Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)

Факультет Информатики и систем управления

Кафедра «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к курсовой работе на тему

«Переписывание MillDB на скриптовый язык»

по дисциплине «Конструирование компиляторов»

Студент ИУ9-72Б _____ Д. С. Чугунов

(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ А. В. Коновалов

(Подпись, дата) (И. О. Фамилия)

2020 г.

Оглавление

Введение

Современный мир невозможно представить без баз данных. Базы данных охватили практически все сферы деятельности человека. Не существует универсальной базы данных, способной удовлетворять всем потребностям пользователей. Так, например, локальные базы данных в телефоне не могут обрабатывать большие объемы данных, в отличие от современных «клиент-серверных» баз данных, но при этом мобильные базы данных не требовательны к ресурсам. Поэтому разработка новых баз данных всегда актуальна.

Одним из классов баз данных являются OLAP системы, которые позволяют хранить и обрабатывать большие объемы исторических данных. Такие системы получили широкое распространение в различных областях, где необходимо обрабатывать данные за длительный промежуток времени. Для более эффективной работы некоторые базы данных генерируются под заранее заданное множество операций. Одним из примеров является генератор высокопроизводительных OLAP баз данных MillDB[1], разработанный в рамках дипломной работы на кафедре «Теоретической информатике и компьютерных технологий».

Генератор MillDB производит трансляцию базы данных, описанную на DDL SQL, в библиотеку, написанную на языке C, способную обрабатывать необходимые запросы к базе данных. Другими словами, генератор является компилятором, переводящим язык SQL в язык C. В качестве языка реализации в компиляторе на текущий момент используется связка C++, Flex[2] и Bison[3]. Главным недостатком такого решения является сложность поддержки и улучшения кода.

Целями данной курсовой работы являются перенос компилятора на более высокоуровневый язык и интеграция новых функциональностей (расширенный язык выражений, дополнительные индексы и фильтр Блума).

1. Обзор MillDB

MillDB – генератор высокопроизводительной «write-only» OLAP базы данных. Под «write-only» понимается отсутствие операций изменения данных (INSERT), а также удаления (DELETE). Базы данных генерируются на языке C, благодаря этому они имеют большую производительность.

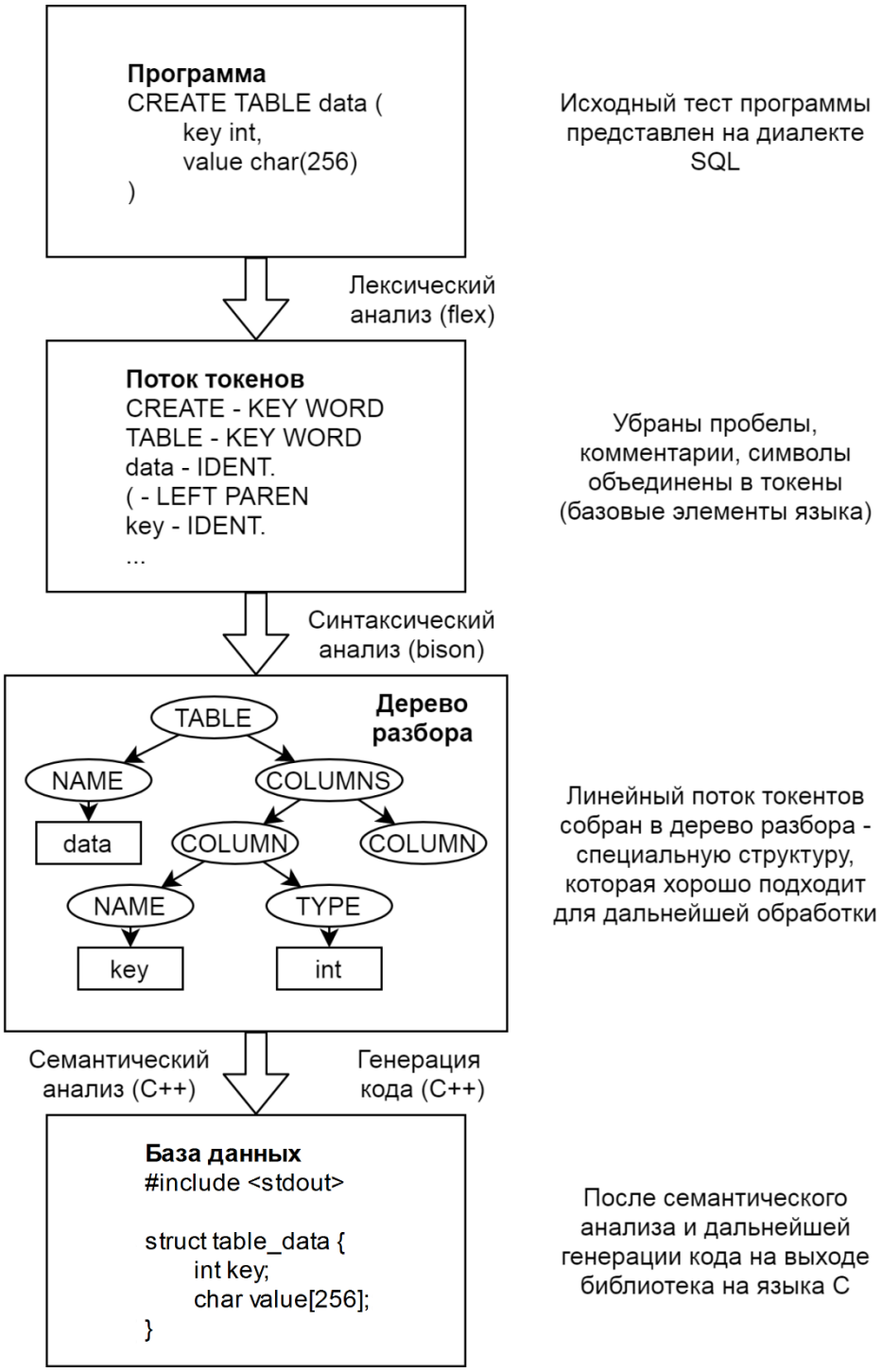

Рисунок 1 – Процесс генерации базы данных

1.1. Генерация базы данных

Основные компоненты генерации базы данных представлены на рисунке 1. Для лексического анализа использовался генератор лексических анализаторов – Flex (Fast Lexical Analyzer), а для синтаксического анализа – Bison. Использование этих утилит позволяет сократить время разработки лексера и парсера, а также позволяет легко вносить новые изменения.

Семантический анализ и генерация кода происходит при помощи языка C++. Код генерируется при помощи потоков ввода/вывода из стандартной библиотеки iostream. Данная библиотека не позволяет делать шаблонизируемые строки и поэтому читаемость кода генератора очень низкая. Для примера на листинге 1 представлен пример генератора небольшого фрагмента кода, а также сам сгенерированный код.

Листинг 1 – пример генерации кода

```
#include <iostream>
// Код для генерации
int main() {
    char table_name[] = "data";

    char key_type[] = "int";
    char key_name[] = "key";

    char value_type[] = "char*";
    char value_name[] = "value";

    std::cout << "struct " << table_name << " {" << std::endl;
    std::cout << "\t" << key_type << " " key_name << ";" << std::endl;
    std::cout << "\t" << value_type << " " << value_name << ";" << std::endl;
    std::cout << "}" << std::endl;

    return 0;
}
// Сгенерированный код
struct data {
    int key;
    char* value;
}
```

Как видно на примере, даже для маленького фрагмента становится сложно разобрать код для генерации. Такой генератор сложно поддерживать и улучшать в дальнейшем.

1.2. Синтаксис диалекта SQL

Грамматика диалекта SQL представлена на листинге 2. Регулярные выражения для терминалов id (идентификатор), pid (параметр), integer (целое положительное число), float (вещественное положительное число) представлены на листинге 3.

Листинг 2 – грамматика SQL диалекта в MillDB в форме РБНФ

```
non-terminal: PROG, E_PROG, TAB, COLS, COL, TYPE, PROC, PARAMS,
              PARAM, STATS, STAT, INS, ARGS, ARG, SEL, SCOLS,
              SCOL, STABS, CONDS, COND, OP ;
terminal: '(', ')', ';', ',', '=', '<', '>', '<>', '<=', '>=',
          'TABLE', 'JOIN', 'SEQUENCE', 'NEXTVAL', 'CURRVAL',
          'CREATE', 'PK', 'BLOOM', 'INDEXED', 'SELECT', 'FROM',
          'WHERE', 'INSERT', 'VALUES', 'PROCEDURE', 'BEGIN',
          'END', 'IN', 'OUT', 'SET', 'INDEX', 'ON', 'AND', 'OR',
          'NOT', 'INT', 'FLOAT', 'DOUBLE', 'CHAR',
          id, pid, integer, float ;

PROG   ::= E_PROG+ ;
E_PROG ::= 'CREATE' (TAB | PROC | SEQ) ;

TAB    ::= 'TABLE' id '(' COLS ')' ';' ;
COLS   ::= COL (',' COL)* ;
COL    ::= id TYPE (('PK' | 'INDEXED' | 'BLOOM') float?)? ;
TYPE   ::= ('INT' | 'FLOAT' | 'DOUBLE' | 'CHAR') ('(' integer ')')? ;

PROC   ::= 'PROCEDURE' id '(' PARAMS ')' 'BEGIN' STATS 'END' ';' ;
PARAMS ::= PARAM (',' PARAM)* ;
PARAM  ::= pid TYPE ('IN' | 'OUT') ;
STATS  ::= STAT+ ;
STAT   ::= INS | SEL ;

INS    ::= 'INSERT' 'TABLE' id 'VALUES' '(' ARGS ')' ';' ;
ARGS   ::= ARG (',' ARG)* ;
ARG    ::= pid | ('CURRVAL' | 'NEXTVAL') '(' id ')' ;

SEL    ::= 'SELECT' SCOLS 'FROM' STABS 'WHERE' CONDS ';' ;
SCOLS  ::= SCOL (',' SCOL)* ;
SCOL   ::= id 'SET' pid ;
STABS  ::= id ('JOIN' id)* ;
CONDS  ::= COND (('OR' | 'AND') COND)* ;
COND   ::= '(' CONDS ')' | 'NOT' COND | id OP (id | pid) ;
OP     ::= '<' | '>' | '<=' | '>=' | '=' | '<>' ;

SEQ    ::= 'SEQUENCE' id ';' ;

axiom PROG ;
```

Листинг 3 – регулярные выражения для терминалов

```
id      = [a-zA-Z][a-zA-Z0-9_]*
pid     = @[a-zA-Z][a-zA-Z0-9_]*
integer = [0-9]*
float   = [0-9]*\.[0-9]+ | [0-9]+\.[0-9]*
```

2. Интеграция новых функциональностей

В курсовых проектах были реализованы новые функциональности. Первая – расширение логических операций в операторе SELECT[4], а вторая – добавление дополнительных индексов, а также фильтра Блума[5]. Данные изменения разрабатывались независимо друг от друга, поэтому одной из задач данной курсовой работы было объединение этих двух функциональностей.

Основной сложностью интеграции является измененная структура класса колонок (в реализации дополнительных индексов и фильтра Блума). На рисунке 2 представлены отличия в двух реализациях.

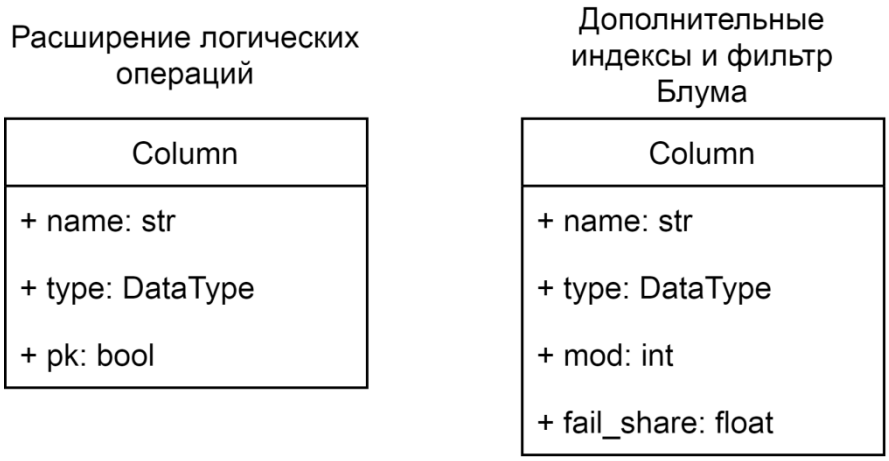

Рисунок 2 – различие структуры класса колонок

В реализации дополнительных индексов добавились новые типы колонок. Если раньше колонка могла быть обычной, либо представлять собой первичный ключ, то теперь для колонки может быть построен дополнительный индекс или фильтр Блума. За тип колонки отвечает поле mod: 0 – обычная колонка, 1 – колонка с фильтром Блума, 2 – индексируемая колонка, 3 – колонка первичного ключа. Такая замена повлияла на участки кода, где проверяется наличие первичного ключа, для таких участков была произведена замена column.pk на column.mod == 3. Остальные конфликты разрешались при помощи merge – стандартной утилиты системы контроля версий git.

2.1. Тестирование интеграции новой функциональности

Для проверки новой функциональности был написан тест, код которого можно увидеть на листингах 4 (код SQL) и 5 (тест).

Листинг 4 – DDL SQL базы данных

```
create table test (
   id int pk,
   val int bloom
);

create procedure add(@id int in, @val int in)
begin
   insert table test values (@id, @val);
end;

create procedure get(@val int in, @oid int out, @oval int out)
begin
   select id set @oid, val set @oval from test where val=@val or not val<=@val;
end;
```

Листинг 5 – тестирование новой функциональности

```
#include <stdio.h>
#include <time.h>
#include "mydb.c"
#include "mydb.h"

#define FILE_NAME "mydb.db"

int main() {
   mydb_open_write(FILE_NAME);
   for (int i = 0; i < 100000; i++) {
      for (int j = 0; j < 10; j++) {
         add(i * 10 + j, j);
      }
   }
   mydb_close_write();
   struct mydb_handle *handle = mydb_open_read(FILE_NAME);

   int check[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   int count = 0;
   struct timespec time_start, time_end;

   struct get_out iter1;
   clock_gettime(CLOCK_REALTIME, &time_start);
   get_init(&iter1, handle, 7);
   while (get_next(&iter1)) {
      check[iter1.data.oval]++;
      count++;
   }
   clock_gettime(CLOCK_REALTIME, &time_end);
   printf("Check 7: %d\n", check[7]);
   printf("Check 8: %d\n", check[8]);
   printf("Check 9: %d\n", check[9]);
   printf("Count: %d ", count);
```

```
    printf("Time: %d\n", time_end.tv_nsec - time_start.tv_nsec);

    mydb_close_read(handle);
    return 0;
}
```

Результат работы программы представлен на листинге 6.

Листинг 6 – результаты тестирования

```
Check 7: 100000
Check 8: 100000
Check 9: 100000
Count: 300000
Time: 10024187
```

В данный тест включены, как и расширенные логические операции, так и фильтр Блума. Как видно из результатов тестирование, интеграцию функциональности можно считать успешной.

3. Разработка генератора MillDB на скриптовом языке

Для эффективного использования ресурсов было решено объединить фазу лексического анализа и фазу синтаксического анализа, а также фазу семантического анализа. Компиляторы, объединяющие фазы, называются однопроходными. Среди преимуществ однопроходных компиляторов стоит выделить низкие требования к памяти, высокую скорость работы, а также правильный порядок вывода ошибок. Среди недостатков стоит выделить сложность разработки, а также негативное влияние на исходный язык (так, если в программе используется таблица до ее определения, то будет выведена ошибка).

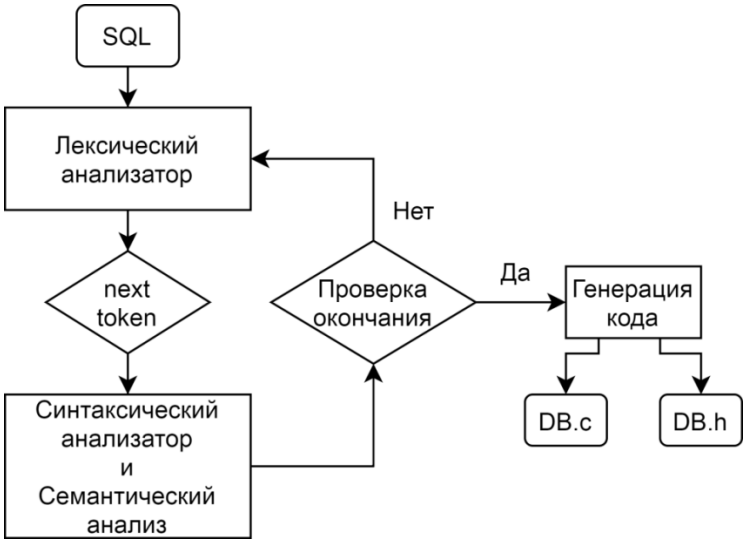

Рисунок 3 – схема работы генератора

Из грамматики диалекта SQL (представленной на листинге 1) видно, что для выбора нового пути разбора необходима только одна лексема. Поэтому в качестве синтаксического анализатора был выбран LL(1) анализатор (сверху-вниз). Стоит отметить, что в Bison используются LR анализаторы (снизу-вверх), но такие анализаторы сложны в самостоятельной реализации. Для синтаксического анализа было решено не использовать готовые генераторы парсеров, так как в дальнейшем это может стать узким местом, когда функциональности генератора будет не хватать. Также генераторы

синтаксических анализаторов не могут сгенерировать парсер способный восстанавливаться при ошибках. На рисунке 3 изображена новая схема генератора MillDB.

3.1. Лексический анализатор

В грамматике MillDB можно выделить 6 основных доменов: идентификатор (id), параметр (pid), целое положительное число (integer), вещественное положительное число (float), ключевые слова и спецсимволы. По грамматике диалекта SQL (листинг 2) и регулярным выражениям (листинг 3) можно построить конечный детерминированный автомат, который представлен на рисунке 4 (для краткости в автомате опущены переходы по пробельным символам в начальное состояние, а также в состояние ошибки по всем другим символам).

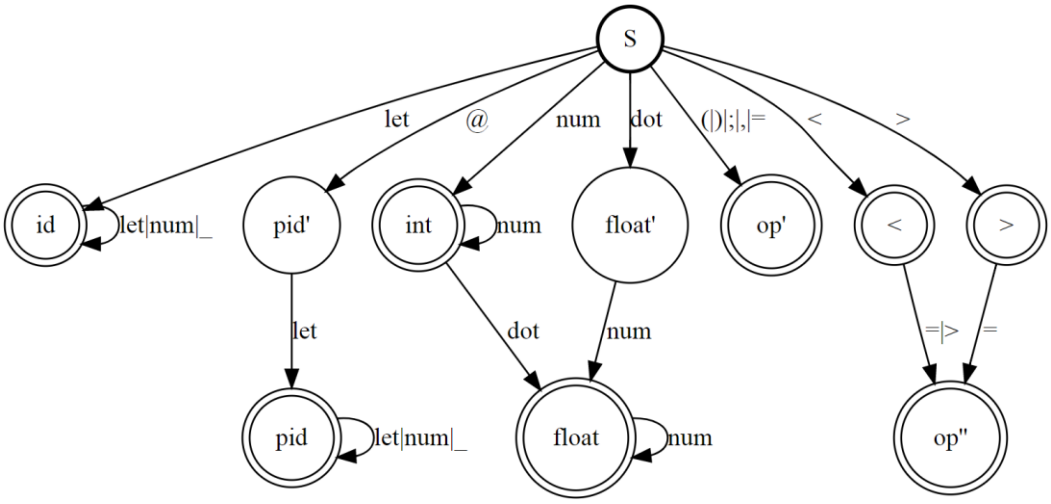

Рисунок 4 – конечный автомат лексического распознавателя

В данном автомате let – это заглавные и строчные латинские буквы, dot – точка, num – цифры. S – начальное состояние. Состояния op’ и op” соответствуют спецсимволам длины один (за исключением символов < и >, которые имеют собственные конечные состояния) и спецсимволам длины два соответственно. Состояние pid – параметр, id – идентификатор, int – целое положительное число, float – целое вещественное число.

Как видно из конечного автомата, для ключевых слов нет собственного конечного состояния. В случае если бы каждое ключевое слово имело бы собственное ключевое состояние, сложность автомата возросла бы в разы. Так как домен ключевых слов вложен в домен идентификаторов, то после распознавания идентификатора происходит проверка на соответствие ключевому слову.

4. Реализация генератора MillDB на скриптовом языке

В качестве языка реализации был выбран Python. Выбор пал именно на этот язык по нескольким причинам:

1) Простота поддержки кода
2) Обилие библиотек, реализующих текстовые шаблонизаторы
3) Python предустановлен практически на все современные Unix системы, что позволит использовать генератор MillDB без установки дополнительных утилит.

4.1. Шаблонизатор Jinja2

Как и было сказано ранее, Python имеет множество библиотек, реализующих текстовые шаблонизаторы. Для дальнейшей разработки в качестве шаблонизатора был выбрана библиотека Jinja2[6].

Jinja2 – это полнофункциональный текстовый шаблонизатор для Python. Он подобен шаблонизатору Django, но предоставляет Python-подобные выражения, обеспечивая исполнение шаблонов в песочнице. Общая схема работы представлена на рисунке 5.

Использование текстового шаблонизатора позволяет улучшить читабельность кода. На листингах 7 и 8 представлены программы, генерирующие один и тот же код (листинг 9), но при этом одна программа написана на C++ без использования шаблонизатора, а другая на языке Python с использованием шаблонизатора Jinja2.

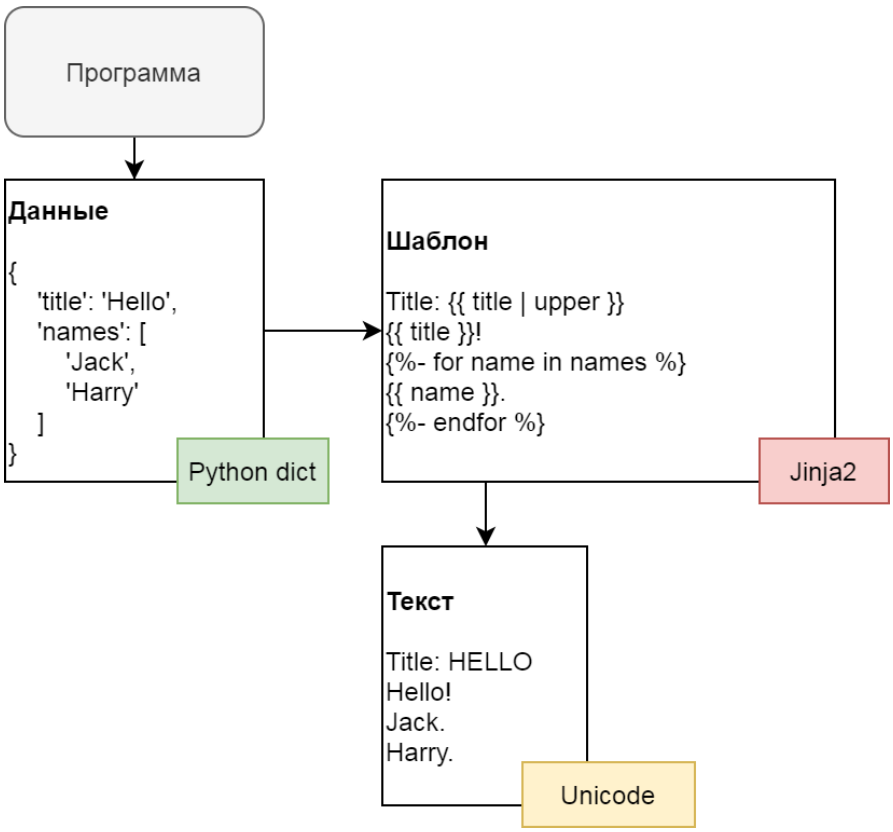

Рисунок 5 – схема работы шаблонизатора Jinja2

Как видно из примеров, код на языке Python более разборчив, чем код на языке C++. В шаблон гораздо легче вносить новые изменения, в отличие от потока вывода iostream в C++.

Листинг 7 – генерация кода с помощью C++ без использования шаблонизатора

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
  std::vector<std::string> cols = {"col1", "col2", "col3"};
  std::string name = "main";
  std::cout << "void " << name << "(";
  for (std::string col : cols) {
    std::cout << "int " << col << ", ";
  }
  std::cout << "col4) {" << std::endl;
  std::cout << "\t// pass" << std::endl;
  std::cout << "}" << std::endl;
  return 0;
}
```

Листинг 8 – генерация кода с помощью Python с использованием шаблонизатора Jinja2

```
import jinja2

def main():
    st = """
void {{ name }}({% for col in cols %}{{ col }}, {% endfor %}col4) {
    // pass
}
    """
    tmp = jinja2.Template(st)
    print(tmp.render(name='main', cols=['col1', 'col2', 'col3']))
```

Листинг 9 – сгенерированный код

```
void main(col1, col2, col3, col4) {
    // pass
}
```

4.1.1. Основные элементы Jinja2

- Выражения.

Выражение в Jinja2 выделяются с помощью двойных фигурных скобок. В выражениях могут быть любые действия над объектами Python. Пример:

```
'1 + 1 = {{ 1 + 1 }}' -> '1 + 1 = 2'
```

- Условные выражения

Условные выражения должны быть заключены между {% if … %} и {% endif %}, также возможны дополнительные блоки, такие как {% elif … %} и {% else %}. Логика использования такая же, как и в условных выражениях в Python, за исключением того, что всякий {% if … %} должен завершаться {% endif %}. Пример:

```
'{% if 1 + 1 > 0 %}1+1>0{% else %}1+1<=0{% endif %}' -> '1+1>0'
```

- Циклы for

Циклы for, также как и условные выражения, должны быть заключены между {% for item in items %} и {% endfor %}. Пример:

```
'{% for i in range(3) %} {{ i }};{% endfor %}' -> ' 0; 1; 2;'
```

- Создание локальных переменных

Jinja2 поддерживает создание локальных переменных прямо внутри шаблона. Эти переменные видны только внутри шаблона и не могут быть получены извне. Создание локальной переменной происходит при помощи выражения {% set val = value %}. Пример:

```
'{% set a = 1 + 1 %}1 + 1 = {{ a }}' -> '1 + 1 = 2'
```

4.2. Лексический анализатор

Для обеспечения однопроходной компиляции лексический анализатор должен располагать методом next_token, который начинает разбор следующей лексемы только при вызове. Язык Python обладает большими возможностями для «ленивых» вычислений с помощью функций-генераторов и ключевого слова yield.

Основой лексического анализатора стал генератор, который, итерируясь по входной последовательности символов программы, последовательно возвращает токены. Каждый токен представляет собой тройку объектов: домен токена, значение токена и «сырое» значение токена (необработанный фрагмент программы соответствующий данному токену). Для упрощения дальнейшей работы, каждый спецсимвол и каждое ключевое слово имеют собственный домен. Для ключевых слов домен – это само ключевое слово заглавными буквами, а для спецсимволов – название.

Каждый домен имеет собственную интерпретацию значения. Для идентификаторов значение – «сырое» значение токена. Для параметра – «сырое» значение токена без символа @. Для ключевых слов – «сырое» значение строчными символами. Для int и float – «сырое» значение приведенному к нужному типу. Для ключевых слов – само ключевое слово строчными буквами. Для спецсимволов – собственный уникальный домен.

4.3. Синтаксический анализатор

Грамматика диалекта SQL – LL(1) и хорошо описывается с помощью РБНФ, поэтому синтаксический анализатор написан с помощью рекурсивного спуска. Небольшая глубина дерева разбора позволяет не беспокоится за выходы за пределы лимита числа рекурсий.

Парсер представляет собой объект с двумя атрибутами: объект токена с методом next и структура, которая является аналогом дерева разбора и заполняется по ходу синтаксического анализа.

После полного синтаксического анализа происходит проверка на наличие ошибок стадий лексического, синтаксического и семантического анализов. В случае наличия ошибок генерация файлов не производится и осуществляется выход из программы, иначе производится генерация файлов.

4.4. Обработка ошибок

Для вывода ошибок использовалась стандартная библиотека Python – logging. Данная библиотека имеет обширную функциональность, в том числе позволяет настраивать тип вывода (в файл, в стандартный поток вывода и пр.), а также настраивать уровни вывода сообщений (DEBUG, WARN, …).

Для упрощения работы ошибки сразу выводятся на экран (или в файл) при этом, не сохраняясь в программе, поэтому важно иметь какой-нибудь «флаг», который несет информацию о падении. Если такого флага не будет, то возможна генерация неправильного файла библиотеки MillDB, так как программа не будет знать о наличие ошибок. Стандартный класс Logger из библиотеки logging не располагает таким флагом, поэтому был написан наследник данного класса, который имеет статическую переменную, информирующую о наличии ошибок. Каждый объект класса изменяет данную статическую переменную, в случае вывода ошибок уровня ERROR. Помимо

статической переменной, в классе был реализован автоматический вывод позиций ошибок.

4.5. Установка и использование компилятора MillDB

Важной частью каждой программы простота установки, а также легкость использования. Компилятор MillDB, написанный на языке Python, был оформлен в виде библиотеки, поэтому для его установки можно использовать различные стандартные средства установки. Наиболее удобным является способ установки через систему управления пакетами pip. Пример:

```
pip install git+https://github.com/bmstu-iu9/mill-db.git
```

В данном примере представлен способ установки из github. Помимо установки через pip можно воспользоваться стандартной установкой пакетов через setup.py, для этого необходимо скачать библиотеку и в ее корне выполнить команду:

```
python3 setup.py install
```

Данные команды установят библиотеку pymilldb для указанной версии Python, а также произведется установка зависимых пакетов.

Для легкого использования был реализован модуль __main__.py, это позволило максимально упросить процесс использования. Для компиляции MillDB, описанной с помощью SQL, записанном в файле по пути <path>, необходимо выполнить следующую команду:

```
python3 -m pymilldb <path>
```

После выполнения данной команды, в текущей директории создадутся два файла <file_name>.c и <file_name>.h, где <file_name> – имя файла указанного в <path>. Эти файлы и будут файлами библиотеки базы данных MillDB.

5. Тестирование

Важной частью данной работы было тестирование компилятора. Полученный в результате код некорректно тестировать на базовые тесты, так как ошибки могли бы себя не проявить. Поэтому было решено сравнивать на отличия файлы, полученные в процессе генерации на Python и C++.

Утилита diff поставляемая вместе с системой учета версий git может сравнивать два файла на отличия. Также данная утилита имеет возможность игнорировать отличия в пробельных символах и пустые строки. Это позволило проверять на эквивалентность файлы, которые имеют небольшие отличия в расстановке пустых строк и различные начальные символы, так в файле, сгенерированным при помощи Flex, Bison, используется символ табуляции, а в файле, сгенерированным при помощи Python, используются пробелы.

Для тестирования использовался SQL, представленный на листинге 4. Сравнение заголовочных файлов не выявило отличий, а единственным отличием в Си-файле стали комментарии.

Заключение

В ходе курсовой работы были изучены возможности генератора MillDB, была произведена интеграция новых функциональностей, а также произведена замена языка компиляции на Python, были приобретены навыки написание однопроходных компиляторов и рекурсивного синтаксического анализатора. В новом компиляторе были реализованы более точные отчеты об ошибках, а также в некоторых случаях было достигнуто самовосстановление в случае ошибок.

Дальнейшее развитие проекта может быть направлено на исправление существующих ошибок в самой базе данных, а также на добавление новой функциональности.

Список литературы

1) Репозиторий MillDB: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/mill-db (Дата обращения: 26.02.2020).

2) Репозиторий Flex: [Электронный ресурс]. URL: https://github.com/westes/flex (Дата обращения: 26.02.2020).

3) GNU Bison: [Электронный ресурс]. URL: https://www.gnu.org/software/bison/ (Дата обращения: 26.02.2020).

4) Расширение языка запросов в СУБД MillDB: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/mill-db/blob/nexterot-2019/docs/note.pdf (Дата обращения: 26.02.2020).

5) Реализация дополнительных индексов и фильтра Блума в Mill-DB: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/mill-db/blob/bartSens-2019/docs/Zapiska_Melnikov_DB-2019.pdf (Дата обращения: 26.02.2020).

6) Jinja: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/mill-db/blob/bartSens-2019/docs/Zapiska_Melnikov_DB-2019.pdf (Дата обращения: 26.02.2020).