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 ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ ПО КУРСУ
Конструирование компиляторов

НА ТЕМУ:

Язык логического программирования
для поиска информации в базе данных

Студент ИУ9-72Б _____ В. Е. Пичугин
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ А. В. Синявин
(Подпись, дата) (И.О.Фамилия)

Москва 2021 г.

Содержание

Введение

Логическое программирование – это парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций [1].

Декларативный подход к написанию программ, сочетающий реляционный взгляд на программирование с мощным символьным сопоставлением с образцом, делает языки логического программирования отличным выбором для построения интерфейсов к базам данных, служащих для поиска информации.

В рамках выполнения данной курсовой работы требуется реализовать интерпретатор для языка логического программирования, являющегося языком запросов к простой базе данных.

1. Обзор предметной области

1.1. Структура базы данных

База данных, из которой проектируемый язык запросов будет извлекать информацию, обладает простейшей структурой. Она состоит из утверждений (англ. assertions), которые представляют собой списки, элементами которых могут быть строки, целые числа, а также другие списки со сходной структурой. На Листинге 1 представлены примеры таких утверждений.

Листинг 1. Примеры утверждений из базы данных.

```
(position (Pichugin Vladislav) (junior developer))
(salary John (90 EUR))
((birth info) (Pichugin Vladislav) (date (19 April)))
```

1.2. Простые запросы

Всякий язык программирования обладает тремя механизмами для комбинации простых понятий с целью получения из них сложных [2]:

- элементарные выражения – представляют минимальные сущности, с которыми язык имеет дело;
- средства комбинирования – с помощью них из простых объектов составляются сложные;
- средства абстракции – с их помощью сложные объекты можно называть и обращаться с ними как с единым целым.

Начнем рассмотрение данных механизмов применительно к проектируемому языку с описания простых запросов, которые являются элементарными выражениями нашего языка.

Простой запрос по структуре похож на утверждение. Он указывает системе на то, что мы ищем в базе данных утверждения, соответствующие некоторому образцу. Особенностью синтаксиса простого запроса, отличающей его от синтаксиса утверждения, является возможность указания переменных образца (см. Листинг 2).

Листинг 2. Пример простого запроса.

```
(position $x (junior developer))
```

Пример выше ищет в базе данных всех младших разработчиков. Таким образом, переменная образца отличается от обычного строкового элемента утверждения лишь наличием знака доллара в начале. Переменных в образце может быть несколько, причем одна и та же переменная может встречаться в образце в нескольких местах (существенно наличие имени у переменной). В образце может и не быть переменных. В этом случае просто проверяется, содержится ли утверждение в базе. Если да, будет одна подходящая под образец запись. Если нет – ни одной.

Имеется также синтаксис для списковой переменной (см. Листинг 3).

Листинг 3. Пример запроса со списковой переменной.

```
(position $x (developer . $type))
```

Данный синтаксис означает, что в качестве $type может быть список с любым количеством значений. Например, подойдут такие записи:

```
(position (Pichugin Vladislav) (developer frontend backend))

(position (Ivan Ivanov) (developer android))

(position (Nikita Nikitin) (developer))
```

Обработку простых запросов в проектируемом языке можно описать следующим образом:

- система находит все присваивания переменным в образце запроса, которые удовлетворяют запросу;
- система отвечает на запрос, перечисляя все конкретизации образца с удовлетворяющими ему присваиваниями переменным (конкретизирует – значит замещает переменные в образце их реальными значениями, полученными в ходе выполнения запроса);
- если образец не содержит переменных, то запрос сводится к выяснению, находится ли образец в базе данных.

1.3. Составные запросы

Проектируемый язык предоставляет средства комбинирования для формирования составных операций: `@and`, `@or` и `@not`.

В общем случае, запросу `(@and <query1> ... <queryN>)` удовлетворяют все наборы значений переменных образца, которые одновременно удовлетворяют `<query1> ... <queryN>`. Как и в случае простого запроса, система при обработке составного запроса находит все присваивания переменным образца, удовлетворяющие запросу, и затем печатает все конкретизации запроса этими значениями.

Аналогично действуют оставшиеся средства комбинирования. Запросу `(@or <query1> ... <queryN>)` удовлетворяют все наборы значений переменных образца, которые удовлетворяют по крайней мере одному из `<query1> ... <queryN>`. А запросу `(@not <query1>)` удовлетворяют все присваивания переменным образца, которые не удовлетворяют `<query1>`.

Последней комбинирующей формой является `@apply`. Когда она стоит в начале образца, то указывает, что следующий элемент является предикатом, который требуется применить к остальным (конкретизированным) элементам как к аргументам. В общем случае, образец

```
(@apply <predicate> <arg1> ... <argN>)
```

удовлетворяется теми присваиваниями переменным образца, для которых применение `<predicate>` к конкретизированным `<arg1> ... <argN>` дает истину. Пример составного запроса, использующего данную комбинирующую форму, представлен на Листинге 4.

Листинг 4. Пример составного запроса.

```
(@and (salary $person $amount) (@apply > $amount 30000))
```

1.4. Правила

Правила в проектируемом языке являются средством абстракции запросов. Общий вид правила представлен на Листинге 5.

Листинг 5. Общий вид правила.

```
(@rule <conclusion> <body>)
```

где `<conclusion>` – это образец, а `<body>` – произвольный запрос. У правила может и не быть тела, в этом случае заключение удовлетворяется любыми значениями переменных. Правило представляет собой все конкретизации заключения при помощи присваиваний переменным, удовлетворяющих телу правила. Образец не обязательно должен явно присутствовать в базе как утверждение, это может быть неявное утверждение, следующее из правила. Пример правил представлен на Листинге 6.

Листинг 6. Примеры правил.

```
(@rule (liveNear $person1 $person2)
      (@and (address $person1 ($town . $rest1))
            (address $person2 ($town . $rest2))
            (@not (same $person1 $person2))))

(@rule (same $x $x))
```

Правила можно использовать в составных запросах, внутри других правил, можно даже задавать рекурсивные правила.

Можно рассматривать правило как логическую импликацию: если присваивание значений переменным образца удовлетворяет телу, то оно удовлетворяет заключению. Следовательно, можно считать, что язык запросов способен производить логический вывод на основании правил.

2. Проектирование системы обработки запросов

Запросная система организована вокруг двух основных операций, которые называются сопоставление с образцом и унификация. Простые и составные запросы реализуются через сопоставление с образцом и организацию информации в виде списка кадров (англ. frame). Для реализации правил требуется унификация – обобщение сопоставления с образцом.

2.1. Сопоставление с образцом

Сопоставитель – это программа, которая проверяет, соответствует ли некоторая структура данных указанному образцу. Сопоставитель принимает на входе образец, структуру данных и кадр, в котором указываются связывания для различных переменных образца (т.е. кадр является ассоциативным массивом, состоящим из связываний переменных – пар, отображающих имя переменной на некоторое значение, полученное в результате предыдущих сопоставлений). Он проверяет, соответствует ли структура данных образцу без противоречия со связываниями переменных, уже находящимися в кадре. Если да, то сопоставитель возвращает кадр, дополнив его связываниями, определенными во время сопоставления. Если нет, он указывает, что сопоставление неудачно.

Например, в результате сопоставления образца `(a $x b)` со списком `(a (a b) b)` при пустом кадре сопоставитель вернет кадр, в котором переменная `$x` будет связана со списком `(a b)`. Однако при сопоставлении тех же элементов, но с кадром, содержащим связывание `$x` с `(a b c)`, сопоставитель вернет неудачу, т.к. обнаружит противоречие с уже имеющимся в кадре связыванием.

2.2. Списки кадров

Получив образец и кадр, процесс сопоставления просматривает элементы базы данных один за другим. Для каждого утверждения он порождает либо неудачу, либо расширение исходного кадра по принципу, описанному в

предыдущем разделе (стоит отметить, что входной кадр никак не изменяется, расширение подразумевает порождение нового кадра, схожего с исходным, но обогащенного дополнительными связываниями). Из результатов сопоставления всей базы собирается список, который затем фильтруется. Получается список всех кадров, которые расширяют исходный кадр за счет сопоставления с какой-либо сущностью из базы.

Рассмотрим следующий пример (см. Рисунок 1): на вход сопоставителю подается образец `($x b $y)` и кадр `{$x: a}` (здесь и далее кадры будут записываться в формате JSON [3], так как данный формат хорошо отражает суть кадра, описанную ранее). Сопоставляя утверждения из базы данных с образцом, программа обнаруживает, что без противоречия с входным кадром из всех кандидатов образцу удовлетворяют лишь утверждения `(a b c)` и `(a b b)`. Соответственно, на выходе у сопоставителя будет список из двух кадров, каждый из которых расширяет исходный кадр: `{$x: a, $y: c}` и `{$x: a, $y: b}`.

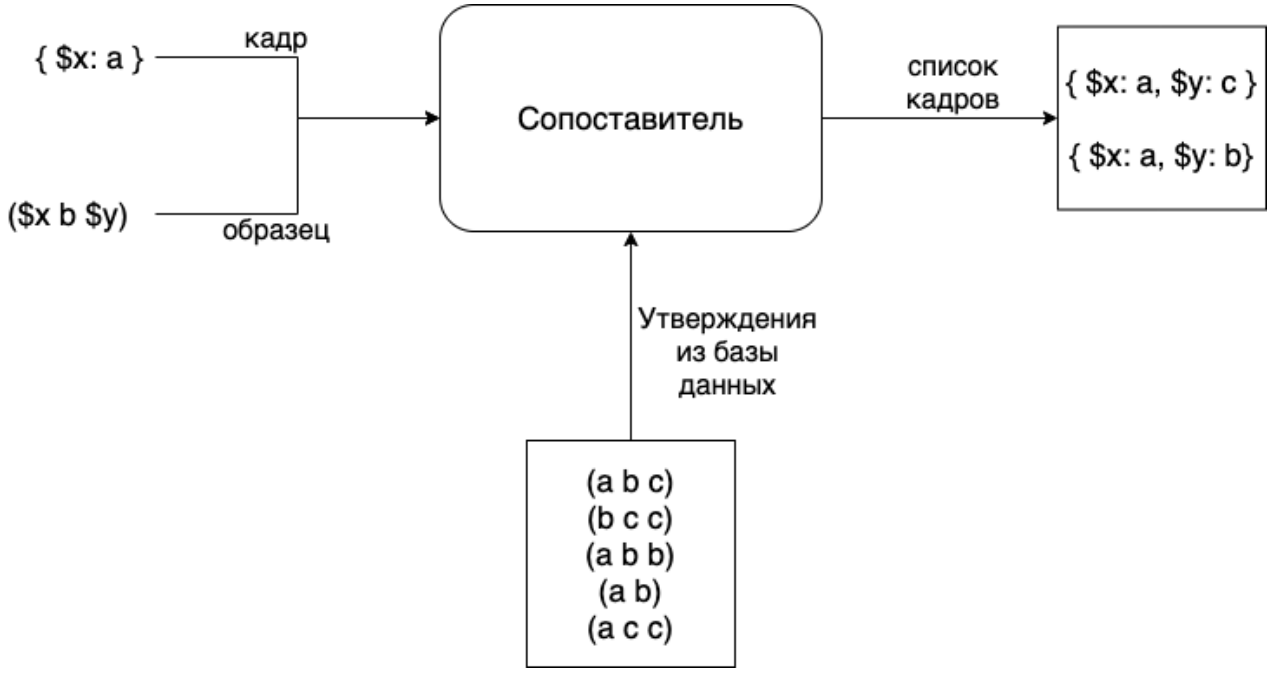

Рисунок 1. Пример работы сопоставителя.

В общем случае, обработчик запроса принимает входной список кадров (так сделано из-за рекурсивной природы интерпретатора, которая будет подробнее рассмотрена при описании реализации системы обработки запросов) и для каждого кадра применяет операцию сопоставления, т.е для каждого кадра

во входном списке генерирует новый отфильтрованный список. Затем эти списки сливаются в один, который и будет результатом запроса.

Например, чтобы ответить на простой запрос, применяем его к списку, состоящему из одного пустого кадра. Список на выходе содержит все расширения пустого кадра. Затем на основе этого списка кадров создается список копий исходного образца с конкретизированными переменными, и этот список печатается.

2.3. Составные запросы

Комбинация двух запросов через `@and` осуществляется последовательной обработкой списка кадров. Кадры, прошедшие через первый запрос, фильтруются и расширяются вторым запросом (см. Рисунок 2).

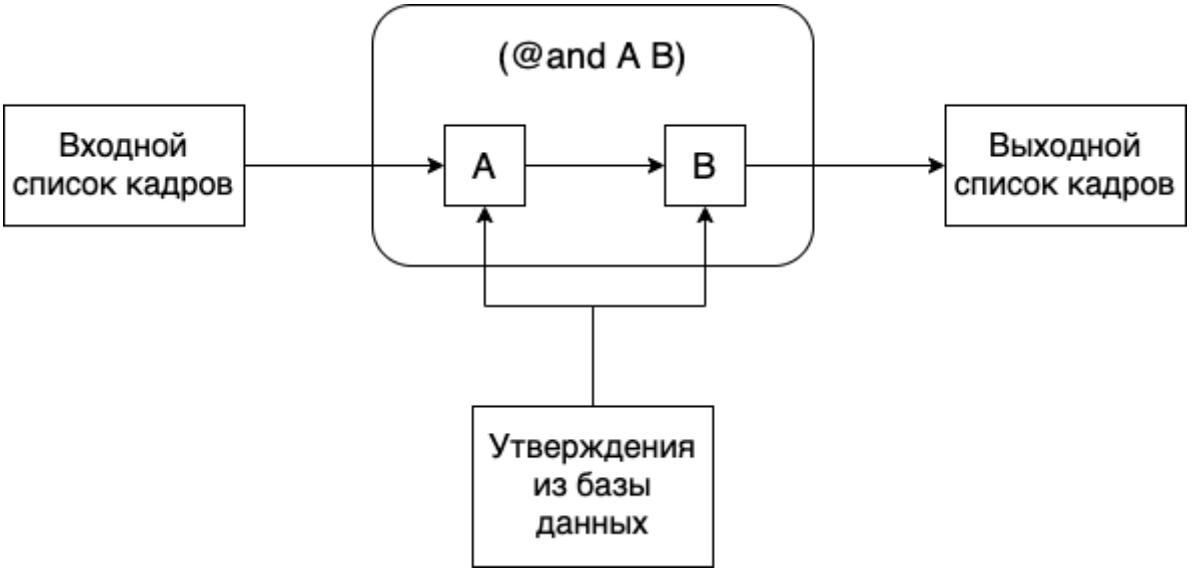

Рисунок 2. Схема работы комбинатора `@and`.

Рассмотрим следующий пример: нужно обработать запрос `(@and ($x c c) ($x b b))`, входной список кадров на начальном этапе состоит лишь из одного пустого кадра, утверждения в базе данных такие же, как и в примере из предыдущего раздела. В результате интерпретации первого конъюнкта получится список из двух расширений пустого кадра (`{$x: b}`, `{$x: a}`). Далее интерпретируется второй конъюнкт, которому на вход подается этот двухэлементный список. В результате выходной список кадров будет состоять

лишь из одного кадра `{$x: a}`, так как сопоставление образца `($x b b)` при кадре `{$x: b}` не удовлетворяется ни одним из имеющихся в базе утверждений.

Комбинация двух запросов через `@or` осуществляется путем параллельной обработки списка кадров и слияния результатов (см. Рисунок 3).

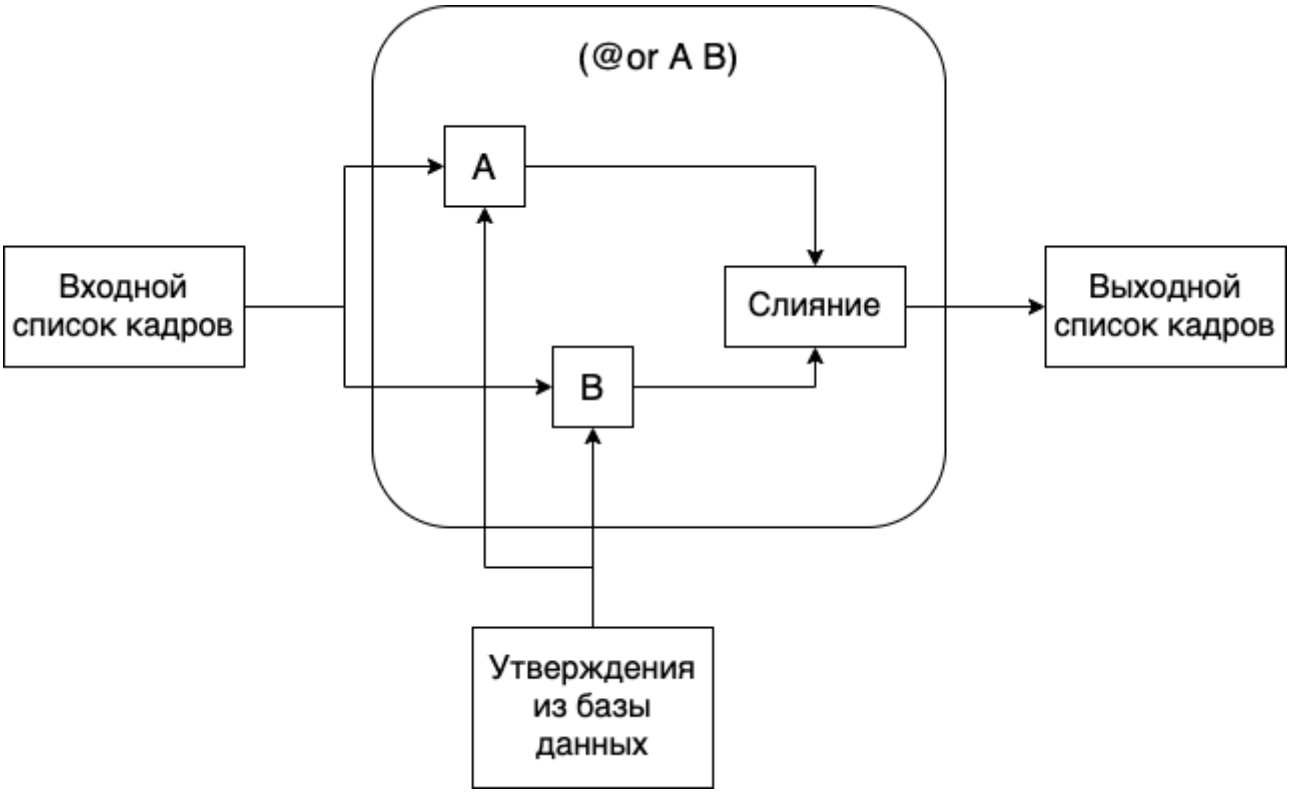

Рисунок 3. Схема работы комбинатора `@or`.

В качестве примера рассмотрим запрос, аналогичный запросу из предыдущего примера: `(@or ($x c c) ($x b b))`. В результате выполнения первого дизъюнкта со списком из одного пустого кадра получится список из двух расширений пустого кадра (`{$x: b}`, `{$x: a}`). Второй дизъюнкт согласно описанной схеме также выполняется со списком из одного пустого кадра, на выходе порождая одноэлементный список, содержащий кадр `{$x: a}`. Далее эти списки сливаются в один, обеспечивая ожидаемый результат.

`@not` работает как фильтр, уничтожающий все кадры, для которых его подзапрос можно удовлетворить. Особая форма `@apply` реализуется при помощи подобного же фильтра для списка кадров. При помощи каждого кадра из списка мы конкретизируем все переменные образца, а затем применяем

предикат. Все кадры, для которых предикат оказывается ложным, мы удаляем из входного списка.

2.4. Унификация

Чтобы обрабатывать правила языка запросов, нужно уметь находить правила, в которых заключения соответствуют данному входному образцу. Заключения правил подобны сущностям базы данных, но только в них могут содержаться переменные, так что потребуется обобщенный вариант сопоставления с образцом – унификация, в котором как образец, так и данные могут содержать переменные.

Унификатор берет два образца, в каждом из которых могут быть константы и переменные, и определяет, возможно ли присвоить переменным значения, которые сделают два образца одинаковыми. Если да, то он возвращает кадр, содержащий эти значения. Т.е. унификатор принимает на вход кадр и проводит унификации, не противоречащие содержимому этого кадра. Можно считать, что процесс унификации решает систему уравнений, описывающую компоненты образцов.

Например, рассмотрим унификацию `($x $x)` и `((a $y c) (a b $z))`. Ее можно представить как систему уравнений

```
$x = (a $y c)
$x = (a b $z)
```

из этих уравнений следует

```
(a $y c) = (a b $z)
$y = b, $z = c, $x = (a b c)
```

В общем случае успешная унификация может не полностью определить значения переменных: какие-то переменные могут остаться неопределенными, а значения других сами могут содержать переменные.

Например, рассмотрим унификацию `($x a)` с `((b $y) $z)`. В результате ее выполнения (с пустым начальным кадром) исходный кадр расширится связываниями `$x` с `(b $y)` и `$z` с `a`. Таким образом, унификация пройдет

успешно, хотя никакой более конкретной информации по `$x` и `$y` получено не было.

2.5. Применение правил

Главной операцией в компоненте запросной системы, который производит логический вывод на основе правил, является унификация. Типичный сценарий обработки запроса, содержащего обращение к правилу такой: с помощью обычного сопоставления проверяется, имеются ли в базе сущности, которые сопоставляются с данным образцом, на следующем шаге происходит попытка унифицировать образец-запрос с заключением каждого правила, так находится искомое правило. Получается кадр, расширенный новыми связываниями (в результате унификации). Теперь по отношению к этому кадру вычисляется составной запрос, содержащийся в теле правила. Успешные сопоставления расширяют кадр, который теперь можно использовать при конкретизации исходного образца-запроса.

Рассмотрим следующий пример. Пусть в базе данных лежат такие утверждения о зарплатах:

```
(salary Vlad 90)
(salary John 330)
(salary Sergey 12)
(salary Viktor 66)
(salary Ekaterina 5)
```

Тогда введем правило, по которому определяется, что у одного человека зарплата больше, чем у другого:

```
(@rule
    ($person1 salary is higher than $person2)
    (@and
        (salary $person1 $amount1)
        (salary $person2 $amount2)
        (@apply > $amount1 $amount2)
    )
)
```

И выполним запрос `($person salary is higher than Sergey)`. На первом этапе пройдут сопоставления с утверждениями из базы данных, среди которых не будет ни одного успешного. Далее произойдет успешная унификация с заключением единственного правила, в результате которого список кадров будет состоять из одного кадра `{$person: $person1, $person2: Sergey}`. Далее этот список кадров будет подан на вход обработчику составного запроса, находящегося в теле правила. В ходе выполнения первого конъюнкта образуется список из кадров, в которых будут находится связывания для всех утверждений из базы. На втором шаге каждый из этих кадров обогатится информацией о зарплате Сергея. Наконец, на последнем шаге будут отфильтрованы кадры, в которых связывания не удовлетворяют искомому предикату сравнения. После конкретизации исходного запроса с помощью кадров выходного списка, получим следующий результат:

```
(Vlad salary is higher than Sergey)
(John salary is higher than Sergey)
(Viktor salary is higher than Sergey)
```

2.6. Простые запросы

Теперь можно описать, как вычисляются простые запросы в общем случае.

Получая запрос-образец и список кадров, мы порождаем для каждого входного кадра два новых списка:

- список расширенных кадров, полученных сопоставлением образца со всеми утверждениями из базы данных (с помощью сопоставителя);
- список расширенных кадров, полученных применением всех возможных правил (с помощью унификатора).

  Далее все эти списки сливаются в один большой.

2.7. Управляющий цикл

В программе должна быть некоторая функция, координирующая операции сопоставления. Она принимает на вход запрос и список кадров. Ее

выходом служит список кадров, соответствующих успешным сопоставлениям с запросом, которые расширяют какой-либо кадр во входном списке. Данная функция распознает различные типы запросов и для каждого из них вызывает соответствующую функцию (`@and`, `@or`, `@not`, `@apply`, и еще одну для простых запросов).

Управляющий цикл считывает запросы с терминала. Для каждого запроса он вызывает координирующую функцию с запросом и списком из одного пустого кадра. Для каждого кадра в выходном списке управляющий цикл конкретизирует входной запрос, затем список конкретизированных запросов печатается на экран.

Кроме того, управляющий цикл распознает особую команду `@new`, которая говорит, что на вход поступает не запрос, а новое утверждение или правило, которое следует положить в базу данных. Пример использования данной команды представлен на Листинге 7.

Листинг 7. Пример использования команды `@new`.

```
(@new
   (@rule (bigBoss $person)
       (@and
           (boss $middleManager $person)
           (boss $x $middleManager)
       )
   )
   (position Denis developer)
)
```

2.8. Формализация языка запросов

Наконец, стоит более подробно рассмотреть лексическую структуру и грамматику проектируемого языка.

В языке запросов выделяются следующие лексические единицы:

- Числа (Number): `[0-9]+`
- Слова (Word): `[a-zA-Z]+[0-9]*`
- Переменные (Var): `\$[a-zA-Z]+[0-9]*`

- Особые имена: `\(|\)|@new|@rule|@apply|@and|@or|@not|<|>|\.`

В соответствии со сформулированными требованиями к возможностям языка была составлена следующая грамматика (см. Листинг 8), записанная в РБНФ [4].

Листинг 8. Грамматика языка запросов.

```
Command ::= '(' Insert | Query ')'
Insert ::= '@new' Entity+
Entity ::= '(' Assertion | Rule ')'
Assertion ::= ('(' Assertion ')' | Word | Number)*
Rule ::= '@rule' '(' SimpleQuery ')' ('(' Query ')')?
Query ::= SimpleQuery | AndQuery | OrQuery | NotQuery
AndQuery ::= '@and' InnerQueries
OrQuery ::= '@or' InnerQueries
NotQuery ::= '@not' InnerQuery
InnerQuery ::= '(' Query | Apply ')'
InnerQueries ::= InnerQuery+
Apply ::= '@apply' Predicate ApplyArguments
Predicate ::= '<' | '>' | Word
ApplyArguments ::= (Var | Word | Number)+
SimpleQuery ::= ('(' SimpleQuery ')' | Var | Word | Number)* ('.' Var)?
```

3. Реализация системы обработки запросов

В качестве языка программирования для реализации проекта был выбран Python [5]. Соответственно, логика проекта разбита на пакеты, которые, в свою очередь, разбиты на модули. Общая файловая структура представлена на Рисунке 4.

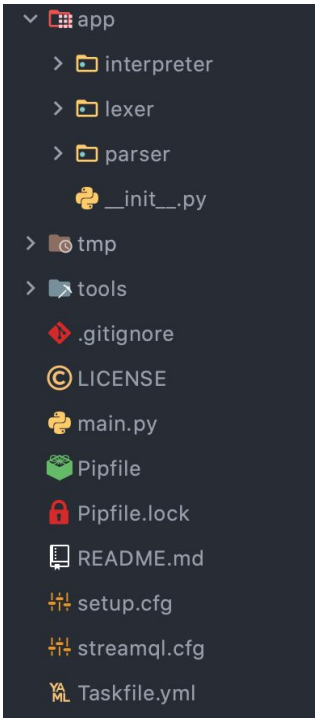

Рисунок 4. Файловая структура проекта.

3.1. Реализация лексического анализатора

Одним из трех основных пакетов является пакет lexer, в котором сосредоточена логика лексического анализа входной последовательности кодовых точек в соответствии с ранее описанной лексической структурой языка запросов.

Лексер работает на основе поиска в тексте по образцу, заданному регулярным выражением. Идея лексического анализа состоит в использовании групп, представляющих собой фрагменты регулярных выражений, заключенные в круглые скобки, значения которых запоминаются при сопоставлении текста с образцом.

Класс, инкапсулирующий логику лексического анализа, является итератором с методом `next_token()`. Данный метод при каждом вызове возвращает следующий токен из входной последовательности. Токен является объектом, хранящим информацию о домене, значении и координатах лексемы.

При обнаружении ошибки во входной последовательности лексический анализатор выдает сообщение с указанием координаты. Также после обнаружения ошибки лексер восстанавливается по следующей схеме: из входной последовательности пропускаются все подряд идущие символы до нахождения следующей лексемы.

3.2. Реализация синтаксического анализатора

Вторым основным пакетом в проекте является пакет parser, содержащий логику синтаксического анализа последовательности токенов, порожденной лексическим анализатором.

Нетрудно увидеть, что описанная ранее грамматика языка запросов является LL(1)-грамматикой [6], записанной в РБНФ. Поэтому в качестве алгоритма синтаксического разбора было удобно взять метод рекурсивного спуска [7], при котором каждому нетерминалу грамматики соответствует отдельная функция, в которой закодирован эффект применения соответствующего нетерминалу правила.

Целью парсера является проверка текста входной программы на соответствие грамматике, а также порождение синтаксической структуры, удобной для дальнейшего использования интерпретатором. В качестве такой структуры выступает список, элементами которого могут быть атомарные единицы интерпретации, представляющие собой совокупность домена и значения, а также другие списки с идентичной структурой.

Также стоит отметить, что при обнаружении синтаксической ошибки парсер порождает специальное исключение, содержащее сообщение об этой ошибке с указанием координат во входной последовательности.

3.3. Реализация интерпретатора языка запросов

Третьим основным пакетом в проекте является пакет interpreter, содержащий логику интерпретации синтаксической структуры, полученной от парсера.

Основной сущностью в этом пакете является класс Interpreter, экземпляр которого хранит все сохраненные пользователем утверждения и правила (таким образом, все сущности базы данных хранятся в оперативной памяти), словарь с доступными предикатами для особой формы `@apply` (на данный момент определено лишь два предиката: «больше» и «меньше», но их количество легко увеличить, достаточно лишь добавить новый обработчик в словарь) и потребителя конкретизированных образцов, полученных в результате выполнения запроса. Конкретная версия потребителя передается при создании экземпляра класса, это помогает сделать код более тестируемым.

Данный класс имеет лишь один публичный метод, который в качестве единственного аргумента принимает строковое представление команды, которую необходимо выполнить. Внутри себя с помощью ранее описанного парсера он получает синтаксическую структуру, которую и начинает интерпретировать.

В случае, если полученная команда – это запрос на создание нового утверждения или правила, то система действует следующим образом. Так как сопоставление с образцом – весьма дорогая операция, то хочется строить базу данных так, чтобы для некоторых запросов многие кандидаты на сопоставление отсеивались на самом начальном этапе во избежание сопоставления со всеми сущностями базы. Таким образом, хочется проиндексировать базу данных. Поэтому помимо хранения одного большого списка со всеми утверждениями, в отдельных списках хранятся утверждения, первые элементы которых являются словами (Word) или числами (Number). Чтобы получить утверждения, которые могут сопоставляться с образцом, сначала проверяется, не является ли первый элемент образца словом или числом. Если это так, то сопоставителю возвращаются только элементы из соответствующего списка. Иначе

возвращаются все хранимые утверждения. Правила хранятся аналогичным образом, с использованием первого элемента заключения. Единственное отличие заключается в том, что заключение в качестве первого элемента может содержать переменную. Образец, первым элементом которого является слово или число, может сопоставляться не только с правилами, у которых первый элемент заключения содержит ту же константу, но и с правилами, где в начале заключения стоит переменная. Таким образом, при поиске правил, которые могут сопоставляться с образцом, у которого в начале константное значение, сопоставителю возвращаются как правила с этим значением в первом элементе заключения, так и правила с переменной в начале заключения. Ради этого правила с переменными в начале заключения хранятся в отдельном списке. Пример такой индексации можно увидеть на Рисунке 5.

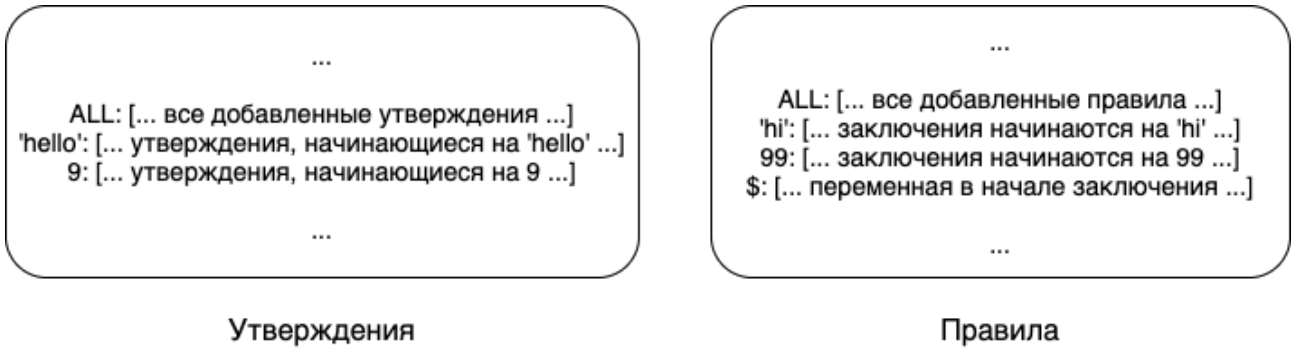

Рисунок 5. Пример индексации базы данных.

Если же полученная команда оказывается запросом на извлечение данных из базы, то управление передается координирующей функции, описанной в разделе 2.7. Обработчики запросов (простых, составных, содержащих обращение к правилу) построены на рекурсии, поэтому существенно, что в качестве параметра они принимают именно список кадров, а не просто отдельный кадр.

Обработчики составных запросов реализованы ровно так, как и описывалось при проектировании. Реализация обработчиков `@and` и `@or` представлена на Листинге 9.

Листинг 9. Реализация обработчиков `@and` и `@or`.

```
def _and(self, conjuncts: AST, frames: List[Frame]) -> List[Frame]:
    if len(conjuncts) == 0:
```

```
            return frames
        return self._and(
            conjuncts[1:],
            self._run_query(conjuncts[0], frames)
        )

    def _or(self, disjuncts: AST, frames: List[Frame]) -> List[Frame]:
        if len(disjuncts) == 0:
            return []
        return list(chain(*[
            self._run_query(disjuncts[0], copy.deepcopy(frames)),
            self._or(disjuncts[1:], frames)
        ]))
```

При обработке простого запроса происходит сопоставление образца с утверждениями базы данных. Основная идея сопоставления состоит в том, чтобы сравнивать образец с данными, элемент за элементом, и собирать при этом связывания переменных образца. Если образец и объект данных совпадают, то сопоставление оказывается успешным, и возвращается список собранных связываний. В противном случае, если образец является переменной, то производится попытка расширения имеющегося кадра, за счет связывания переменной с данными, если это не противоречит уже имеющимся в кадре связываниям. Если и образец, и данные являются списками, то происходит рекурсивное сопоставление (особым образом обрабатывается случай списковой переменной). Если ни один из этих случаев не применим, сопоставление терпит неудачу, и возвращается None. Исходный код функций, отвечающих за сопоставление с образцом представлен на Листинге 10.

Листинг 10. Исходный код функций сопоставления с образцом.

```
def _pattern_match(
        self,
        pattern: AstNode,
        data: AstNode,
        frame: Optional[Frame]
) -> Optional[Frame]:
    if frame is None:
        return None
    if pattern == data:
        return frame
```

```
    if is_var(pattern):
        return self._extend_frame(pattern.value, data, frame)
    if is_non_empty_list(pattern) and is_list(data) and is_dot(pattern[0]):
        return self._pattern_match(pattern[1], data[0:], frame)
    if is_non_empty_list(pattern) and is_non_empty_list(data):
        return self._pattern_match(
            pattern[1:],
            data[1:],
            self._pattern_match(pattern[0], data[0], frame)
        )
    return None

def _extend_frame(self, var: str, data: AST, frame: Frame) -> Optional[Frame]:
    binding = frame.get(var)
    if binding is None:
        frame[var] = data
        return frame
    return self._pattern_match(binding, data, frame)
```

Функция, отвечающая за применение правила, работает так, как и описывалось при проектировании. Однако перед унификацией она переименовывает все переменные в правиле и дает им уникальные новые имена. Это делается для того, чтобы переменные из различных применений правил не смешивались друг с другом. Алгоритм унификации в основном подобен алгоритму сопоставления, но только он симметричен – переменные разрешаются с обеих сторон сопоставления. Также при попытке связывания переменной может произойти следующая нехорошая ситуация: программа захочет связать эту переменную с образцом, который ее саму содержит. Реализованный алгоритм унификации учитывает данную ситуацию и не позволяет произойти подобному связыванию.

3.4. Реализация управляющего цикла

Точкой входа в программу является файл main.py, который следует запускать из терминала.

Сначала в нем происходит чтение данных из конфигурационного файла streamql.cfg, в котором должен находиться путь до файла с исходным кодом,

который будет выполнять интерпретатор при считывании команды run из стандартного потока ввода.

Далее создается объект класса Interpreter, которому в качестве потребителя передается анонимная функция, осуществляющая печать конкретизированных образцов в стандартный поток вывода. Таким образом, база данных инициализирована.

После этого определяются функции, осуществляющие печать в консоль справки о программе и запуск интерпретации команды, считанной из файла.

Наконец, определяется бесконечный цикл, в котором происходит консольное взаимодействие с пользователем, считывание команд и их исполнение с обработкой исключений. На данный момент в программе имеется три команды:

1. run – исполняет код из исходного файла, заданного в конфигурации;
2. help – отображает справку о программе на экране;
3. любой другой ввод интерпретируется как путь до файла с исходным кодом.

4. Тестирование системы обработки запросов

В процессе создания приложения проводились постоянные проверки корректности его работы.

Для контроля за написанием качественного кода в проект были добавлены различные инструменты статического анализа кода на языке программирования Python. Также с целью написания более надежного кода в проект был интегрирован инструмент, позволяющий осуществлять статическую проверку типов в языке Python.

Также на каждый из трех основных пакетов приложения были написаны юнит-тесты с помощью библиотеки pytest [8]. В общей сложности в проекте написано 44 юнит-теста, которые покрывают все описанные возможности языка запросов и довольно много случаев его использования. На Рисунке 6 представлен сгенерированный отчет о покрытии исходного кода тестами.

Рисунок 6. Отчет о покрытии кода тестами.

В результате тестирования установлено, что приложение удовлетворяет всем требованиям, сформулированным при проектировании.

Заключение

В рамках настоящей курсовой работы был реализован интерпретатор для языка логического программирования, являющегося языком запросов к примитивной базе данных.

В ходе работы было проведено знакомство с логическим программированием, алгоритмами сопоставления с образцом и унификации, языком программирования Python и его библиотеками.

Кроме того, на практических примерах были отработаны методы лексического анализа на основе регулярных выражений и рекурсивного спуска для синтаксического анализа, ранее рассмотренные в рамках курса «Конструирование компиляторов».

Стоит отметить, что данный проект можно сильно улучшить, среди такого рода планов можно выделить следующие: написание графического клиента для системы обработки запросов, сохранение сущностей базы данных на диск, улучшение индексирования, создание новых особых форм и дополнительных предикатов для `@apply` и т.д.

Список использованных источников

1. Ред. В. Н. Агафонов. Логическое программирование. Сборник статей. — М.: Мир, 1988. — С. 368. — 15 000 экз. — ISBN 5-03-000972-8.
2. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. М.: Добросвет, 2006
3. JSON. URL: http://www.json.org/ (дата обращения 18.01.2021)
4. Pattis, Richard E. "EBNF: A Notation to Describe Syntax". University of California, Irvine. URL: https://www.ics.uci.edu/~pattis/misc/ebnf2.pdf (дата обращения 18.01.2021)
5. Python. URL: https://www.python.org/ (дата обращения 18.01.2021)
6. Белоусов А.И., Ткачев С.Б. Дискретная математика: Учеб. для вузов / Под ред. В.С. Зарубина, А.П. Крищенко. -3-е изд., стереотип. - М.: Изд-во МГТУ им. Н.Э. Баумана, 2004. - 744 с. (Сер. Математика в техническом университете; Вып. XIX)
7. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : Пер. с англ. – М.: Изд. дом Вильямс, 2008. – 1184 с.
8. Pytest. URL: https://docs.pytest.org/en/stable/ (дата обращения 18.01.2021)