамматики. Ключевое значение при данном подходе имеет то, когда и используя какую продукцию применять замену в процессе разбора [2].

Процесс восходящего синтаксического анализа можно рассматривать как процесс «перенос-свертка». Для уже считанных терминалов и полученных в результате применения правил нетерминалов используется стек, остальные же символы располагаются в входном потоке.

На каждом шаге анализатор считывает ноль или более символов до тех пор, пока не появится возможность применить свертку к символам, находящимся на вершине стека. Затем применяется свертка и заголовок помещается обратно на вершину. Процесс повторяется до тех пор, пока не будет получен стартовый символ или обнаружена ошибка [2].

Таким образом, ПС-анализатор может выполнить одно из следующий четырех действий:

1. Перенос (shift). Перенос очередного входного символа на вершину стека.
2. Свертка (reduce). Правая часть сворачиваемой строки располагается на вершине стека, после чего определяется левая граница строки и принимается решение о замене.
3. Принятие (accept). Объявление об успешном завершении анализа.

4. Ошибка (error). Сообщение об ошибке.

3.3.3 LR(k)-анализаторы

Наиболее распространенный тип восходящих синтаксических анализаторов основан на концепции, так называемого, LR(k)-анализа.

Эта концепция содержит в себе следующие договоренности:

1. **L.** Входной поток сканируется слева направо.
2. **R.** Построение правого порождения предполагается в обратном порядке.

При этом число k означает число предпросматриваемых символов, которое необходимо для успешного принятия решения.

Этот класс анализаторов является доминирующим по следующим причинам:

1. Принадлежащие этому классу анализаторы могут быть использованы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика.
2. Метод LR-анализа — наиболее общий метод ПС-анализа без возврата, при этом не уступающий в эффективности более примитивным методам.
3. LR-анализатор способен обнаруживать синтаксические ошибки сразу же, как только это становится возможным при просмотре входного потока.
4. Множество грамматик, которые могут быть проанализированы LR-методами представляет из себя надмножество грамматик, которые могут быть проанализированы LL-методами.

Стоит отметить, что основным недостатком LR-метода считается сложность ручного построения LR-анализатора по заданной грамматике [1]. Однако на сегодняшний день существует значительное число автоматических генераторов таких анализаторов, об одном из которых далее и пойдет речь.

Flex & Bison

В этой главе будут рассмотрены практические аспекты реализации лексического и синтаксического анализа, методы построения абстрактных синтаксических деревьев и сравнение эффективности их работы.

1 Flex

Как уже было замечено, считается, что для наиболее эффективного и простого синтаксического анализа выражений необходим предварительный лексический анализ. Для этого возникает необходимость в инструменте, позволяющем автоматически создавать лексический анализатор по заданному описанию грамматики. Одним из таких инструментов является Flex.

1.1 Описание

Flex — кроссплатформенное ПО с открытым исходным кодом для создания эффективных генераторов лексических анализаторов.

Часто применяется в системных библиотеках UNIX-подобных операционных систем. Является улучшенной версией более раннего генератора лексических анализаторов GNU Lex.

Асимптотическая сложность лексических анализаторов, созданных при помощи Flex составляет $O(n)$, так как над каждым символов выполняется лишь некоторое небольшое постоянное число операций. Для генерации кода допускается использование языков C/C++, что также гарантирует достаточно высокую производительность [1].

1.2 Пример программы

Пример простейшей программы Flex, выполняющей подсчет количество символов, слов и строк, представлен ниже:

```
1   // Блок объявлений и подстановок
2   %{
3       int chars = 0;
4       int words = 0;
5       int lines = 0;
6   }
7   %%
8   // Блок правил
9   [a-zA-Z]+ {words++; chars += strlen(yytext); }
10  \n {chars++; lines++; }
11  . {chars++; }
12  %%
13  // Блок пользовательского кода
14  main(int argc, char** argv) {
15      // Запуск работы анализатора
```

```
16          yylex();
17          // Вывод результата
18          printf("%8d%8d%8d\n", lines, words, chars);
19      }
```

Содержимое программы состоит из трех блоков, разделенных %%. Первый блок содержит в себе объявления и параметры (в нашем случае это объявление и инициализация переменных `chars`, `words`, `lines`). Содержимое этого блока дословно копируется в полученный исходный код генератора на языке C.

Второй блок представляет из себя описание лексических правил. Каждая строка этого блока содержит шаблон (символ, строку или регулярное выражение) и исполняемый код. Если в ходе лексического анализа будет замечен данный шаблон, то будет исполнен код, заключенный между фигурными скобками.

Третий блок является необязательным и может содержать пользовательские инструкции на C/C++ [1].

2 GNU Bison

Ранее было озвучено, что построение LR(k)-анализаторов вручную представляет из себя достаточно трудную задачу. Для решения этой трудности было придумано множество инструментов, позволяющих генерировать синтаксические анализаторы по заданной грамматике.

В рамках данной работы будет рассмотрен один из таких инструментов — GNU Bison.

2.1 Описание

GNU Bison — кроссплатформенное ПО с открытым исходным кодом для генерации синтаксических анализаторов по заданному описанию грамматики. Является улучшенной версией более раннего генератора синтаксического анализа GNU Yacc. Для генерации кода используются языки C/C++.

Так же, как и Flex часто применяется для системных библиотек в UNIX-подобных системах. Здесь стоит также отметить, что Flex и GNU Bison зачастую применяются вместе.

По умолчанию при задании контекстно-свободной грамматики GNU Bison создает LALR(1)-анализатор, являющийся улучшенной версией LR(0)-анализатора, однако возможно создание канонических LR-анализаторов, а также некоторых других модификаций LR-грамматик [1, 2, 8].

2.2 Пример программы

Простейший пример программы с использованием GNU Bison, вычисляющее значение математического выражения представлен ниже:

```
1  // Блок объявлений
2  %{
3  #include <stdio.h>
4  }
5  %token NUMBER
6  %token ADD SUB MUL DIV ABS
7  %token EOL
8
9  %%
10 // Описание грамматики
11 calclist:
12 | calclist exp EOL { printf("= %d\n", $1); }
13
```

```
14 exp: factor
15 | exp ADD factor { $$ = $1 + $3; }
16 | exp SUB factor { $$ = $1 - $3; }
17 ;
18 factor: term
19 | factor MUL term { $$ = $1 * $3; }
20 | factor DIV term { $$ = $1 / $3; }
21 ;
22 term: NUMBER
23 | ABS term { $$ = $2 >= 0? $2 : - $2; }
24 ;
25 %%
26 // Блок пользовательского кода
27 main(int argc, char** argv) {
28     // Запуск работы анализатора
29     yyparse();
30 }
31 // Обработка сообщения об ошибке
32 yyerror(char* s) {
33     fprintf(stderr, "error: %s\n", s);
34 }
```

Синтаксис GNU Bison практически идентичен с синтаксисом Flex.

Первый и третий блоки имеют то же назначение, второй же блок в данном случае содержит описание КС-грамматики. При этом первый объявленный нетерминал является стартовым, то есть в конечном счете именно в него должно быть свернуто исходное выражение [1].

3 Реализация анализатора выражений

Поставим задачу построения синтаксического анализатора, выполняющего вычисление простейшего математического выражения.

3.1 Построение лексического анализатора

Для начала необходимо определить множество терминалов, которые будут использованы при анализе. С терминалами задача достаточно тривиальная — нам необходимы символы, которые могут быть частью выражения.

В рамках задачи ограничимся множеством L состоящим из всевозможных числовых констант и математических операторов. Таким образом, описание лексики для Flex выглядит следующим образом:

```
1  // Регулярное выражение, соответствующее экспоненциальной части записи числа
2  EXP ([Ee][-+]?[0-9]+)
3  %%
4  // Для математических операторов в качестве имени нетерминала вернем их
   ↪  строковое представление
5  "+" |
6  "-" |
7  "*" |
8  "/" |
9  "|" |
10 "(" |
11 ")"  {return yytext[0]; }
12 // Для чисел внесем в качестве атрибута токена численное значение и вернем
   ↪  нетерминал NUMBER.
13 [0-9]+"."[0-9]*{EXP}? |
14 "."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }
15 \n     { return EOL; }
16 "//".*
17 [ \t]  { }
18 .      { printf("Mystery character %s\n", yytext); }
19
20 %%
```

Математические операторы здесь описаны единственным символом, а числа заданы с помощью, так называемых, регулярный выражений. При анализе Flex может отслеживать, соответствует ли некоторая последовательность символов заданному регулярному выражению.

Переменная yytext содержит в себе текст текущего токена, поэтому в случае математического оператора мы возвращаем первый символ текста. Для чисел используется функция atof, преобразующая символьное представление числа в формат с плавающей точкой. Сохраняем значение в виде атрибута и возвращаем нетерминал NUMBER, который будет отвечать за численные значения. В случае, если из входного потока пришел \n, вернем нетерминал EOL.

Теперь необходимо определиться с описанием грамматики.

3.2 Построение синтаксического анализатора

3.2.1 Определение грамматики

Определим наиболее интуитивную грамматику для анализа математического выражения. Наименьшей синтаксической единицей определим числа, которые в сочетании с математическими операторами образуют выражения. Кроме того, определим стартовый нетерминал, который должен быть получен в результате анализа. Он, очевидно, должен быть получен с помощью выражения и нетерминала, сигнализирующего о конце строки. В результате получим следующее описание грамматики:

```
// Стартовый нетерминал
calclist:
    | calclist exp EOL {/* действия */};
    }
    | calclist EOL {/* действия */}
;

exp: exp '+' exp {/* действия */}
    | exp '-' exp {/* действия */}
    | exp '*' exp {/* действия */}
    | exp '/' exp {/* действия */}
    | '|' exp {/* действия */}
    | '(' exp ')' {/* действия */}
    | '-' exp %prec UMINUS {/* действия */}
    | NUMBER {/* действия */}
;
```

Однако попытка использовать такую грамматику закончится неудачей.

3.2.2 Разрешение неоднозначностей

Построение грамматики по некоторому языку практически неизбежно приводит к некоторым трудностям — при реализации могут возникнуть неоднозначности, которые полученный анализатор не всегда сможет разрешить ожидаемым образом.

GNU Bison отслеживает такие ситуации и выводит сообщение об ошибке, однако, конечно, он не сможет построить корректный анализатор в случае задания неоднозначной грамматики.

В связи с этим необходимо составить КС-грамматику, сохранив при этом однозначность разбора. Среди наиболее возможных причин неоднозначности грамматики может быть приоритет операторов или их ассоциативность.

Наиболее очевидным решением является задание приоритетности операторов с помощью введения новых нетерминальных символов, однако это достаточно громоздкий метод. GNU Bison предоставляет возможность более удобной работы с ассоциативностью и приоритетом операторов.

Для указания ассоциативности применяются директивы `%left`, `%right` и `%nonassoc`. Приоритетность операторов определяется порядком расположения этих директив [1, 9].

3.2.3 Реализация синтаксического анализатора

Осталось лишь сообщить GNU Bison, каким образом действовать при выполнении того или иного правила вывода. Математические операторы действуют по обычным правилам, поэтому для правила `exp: exp '+' exp` определим результат выражения как `$$ = $1 + $3`, где `$$` — значение, помещаемое на вершину стека, `$1` и `$3` — значения левой и правой части выражений. Аналогичным образом поступим для остальных операторов, а при получении стартового нетерминала выведем результат.

Кроме того, воспользуемся директивой `%union`, в которой определим список возможных значений типов данных для семантических значений [1, 9]. На текущий момент единственным возможным типом данных будет `double`.

Ну и, наконец, необходимо подключить необходимые библиотеки, а также объявить сигнатуры функций `yylex`, `yyparse` и `yyerror` и переменной `yylineno`, отвечающей за обработку символ конца строки

Получим следующее содержимое файла:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
int yyparse();
void yyerror();
int yylineno;
%}

// Указываем возможные типы даннхы для семантических значений
%union {
    double d;
}

// Объявляем токен NUMBER и его тип
%token <d> NUMBER
%token EOL
// Указываем приоритетность
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

// Связываем нетерминал exp с типом double
%type <d> exp

%%

// Указывем правила для стартового нетерминала
calclist:
    | calclist exp EOL { printf("= %f\n", $2);
    printf("> ");
    }
    | calclist EOL { printf("> "); }
;

// Указывем правила для создания выражений
exp:  exp '+' exp {$$ = $1 + $3; }
    | exp '-' exp {$$ = $1 + $3; }
    | exp '*' exp {$$ = $1 * $3; }
    | exp '/' exp {$$ = $1 / $3; }
    | '|' exp {$$ = $2 >= 0? $2 : - $2; }
```

```
42      | '(' exp ')' {$$ = $2; }
43      | '-' exp %prec UMINUS { $$ = -$2; }
44      | NUMBER {$$ = $1; }
45  ;
46
47  %%
48
49  int main(int argc, char** argv) {
50      printf("> ");
51      return yyparse();
52  }
53
54  void yyerror(char* s) {
55      fprintf(stderr, "%d: error: ", yylineno);
56      fprintf(stderr, "\n");
57  }
```

3.2.4 Сборка проекта

Полученный проект необходимо собрать. Процесс сборки состоит из следующих шагов:

1. Необходимо сгенерировать исходный код синтаксических анализатора с помощью консольной команды `bison`.
2. Необходимо сгенерировать исходный код лексического анализатора с включенным в него с помощью `include` кодом синтаксического анализатора с помощью консольной команды `flex`.
3. Полученные файлы связать в единую программу на языке C с помощью консольной команды, выполняющей компиляцию кода на языке C (например, с помощью команды `cc`)

Для удобства разместим эти инструкции в `Makefile` и выполним с помощью системной утилиты `make`. Полученный файл может незначительно различаться в зависимости от целевой платформы, для OS X Big Sur 11.2.3 он выглядит следующим образом:

```
1  calc.out: calc.l calc.y
2          bison -d calc.y
3          flex calc.l
4          cc -o $@ calc.tab.c lex.yy.c -ll
```

Теперь программу можно запустить:

Рисунок 3 – Демонстрация работы программы

4 Абстрактные синтаксические деревья

4.1 Мотивировка

Несмотря на то, что программа работает корректно, в ней присутствует достаточно много недостатков. А именно:

1. Изменение и улучшение программы невозможно без затрагивания ее функциональности.
2. Невозможно повторное использование уже полученных лексическим анализатором данных.
3. Использования возможностей только КС-грамматики может быть недостаточно для решения поставленной задачи.

Эти и некоторые другие задачи позволяет решить концепция абстрактных синтаксических деревьев.

4.2 Определение

Абстрактное синтаксическое дерево (АСД) — конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья — с соответствующими операндами [10].

Рассмотрим выражение $1 + 2 * 3$. Вид синтаксического дерева для такого выражения изображен на рис. 4:

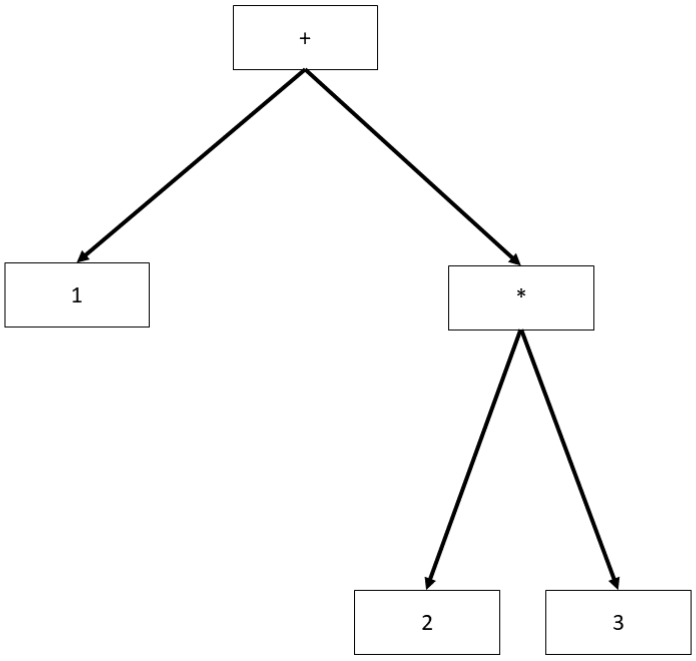

Рисунок 4 – Пример абстрактного синтаксического дерева

Обход такого дерева для получения результата достаточно интуитивен: достаточно обойти дерево начиная с листьев, последовательно вычисляя значения в вершинах, соответствующих математическим операторам.

4.3 Построение

4.3.1 Определение структуры данных

Для построения абстрактного синтаксического дерева в рамках нашей задачи потребуются описание некоторой структуры данных, хранящей наши узлы.

В качестве узла возьмем структуру, хранящую тип вершины и указатели на дочерние элементы. Нам будет вполне достаточно иметь две вершины в качестве дочерних, так как мы имеем дело исключительно с унарными и бинарными операторами.

Таким образом, получим следующий вид структуры:

```
1 typedef struct node {
2     // Тип ячейки
3     int nodetype;
4     // Указатель на "левый" дочерний элемент
5     struct node* l;
6     // Указатель на "правый" дочерний элемент
7     struct node* r;
8 } node;
```

Листья не содержат дочерних элементов, поэтому реализуем отдельную структуру данных под хранение листовых элементов дерева:

```
1 typedef struct value {
2     // Тип ячейки
3     int nodetype;
4     // Численное значение ячейки
5     double number;
6 } value;
```

4.3.2 Построение дерева

Теперь необходимо реализовать функции `newnum` и `newast`, которые будут осуществлять создания узлов для чисел (в рамках дерева представляющих из себя исключительно листья) и для узлов, содержащих операторы.

Получим следующую реализацию функции `newast`:

```
struct node* newast(int nodetype, struct node* l, struct node* r) {
    // Выделяем память стандартной функцией malloc
    struct node* a = malloc(sizeof(struct node));]
    // Проверяем, была ли память выделена
    if (!a) {
        yyerror("Not enough memory");
        exit(0);
    }
    // Инициализируем поля нужными значениями
    a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    return a;
```

Аналогичным образом реализуем функцию `newnum`:

```
struct node* newnum(double d) {
    // Выделяем память стандартной функцией
    struct value* a = malloc(sizeof(struct value));
    // Проверяем, была ли память выделена
    if(!a) {
        yyerror("Not enough memory");
        exit(0);
    }
    // Инициализируем поля
    a->nodetype = 'K';
    a->number = d;
    // Возвращаем приведенный к типу node* указатель
    return (struct node*) a;
}
```

Отметим, что в данном случае результатом функции является приведенный к `node*` указатель на вершину с номером. В дальнейшем мы будем это учитывать, приводя, где необходимо, указатель обратно к указателю на `value*`.

4.3.3 Обход полученного АСД

Заметим, что для вычисления результатов необходимо выполнить обход дерева, получив значение в корне дерева. Для этого реализуем функцию `eval`. Функция будет представлять из себя рекурсивный вызов от левого и правого поддеревьев до тех пор, пока не будет найден лист. Получим следующий код:

```
double eval (arena* arena, struct node* a) {
    double v;
    unsigned int type = a->nodetype;
    node* block = arena->arena;
    // Выполняем действия в зависимости от типа узла
    switch (type) {
    case 'K': {
        value* val = (value*) a;
        v = val->number;
        break;
    }
    case '+':
        v = eval(arena, block + a->l) + eval(arena, block + a->r);
        break;
    case '-':
        v = eval(arena, block + a->l) - eval(arena, block + a->r);
        break;
    case '*':
        v = eval(arena, block + a->l) * eval(arena, block + a->r);
        break;
    case '/':
        v = eval(arena, block + a->l) / eval(arena, block + a->r);
        break;
    case '|':
        v = fabs(eval(arena, block + a->l));
        break;
    case 'M':
        v = -eval(arena, block + a->l);
        break;
    default:
        printf("internal error: bad node %c\n", a->nodetype);
        break;
    }
    return v;
}
```

Запуск такой функции от корня позволит рекурсивно прямым обходом пройти полученное дерево и получить искомый результат.

4.3.4 Освобождение выделенной памяти

Наконец, реализуем функцию `treefree` освобождения памяти нашего абстрактного синтаксического дерева:

```
void treefree(struct node* a) {
    // В зависимости от числа дочерних элементов выполняем рекурсивное
    ↪ освобождение памяти
    if (a->nodetype == '+' || a->nodetype == '-' || a->nodetype == '*' ||
    ↪ a->nodetype == '/') {
        treefree(a->r);
        treefree(a->l);
        free(a);
    }
    else
    if (a->nodetype == '|' || a->nodetype == 'M') {
        treefree(a->l);
        free(a);
    }
    else
    if (a->nodetype == 'K') {
        free(a);
    }
    else {
        printf("internal error: free bad node %c\n", a->nodetype);
    }
}
```

4.3.5 Модификация программы анализатора

Осталось лишь сообщить GNU Bison, каким образом обращаться к нашей структуре данных. Изменим содержимое второго блока и получим следующую реализацию:

```
%{
#include <stdio.h>
#include <stdlib.h>
// Добавим подключение заголовочного файла, содержащего описание нашей
↪ структуры данных
#include "ast.h"
int yylex();
int yyparse();
%}
```

```
9  // Укажем, что одним из возможных значений элемента стека может быть указатель
   ↪  на нашу структуру данных
10 %union {
11     struct node* a;
12     double d;
13 }
14
15 %token <d> NUMBER
16 %token EOL
17
18 %left '+' '-'
19 %left '*' '/'
20 %nonassoc '|' UMINUS
21
22 // Укажем, что нетерминал exp представляет из себя указатель на node*
23 %type <a> exp
24
25 %%
26
27 calclist:
28     // Если дерево построено - необходимо обойти его и вывести результат,
       ↪  после чего освободить память
29     | calclist exp EOL { printf("= %f\n", eval($2));
30     treefree($2);
31     printf("> ");
32     }
33     | calclist EOL { printf("> "); }
34 ;
35 // Иначе если мы применяем одно из правил - построим новый узел в соответствии
   ↪  с правилом
36 exp: exp '+' exp {$$ = newast('+', $1, $3); }
37     | exp '-' exp {$$ = newast('-', $1, $3); }
38     | exp '*' exp {$$ = newast('*', $1, $3); }
39     | exp '/' exp {$$ = newast('/', $1, $3); }
40     | '|' exp {$$ = newast('|', $2, NULL); }
41     | '(' exp ')' {$$ = $2; }
42     | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
43     | NUMBER {printf("%c", $1);$$ = newnum($1); }
44 ;
45
46 %%
```

```
47
48 int main(int argc, char** argv) {
49     printf("> ");
50     return yyparse();
51 }
52
53 void yyerror(char* s) {
54     fprintf(stderr, "%d: error: ", yylineno);
55     fprintf(stderr, "\n");
56 }
```

Полученную программу можно собрать, воспользовавшись вызовом `make` над модифицированным `Makefile`:

```
1     calc.out: calc.l calc.y ast.h
2         bison -d calc.y
3         flex calc.l
4         cc -o $@ calc.tab.c lex.yy.c ast.c
```

после чего запустить.

Результат работы программы представлен на рис. 5:

Рисунок 5 – Демонстрация работы программы

Как мы позже увидим, текущая версия значительно производительнее версии без абстрактного синтаксического дерева.

4.4 Управление памятью на основе регионов

4.4.1 Мотивировка

Текущая реализация абстрактного синтаксического дерева имеет следующие недостатки:

1. Выделение памяти стандартным методом может значительно фрагментировать оперативную память, затрудняя доступ к ней.

2. Любое выделение и удаление памяти требует вмешательства системных вызовов, что может стать причиной дополнительных издержек во время работы программы.
3. Программист не имеет возможности ручного управления выделяемой им памятью.

Избавиться от этих недостатков можно используя различные оптимизации. В рамках этой работы воспользуемся управлением памятью на основе, так называемых, регионов (арен, зон) [11].

Под регионом далее будем понимать непрерывную область памяти, содержащую внутри себя объекты. При запуске программы выделим регион некоторого размера, при необходимости увеличивая его размер в некоторое постоянное число раз.

Этот подход имеет следующие преимущества:

1. Элементы располагаются последовательно, в связи с чем минимизируется фрагментация и упрощается доступ к объектам.
2. Выделение и освобождение памяти выполняется с минимальными издержками.
3. Программисту предоставляется большая свобода для управления выделенной памятью.

4.4.2 Построение

Формально определим требования к системе:

1. Регион должен представлять из себя некоторый непрерывный участок размера n байт (в начальный момент времени размер равен некоторой начальной величине n_0).
2. При обращении к региону он должен предоставить k байт памяти и вернуть некоторый идентификатор этого участка для последующего обращения.
3. При заполнении региона должна быть возможность увеличить объем доступной памяти в некоторое число раз, которое далее будем называть коэффициентом увеличения.
4. Должна быть доступна возможность эффективного освобождения всей выделенной регионом памяти.

Единственной сложной операцией над регионом является его увеличение. Так как выделение нового участка потенциально может сопровождаться изменением адресов объектов, то необходимо организовать доступ к ним независимо

от первоначального адреса. Для этого для каждого объекта будем получать доступ к нему через некоторый индекс.

Кроме того, коэффициент увеличения должен быть выбран таким образом, чтобы был соблюден баланс между оптимальным объемом выделенной памяти и частотой системных вызовов.

4.4.3 Определение структуры

Определим нашу структуру следующим образом:

```
typedef struct arena {
    // Указатель на начало региона
    struct node* arena;
    // Размер региона
    unsigned int size;
    // Объем выделенной регионом памяти
    unsigned int allocated;
} arena;
```

4.4.4 Инициализация

Теперь определим функцию `arena_construct`, выполняющую начальную инициализацию состояния региона:

```
int arena_construct (arena* arena) {
    // Начальный размер региона равен некоторой постоянной, равной
    ↪ DEFAULT_ARENA_SIZE
    arena->size = DEFAULT_ARENA_SIZE;
    arena->allocated = 0;
    // Выделим необходимое число памяти
    arena->arena = malloc(sizeof(node) * DEFAULT_ARENA_SIZE);
    // Если выделение прошло неудачно - вернем в качестве кода ошибки отличное
    ↪ от 0 значение.
    if (arena->arena == NULL) {
        return (!0);
    }
    return 0;
}
```

4.4.5 Выделение памяти

После выделения некоторого объема памяти возможно обращение к ней. Определим это обращение с помощью функции `arena_allocate`:

```
int arena_allocate (arena* arena, unsigned int count) {
    // Если места в регионе недостаточно
    if (arena->allocated + count >= arena->size) {
        // Определим новый размер региона
        unsigned int newSize = MULTIPLY_FACTOR * arena->size;
        // Выделим регион большего размера и освободим ранее занятую память
        node* newArena = realloc(arena->arena,
            newSize  * sizeof(node));
        if (NULL == newArena) {
            return -1;
        }
        arena->arena = newArena;
        arena->size = newSize;
    }
    // В качестве результата вернем индекс первого свободного участка региона
    unsigned int result = arena->allocated;
    // Сместим индекс на объем выделенной памяти
    arena->allocated += count;
    // Вернем результат
    return result;
}
```

Отметим, что наиболее часто значением `MULTIPLY_FACTOR` оказывается числа 1.5 и 2. Это позволяет достичь амортизационно константного времени выполнения операции выделения памяти [12].

4.4.6 Освобождение выделенной памяти

Наконец, реализуем освобождение выделенной региону памяти с помощью функции `arena_free`

```
void arena_free (arena* arena) {
    if (arena->arena != NULL)
        free(arena->arena);
    arena->arena = NULL;
}
```

4.4.7 Модификация абстрактного синтаксического дерева

Осталось изменить исходный код программы, чтобы обеспечить выделение памяти с помощью полученной нами структуры данных.

Для этого воспользуемся директивой `%param` и заявим в качестве параметра переменную типа `arena*`. В функциях `eval`, `newnum`, `newast` внесем из-

менения, чтобы обеспечить выделение памятью с помощью написанных ранее функций.

С полным кодом программы можно ознакомиться в приложении А.

4.4.8 Сборка проекта

Теперь проект можно собрать, незначительно изменив `Makefile`:

```
calc.out: calc.l calc.y arena_ast.h
	bison -d calc.y
	flex calc.l
	cc -o $@ calc.tab.c lex.yy.c arena_ast.c arena.c
```

и запустить. Результат работы программы представлен на рис. 6

Рисунок 6 – Демонстрация работы программы

5 Сравнение полученных реализаций

Проведем анализ производительности полученных версий анализатора. В качестве данных для тестирования возьмем выражения вида $\underbrace{2+2+2\cdots+2}_{n}$ для $n = 1\ldots100$ с шагом 1. Для вычисления времени выполнения воспользуемся библиотекой `time` Python 3.9.5. Автоматизацию обеспечим с помощью библиотеки `subprocess`. Получим следующий код:

```
import subprocess as sb
import time
import sys
def run(args):
    return sb.run(args,
        capture_output=True, ).stdout.decode().strip()

def main():
    if (len(sys.argv) < 6):
        print("Invalid arguments")
        return
    exe_path = sys.argv[1]
    out_path = sys.argv[2]
    right_bound = int(sys.argv[3])
    step = int(sys.argv[4])
    iter = sys.argv[5]

    f = open(out_path, "w")
    f.write(f"{exe_path}\n")
    f.close()
    for expr_len in range(1, right_bound, step):
        test_string = "+".join(['2'] * expr_len)
        args = [exe_path, test_string, iter]
        t = time.monotonic()
        run(args)
        end_t = time.monotonic()
        f = open(out_path, "a")
        f.write(f"{expr_len} {(end_t - t) / (int(iter))}\n")
        f.close()
        print(f" Step {expr_len} finished")

if __name__ == "__main__":
    main()
```

Кроме того, отметим, что в ранее написанные программы были внесены некоторые изменения для проведения эксперимента. Ознакомиться с ними можно в приложении А.

Ознакомиться с полным исходным кодом программы, осуществляющей исследование производительности можно в приложении Б.

Для большей наглядности графики интерполированы полиномом с помощью функции `polyfit` библиотеки `numpy`.

Ознакомиться с полным исходным кодом программы, осуществляющей анализ полученных результатов можно в приложении В.

Результаты исследования изображены на рис. 7:

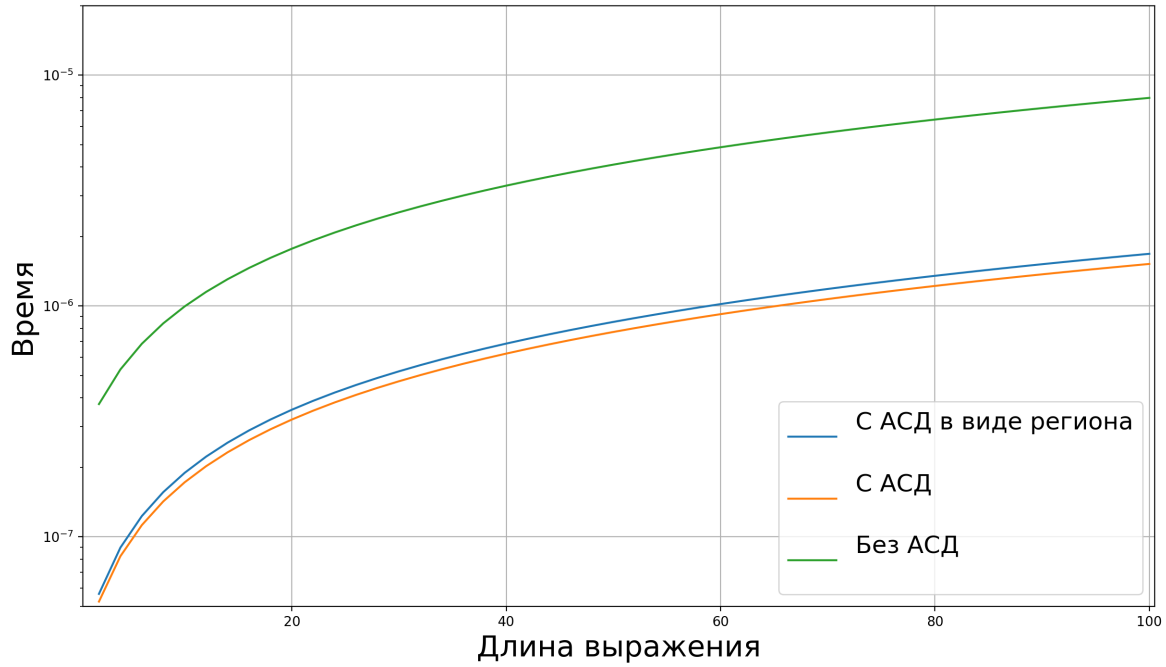

Рисунок 7 – Сравнение полученных результатов

Исследование показало, что использование абстрактных синтаксических деревьев позволяет уменьшить время работы программы более чем в 5 раз, что существенно заметно для выражений любой длины.

Также из графиков видно, что в рамках данной работы не удалось добиться большей производительности при управлении памятью на основе регионов. Тем не менее, она все еще может считаться более предпочительной ввиду перечисленных ранее преимуществ.

ЗАКЛЮЧЕНИЕ

В ходе данной работы:

1. Были изучены теоретические основы построения лексических и синтаксических анализаторов.
2. Проанализированы особенности реализации лексический и синтаксических анализаторов.
3. Были изучены принципы работы генераторов лексического и синтаксического анализа на примере Flex и GNU Bison.
4. Были созданы лексический и синтаксический анализаторы для анализа математического выражения.
5. Было изучено понятие абстрактного синтаксического дерева.
6. Проведен анализ производительности полученных реализаций.

Таким образом, все поставленные в рамках работы задачи выполнены.

Результаты исследования показали, что абстрактные синтаксические деревья позволяют добиться увеличения производительности в 5–6 раз.

А это, в свою очередь, позволяет утверждать о том, что концепция абстрактных синтаксических деревьев является крайне важной в информатике и ее приложениях, в частности, при создании синтаксических анализаторов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Levine, J.* flex & bison: Text Processing Tools / J. Levine. — O'Reilly Media, Inc., 2009.

2 *Aho, A. V.* Compilers, Principles, Techniques, and Tools / A. V. Aho, R. Sethi, J. D. Ullman, M. S. Lam. — Addison-Wesley, 1986.

3 О применении автоматов при реализации алгоритмов дискретной математики (на примере АВЛ-деревьев) [Электронный ресурс]. — URL: `http://is.ifmo.ru/works/_avl.pdf` (Дата обращения 25.05.2021). Загл. с экр. Яз. рус.

4 *Пентус, А. Е.* Теория формальных языков / А. Е. Пентус, М. Р. Пентус. — Москва: МЦНМО, 2013.

5 Методы трансляций, конспект лекций факультета ПМИ НГТУ [Электронный ресурс]. — URL: `http://window.edu.ru/resource/262/29262/files/nstu02.pdf/` (Дата обращения 19.05.2021). Загл. с экр. Яз. рус.

6 *Hopcroft, J. E.* Introduction to Automata Theory, Languages, and Computation / J. E. Hopcroft, J. D. Ullman. — Addison-Wesley, 1979.

7 *Grune, D.* Parsing Techniques / D. Grune, C. J. Jacobs. — New York: Springer-Verlag, 2008.

8 *Svenheim, G.* Implementation of an LALR(1) parser generator. — Rochester Institute of Technology, New York: 1980. — Pp. 11–12.

9 GNU Bison Manual [Электронный ресурс]. — URL: `https://www.gnu.org/software/bison/manual/html_node/index.html#SEC_Contents` (Дата обращения 24.05.2021). Загл. с экр. Яз. англ.

10 *Rabinovich, M.* Abstract syntax networks for code generation and semantic parsing // Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). — Vancouver, Canada: Association for Computational Linguistics, 2017. — Pp. 1140–1141. `https://www.aclweb.org/anthology/P17-1105`.

11 *Wang, D. C.-A.* Managing memory with types / D. C.-A. Wang. — Princeton University, 2002. — Pp. 29–55.

12 Facebook open-source library documentation [Электронный ресурс]. — URL: https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md (Дата обращения 25.05.2021). Загл. с экр. Яз. англ.

ПРИЛОЖЕНИЕ А

Flash-носитель с исходным кодом программ, использующихся в работе

Папка `src` содержит оригинальный исходный код программы:

Папка `naive` — реализация без АСД

Папка `naiveast` — реализация с АСД

Папка `arena` — реализация с АСД на основе региона

Папка `extsrc` содержит измененный исходный код, необходимый для исследования производительности:

Папка `naive` — реализация без АСД

Папка `naiveast` — реализация с АСД

Папка `arena` — реализация с АСД на основе региона

ПРИЛОЖЕНИЕ Б

Исходный код программы на Python, осуществляющей исследование производительности полученных реализаций

```
import subprocess as sb
import time
import sys
def run(args):
    return sb.run(args,
        capture_output=True, ).stdout.decode().strip()

def main():
    if (len(sys.argv) < 6):
        print("Invalid arguments")
        return
    exe_path = sys.argv[1]
    out_path = sys.argv[2]
    right_bound = int(sys.argv[3])
    step = int(sys.argv[4])
    iter = sys.argv[5]

    f = open(out_path, "w")
    f.write(f"{exe_path} \n ")
    f.close()
    for expr_len in range(1, right_bound, step):
        test_string = "+".join(['2'] * expr_len)
        args = [exe_path, test_string, iter]
        t = time.monotonic()
        run(args)
        end_t = time.monotonic()
        f = open(out_path, "a")
        f.write(f"{expr_len} {(end_t - t) / (int(iter))} \n ")
        f.close()
        print(f" Step {expr_len} finished")

if __name__ == "__main__":
    main()
```

ПРИЛОЖЕНИЕ В

Исходный код программы на Python, осуществляющей анализ полученных результатов

```
import subprocess as sb
from time import time
import matplotlib.pyplot as plt
import sys
import numpy as np

legend = []
for index in range(1, len(sys.argv)):
    file_name = sys.argv[index]
    f = open(file_name, "r")
    try:
        parser_type = f.readline()
    except StopIteration:
        parser_type = "Undefined parser"
    legend.append(parser_type)

    x_axis = []
    y_axis = []
    for line in f:
        try:
            w, h = [float(x) for x in next(f).split()]
        except StopIteration:
            break
        x_axis.append(w)
        y_axis.append(h)
    plt.xlabel("Длина выражения", size = 22)
    plt.ylabel("Время", size = 22)
    p = np.polyfit(x_axis, y_axis, 1)
    for i in range(len(x_axis)):
       y_axis[i] = p[0] * x_axis[i] + p[1]

    plt.plot(x_axis, y_axis)
plt.xlim(0.5, 100.5)
plt.ylim(5e-8, 2e-5)
plt.rcParams.update({'font.size': 18})
#plt.yscale("log")
plt.grid(True)
plt.legend(legend, loc="lower right")
```

```
39 plt.show()
```