Федеральное государственное автономное образовательное учреждение высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»»**

**Факультет информатики, математики и компьютерных наук**

**КУРСОВАЯ РАБОТА**

Создание встраиваемых ML компонент для IoT

по направлению подготовки 09.03.04 «Программная инженерия»

образовательная программа «Программная инженерия»

Выполнил:

Студент группы 19ПИ-1

Смирнов Григорий Андреевич

Руководители:

Башев Владимир

Ильин Николай

Нижний Новгород 2020

**Содержание**

1.Введение

2.Постановка задачи

3.Исследование Предметной Области

3.1

3.2

3.3

3.4

3.5

4.Программная реализация

4.1 Описание переменных

4.2 Описание функций

5.1

5.2

5.3

6.Инструкция пользователя

7.Заключение

7.1 Результат работы программы

7.3 Ссылка на GitHub

8.Список литературы

1.**Введение**

Интернет вещей – концепция передачи данных между физическими объектами, наделенными специальным функционалом, для коммуникации друг с другом и с внешней средой. Считается, что сети такого вида могут значительно улучшить многие общественные и экономические процессы, исключив из части операций и действий необходимость участия человека.

Часто в интернете вещей нужно проводить анализ, поиск паттернов в потоках постоянно поступающих данных. Как пример можно рассмотреть датчики температуры и торговые автоматы. В первом случае в потоке постоянно поступающих результатов измерений, нам нужно вычленять те данные, которые свидетельствуют о превышении определенного порога температуры, для последующей передачи сообщений в системы исправления неисправностей или в системы пожаротушения. Во втором случае в постоянно поступающем от торгового автомата потоке данных необходимо вычленять как данные о том, что тот или иной товар закончился, так и данные о конкретных товарах для последующего преобразования их использования в моделях машинного обучения связанных с улучшением качества обслуживания.

Таким образом в контексте синтаксического анализа было принято решения сосредоточить мою работу на алгоритмах, связанных с регулярными выражениями и конечными автоматами.

Конечный автомат – это абстрактное дискретное устройство, имеющее конечный набор состояний и изменяющее состояния с зависимости от входных данных. Для реализации поиска шаблонов зачастую используются конечные автоматы.

Регулярное выражение – простой способ задать шаблон поиска.

2.**Постановка задачи**

В рамках Эко Платформы (Eco Framework), которая является развитием COM архитектуры и создателем и архитектором которой является Башев В.Н. (Технический директор компании «ПИРФ» и инженер-разработчик АО “НЭК Нева коммуникационные системы”), мне была поставлена задача реализации алгоритма Томпсона для построения конечного автомата по регулярному выражению как компонент для EcoOS. Полученный компонент может быть использован при разработке приложений на языке Си, поскольку удовлетворяет условиям стандарта 89, а также обладает особенностями, присущими всем компонентам Эко Платформы, а именно:

• позволяет повторно применять компоненты в различных решениях

• избавляет от переписывания кода под поколения операционных систем;

• дает возможность добавлять новый функционал, используя механизмы включения или агрегирования без переписывания кода;

Для достижения поставленной цели мною были определены следующие задачи:

1. Изучить теорию формальных языков, разобраться в ключевых понятиях
2. Дать формальные определения понятиям регулярного выражения и конечного автомата
3. Изучить алгоритм построения конечного автомата, конструкцию Томпсона
4. Реализовать алгоритм для построения конечного автомата по регулярному выражению как компонент для EcoOS.
5. Изучить алгоритмы минимизации, конвертации NFA в DFA
6. Изучить материалы, связанные с построением пайплайнов для ML, видами пайплайнов

3.**Исследование Предметной Области**

Алфавит, слово, язык

Итак, моя тема связана с *регулярными выражениями* и *конечными автоматами.* Чтобы в последствие говорить об алгоритмах детерминизации, минимизации и т.д. я предлагаю глубже погрузиться в тему и разобраться с некоторой базовой терминологией.

*Теория регулярных языков* и *теория конечных автоматов* являются основой просто огромной темы, называемой «Теория Формальных Языков».

Говоря «Формальный язык» мы имеем в виду язык искусственный, придуманный людьми для специальных целей. Как пример можно привести языки программирования. Главное отличие стихийно возникающих и развивающихся естественных языков от «формальных» в том, что естественные языки характеризуются довольно обширными и сложными грамматическими правилами, довольно жестко формализованы. Но какой-то непреодолимой преграды между ними нет.

Перейдем к терминологии.

Самым основополагающим понятием в теории языков является *алфавит*. *Алфавит –* это произвольное конечное непустое множество V = {a1, a2, a3, … an}, элементы которого называют символами или буквами.

*Словом* в алфавите V называют произвольный кортеж из «букв» этого алфавита. Например: a, b, c, ab, bc, abc, aabc, abcbca, cccc, bb являются словами в алфавите V={a, b, c}.

Множество всех слов в алфавите V принято обозначать как V\*. *Языком* в алфавите V называют произвольное подмножество множества V\*.

Семантика и синтаксис

При изучении языков нас будет интересовать два их главных аспекта –

*Семантика* предполагает сопоставление слов языка с некоторым «смыслом», «значением». То есть, аппелируя теми или иными словами естественного языка, мы знаем, что то или иное слово означает, и можем составлять их в последовательности, несущие под собой некоторую определенную информацию, которую нам необходимо передать. Аналогично и в программировании, используя ключевые слова языка (if, for, class, explicit, final…) мы описываем вполне определенные вещи, необходимые нам. Как пример нарушения семантики можно привести сюрреалистичные словосочетания по типу «Съел поезд», «Упавшее солнце». Здесь слово, обозначающее действие не согласуется с самим объектом, над которым это действие производится, отсюда появляется диссонанс, смысл конструкции теряется, хоть синтаксически она и правильна. По схожей причине семантически неправильны и последовательности вида «explicit for» или «void class».

*Синтаксис языка* – это такой набор правил, в соответствии с которыми можно строить «правильные» последовательности букв и слов (предложений, программ) относительно этого языка. Например: последовательность «Я люблю ты» является синтаксически неправильной для русского языка. Последовательность «x:=t» является неправильной для языка С но при этом абсолютно правильна для языка Pascal. Также «while i < 0:» - неправильна для C, но правильна для Python.

Тогда необходимо разработать, с одной стороны, механизмы перечисления (порождения), слов в соответствии с заданными правилами, а с другой, механизмы проверки того, что заданное слово принадлежит заданному языку. Прежде всего, именно эти механизмы и изучает теория формальных языков.

Регулярные выражения

Формальный язык может быть определен несколькими способами, например:

1. Простым перечислением слов, входящих в данный язык
2. Словами, порожденными некоторой *формальной грамматикой*
3. Словами, порожденными *регулярным выражением*
4. Словами, распознаваемыми некоторым *конечным автоматом*

Регулярное выражение над алфавитом {c1,c2,c3…cn} – способ порождение языка над этим алфавитом. Определяется рекурсивно в соответствие со следующими правилами:

1. Любое ci является регулярным выражением задающим язык из одного слова ci
2. Существует язык, состоящий из пустой строки. Существует пустой язык.
3. Если регулярное выражение a1 задает язык L1, a2 задает L2, то (а1)|(а2) задает язык L1vL2
4. Если регулярное выражение a1 задает язык L1, a2 задает L2, то (а1)(а2) задает L1L2
5. Если регулярное выражение a1 задает язык L1, то а1\* задает L1\*
6. Операции указаны в порядке приоритета, при этом скобки повышают приоритет операции, как в арифметических выражениях.

Применение регулярных выражений

Регулярные выражения применяются как формальный язык поиска и манипуляций с подстроками в тексте, основанный на использовании «метасимволов» (то есть знаков, имеющих специальное, заранее определенное значение при записи образцов). Для поиска используется строка-образец (шаблон), задающая правило поиска.

Примеры метасимволов:

В разных типах регулярных выражений этот набор может различаться

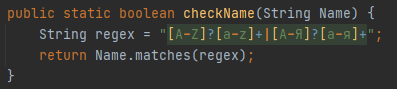


Регулярные выражения находят применение в задачах поиска совпадений подстроки в строке, согласно стандарту POSIX Basic Regular Expression (функции match, search).

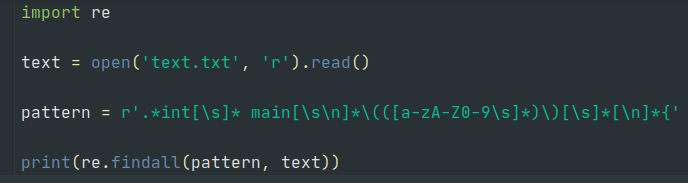
Большинство современных языков программирования имеют поддержку регулярных выражений, регулярные выражения используются в UNIX утилитах sed и grep, а так же в некоторых текстовых редакторах.

Примеры использования регулярных выражений в разных языках программирования:

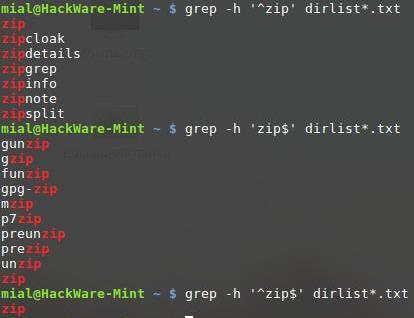
Java:



Python:



Unix Bash утилита *grep*:



Конечный автомат

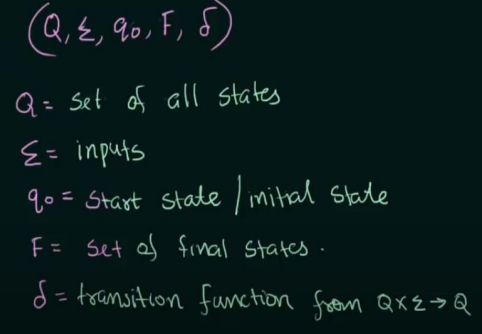
Регулярный язык в теории формальных языков – множество слов, которое распознает *конечный автомат.*

В теории алгоритмов *Конечный автомат –* математическая абстракция, модель дискретного устройства, принимающая на вход символы некоторого конечного алфавита (события) и в каждый момент времени, находящаяся в одном состоянии из множества возможных. Выделяют начальное состояние и одно или несколько конечных состояний. Если по окончании поступления на вход последовательности событий, автомат находиться в одном из конечных состояний, считается, что он «принял» последовательность.

Различают детерминированные и недетерминированные конечные автоматы (DFA и NFA).

Главные отличия NFA от DFA – из некоторого состояния могут наличествовать переходы по одному и тому же символу, могут наличествовать переходы по пустому входу, у состояния может не быть переходов по тем или иным символам.

Формальное определение конечного автомата



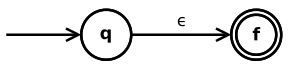
Формально конечный автомат определяется как пятерка:

1. Множество всех состояний автомата
2. Входной алфавит (все возможные входные символы)
3. Начальное состояние
4. Множество конечных состояний
5. Функция перехода между состояниями

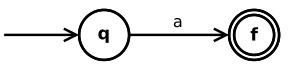
Конструкция Томпсона

Согласно правилам конструкции Томпсона, каждая операция регулярного выражения преобразуется в заданный набор связей в выходном конечном автомате.

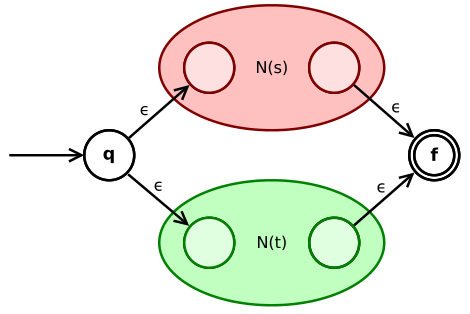
Пустое выражение преобразуется в:



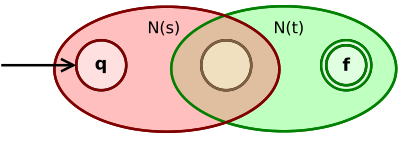
Символ входного алфавита преобразуется в:



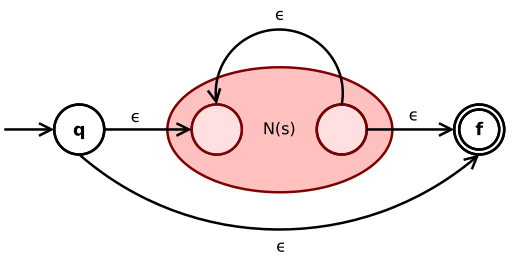
Объединение s|t преобразуется в:



Конкатенация st преобразуется в:



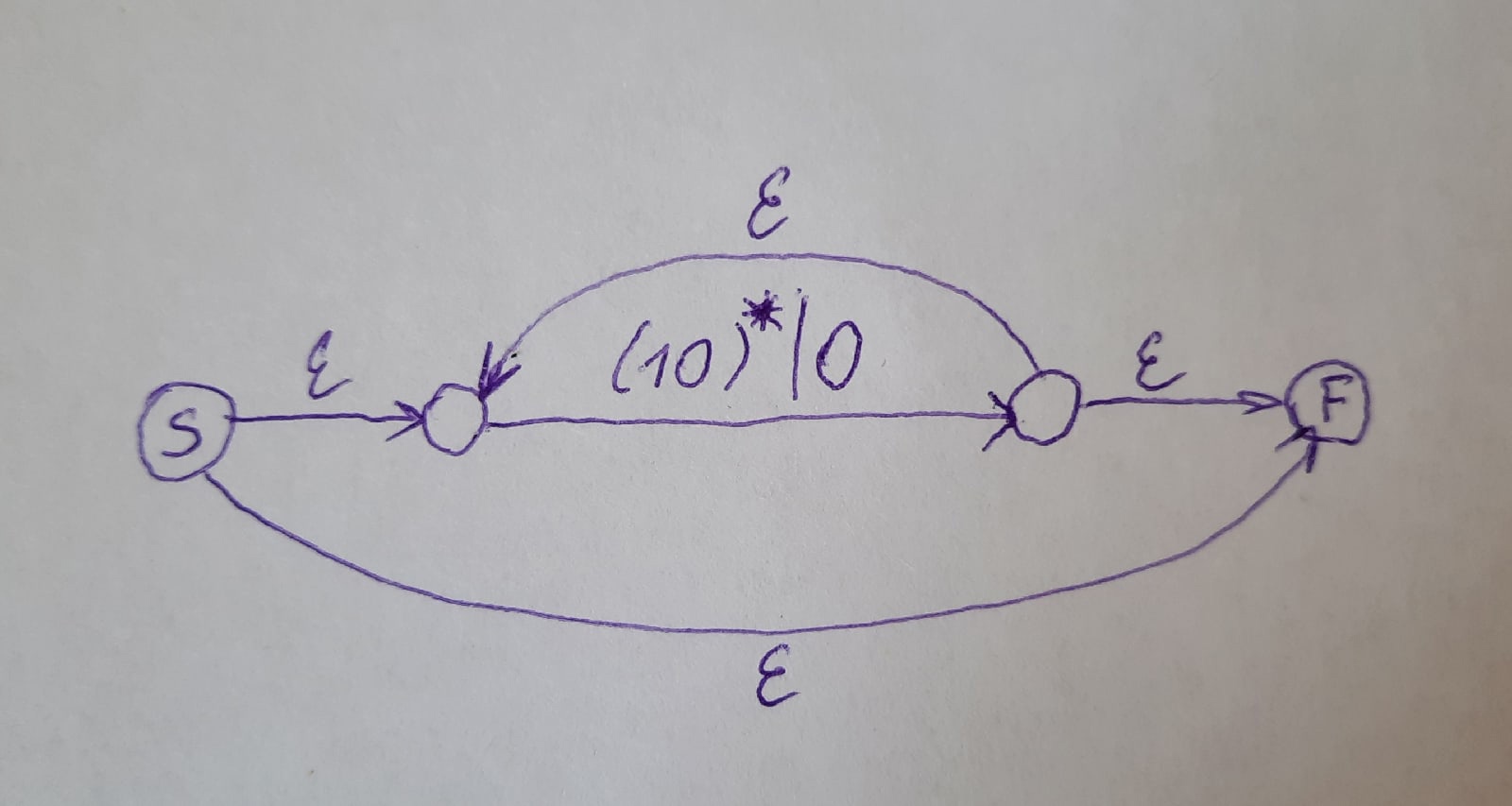
Выражение со звездой Клини s\* преобразуется в:



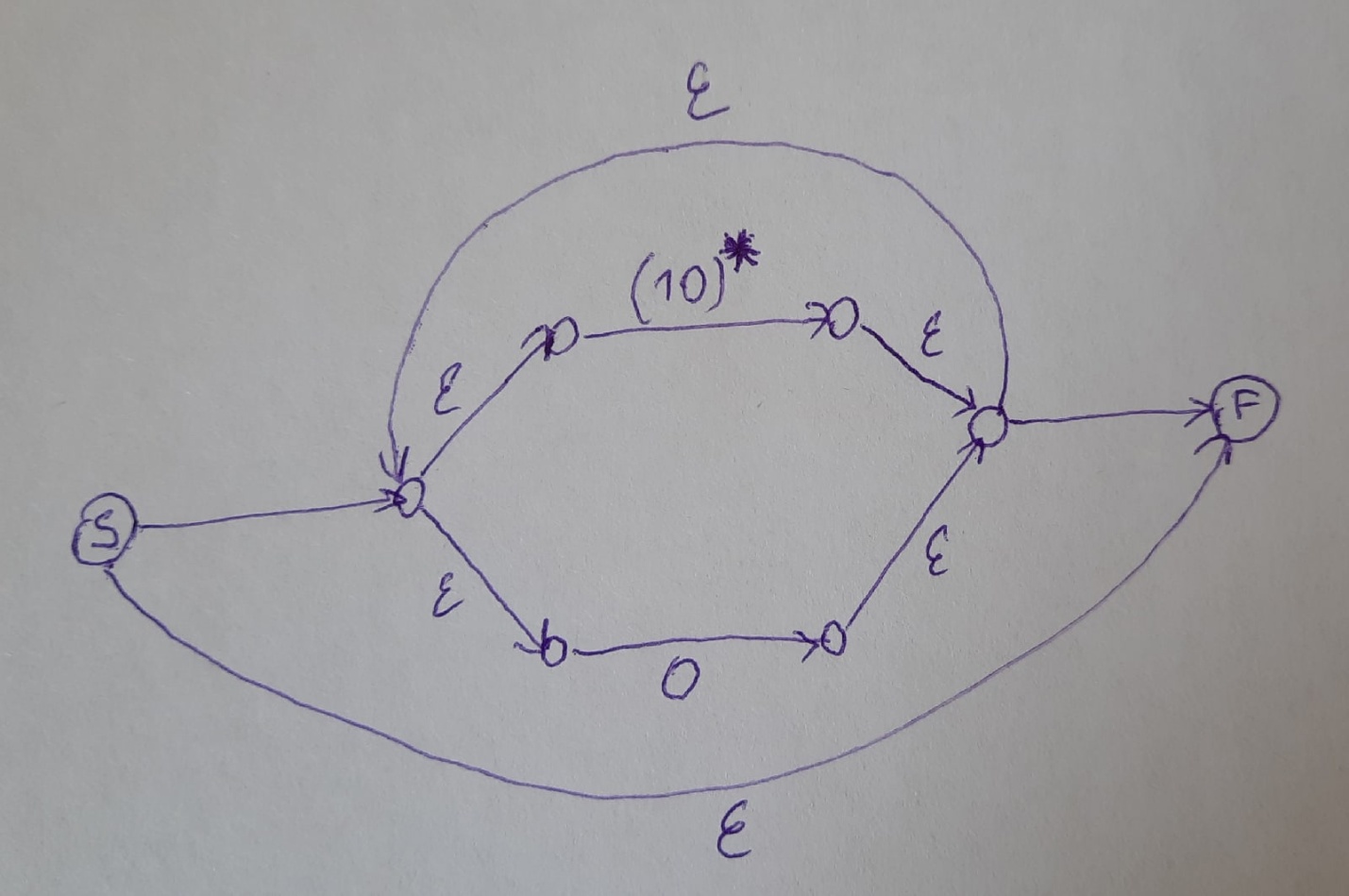
Пример построения NFA по конструкции Томпсона. Этапы

Построим NFA для следующего регулярного выражения **((10)\*|0)\*** где «0» и «1» – символы входного алфавита

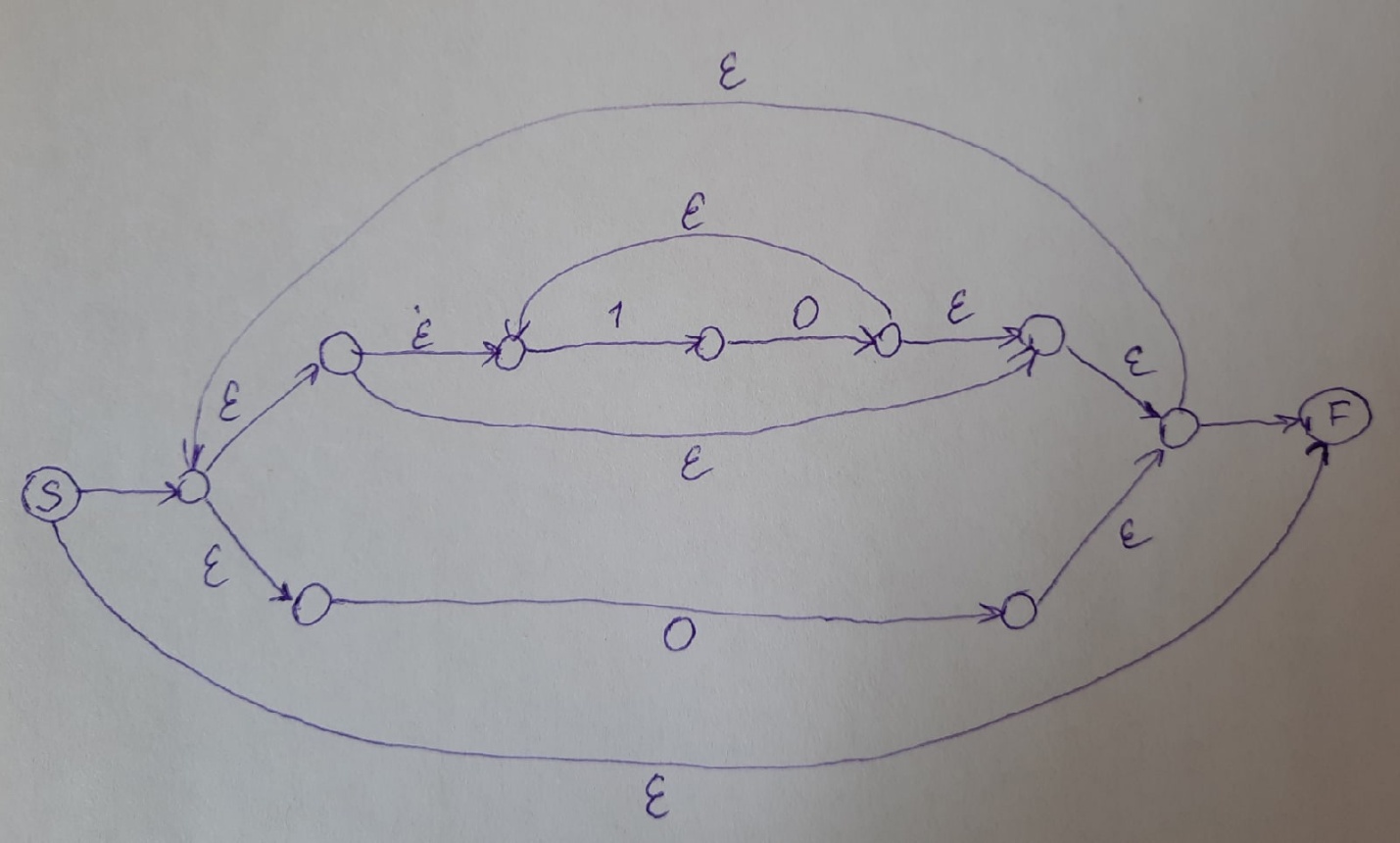
Этап 1, обработка звезды Клини



Этап 2, обработка дизьюнкции



Этапы 3,4 обработка звезды Клини и конкатенации



**Описание алгоритма**

Функция построения NFA по регулярному выражению рекурсивна. На вход принимает начальное состояние, конечное состояние, регулярное выражение и автомат, в который она будет добавлять новые состояния и переходы. Алгоритм работы словесно можно описать так:

Вначале нам нужно найти самую «верхнюю» операцию. Для этого в цикле идем по символам регулярного выражения, если встречаем «(» - закладываем в стек, если встречаем «)» - убираем скобку из стека. Первая операция при пустом стеке скобок и будет искомой.

Это может быть конъюнкция или дизъюнкция. Добавляем в автомат состояния и связи в соответствии с конструкцией Томпсона, разбиваем регулярное выражение относительно операции, делаем два рекурсивных вызова относительно новых состояний, в которые передаем соответствующие части регулярного выражения.

Если в проходе по регулярному выражению, мы не нашли конъюнкцию или дизъюнкцию при пустом стеке, значит – самая верхняя операция – звезда Клини. Переходим к ее обработке, стоим по конструкции Томпсона, удаляем звезду Клини, делаем один рекурсивный вызов функции, относительно нового регулярного выражения.

Если нам пришла строка из одного символа – просто строим переход для него и вызываем return.

**Интерфейсы компонента**

IEcoToolchainThompson1.h

#ifndef \_\_I\_ECO\_TOOLCHAIN\_THOMPSON\_1\_H\_\_

#define \_\_I\_ECO\_TOOLCHAIN\_THOMPSON\_1\_H\_\_

#include "IEcoBase1.h"

#include "IEcoFSM1.h"

/\* IEcoToolchainThompson1 IID = {03B395DD-613C-4A50-A65C-563400076287} \*/

#ifndef \_\_IID\_IEcoToolchainThompson1

static const UGUID IID\_IEcoToolchainThompson1 = {0x01, 0x10, 0x03, 0xB3, 0x95, 0xDD, 0x61, 0x3C, 0x4A, 0x50, 0xA6, 0x5C, 0x56, 0x34, 0x00, 0x07, 0x62, 0x87};

#endif /\* \_\_IID\_IEcoToolchainThompson1 \*/

typedef struct IEcoToolchainThompson1VTbl {

/\* IEcoUnknown \*/

int16\_t (\*QueryInterface)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ const UGUID\* riid, /\* out \*/ void \*\*ppv);

uint32\_t (\*AddRef)(/\* in \*/ struct IEcoToolchainThompson1\* me);

uint32\_t (\*Release)(/\* in \*/ struct IEcoToolchainThompson1\* me);

/\* IEcoToolchainThompson1 \*/

int16\_t(\*set\_EscapeCharacter)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ char\_t code);

int16\_t(\*set\_DisjunctionCharacter)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ char\_t code);

int16\_t(\*set\_GroupBrakets)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ char\_t leftCode, /\* in \*/ char\_t rightCode);

int16\_t(\*set\_KleeneStar)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ char\_t code);

int16\_t (\*ConstructionFA)(/\* in \*/ struct IEcoToolchainThompson1\* me, /\* in \*/ char\_t\*\* re, /\* in \*/ uint16\_t size, /\* out \*/ IEcoFSM1StateMachine\*\* pIFA);

} IEcoToolchainThompson1VTbl, \*IIEcoToolchainThompson1VTblPtr;

interface IEcoToolchainThompson1 {

struct IEcoToolchainThompson1VTbl \*pVTbl;

} IEcoToolchainThompson1;

#endif /\* \_\_I\_ECO\_TOOLCHAIN\_THOMPSON\_1\_H\_\_ \*/

**Пайплайны и их виды**

ETL

От английского – Extract, Transform, Load – извлечение, преобразование, загрузка.

Целью ETL в первую очередь является структурирование данных, фильтрация их и подготовка к дальнейшему анализу. Часто ETL используется как промежуточный слой между OLTP системами и OLAP системой или корпоративным хранилищем.

Extract

Слой Extract отвечает за извлечение данных из логов приложений, баз данных, обычных файлов с данными, других источников. Выгрузка занимает определенное время и поэтому при разработке процедуры извлечения данных необходимо четко определить частоту запросов на выгрузку.

Transform

Цель слоя Transform – преобразовать данные, привести их к более удобному для дальнейшего анализа виду. То, как данные будут преобразовываться, степень качества данных определяется исходя из требований аналитиков.

Слой Transform может выполнять самые разные функции, например:

*Очистка*

*Фильтрация*

*Обработка пропущенных значений*

*Устранение несоответствий*

*Преобразование структуры данных*

*Агрегация*

*Нормализация, Стандартизация*

Стоит еще раз отметить то, все эти функции могут быть выполнены по-разному в зависимости от требований к конечному результату. К примеру, проблема пропущенных значений может быть решена простым удалением строк или столбцов, где таких значений много. Также могут быть применены сложные алгоритмы по заполнению недостающих значений, чтобы размерность данных не уменьшилась, если это критично для дальнейшего анализа.

Load

Этот слой доставляет измененные данные в место назначения. Им может быть OLAP система или, например, следующий ETL pipeline. При очередной загрузке переносится не вся информация, а только та, которая была изменена с момента предыдущей загрузки.

Фреймворки для работы c ETL пайплайнами

*Airflow*

*Luigi*

*Gobblin*

*Apache Chukwa, Sqoop, Falcon*

*И т.д.*

Схожие подходы: ELT, ELTL, ELLT, EL now T later

ELT

Происходит извлечение и загрузка в итоговое хранилище и последующее преобразование. Преобразование выполняется средствами SQL. Недостатком является то, что ELT сильно полагается на возможность изначально хранить большие объемы необработанных данных. Из плюсов можно выделить масштабируемость, нет дополнительной задержки при загрузке данных после преобразования, так как данные уже загружены, SQL довольно хорошо обслуживает большинство преобразований для аналитики. Также, код преобразования может быть представлен как логически уровень самого хранилища данных, что упрощает настройку и обнаружение проблем.

ELTL

Происходит немедленная загрузка данных в промежуточное дешевое хранилище, откуда они преобразуются и загружаются на более продвинутый уровень представления. Полезный подход, если требуется сначала обработать данные, чтобы они соответствовали табличной структуре.

ELLT

Данные загружаются в дешевое хранилище, после чего передаются в промежуточную область хранения, на этом этапе данные преобразуются, чтобы соответствовать структуре хранилища.

EL now T later

Данные загружаются необработанными и агрегируются, согласуются только при выполнении запроса. Это может создавать проблемы с управление данными.

**Дополнительные материалы**

DSL and Language Oriented Programming

DSL ( от английского domain-specific language) – Язык программирования, специализирующийся на конкретной области применения. Построение такого языка и/или его структура данных отражает специфику решаемых с его помощью задач. Является ключевым понятием языково-ориентированного программирования.

Языково-ориентированное программирование (Language Oriented Programming) – парадигма программирования, заключающаяся в разбиении процесса разработки на стадии разработки DSL и описания решения задач с их использованием.

Примеры DSL:

TeX/LaTeX для подготовки текстовых документов

SQL для Систем Управления Базами Данных

HTML для разметки документов

Perl для манипулирования текстами

Mathematica и Maple для символьных вычислений

Unix Shell - язык интерактивный командной оболочки

Как пример DSL из личного опыта могу привести язык Lama, на котором я писал, проходя курс по компиляторам в первом полугодии. Язык Lama является ярким примером DSL, так как был создан специально для обучения студентов принципам работы компиляторов и интерпретаторов. В течение курса мы шаг за шагом писали свой компилятор для языка Lama. Насколько мне известно, Lama написан на OCaml.

Преимущества DSL

Прежде всего, стоит отметить то, что т.к. Предметно-специфический язык разрабатывается для решения определенных задач в предметной области, то описание задач на нем будет гораздо лучше отражать суть чем аналоги на языках программирования общего назначения («Построение такого языка и/или его структура данных отражает специфику решаемых с его помощью задач»). Следовательно такой DSL будет гораздо удобнее и легче в освоении для специалистов предметной области, ранее не знакомых с программированием. Также это позволяет пользователям самостоятельно расширять программные системы.

Следующим пунктом стоит отметить увеличение коэффициента повторного использования кода. Т.к. DSL инкапсулирует специфические значения предметной области, то он может быть использован не только для решения задачи, ради которой разрабатывался, но и для решения схожих задач в предметной области.

Недостатки DSL

Главные недостатки разработки DSL как подхода – высокий порог вхождения для разработчиков языков (хорошая теоретическая подготовка, свободное владение семантически разными языками программирования и их расширениями) и трудности обеспечения вычислительной производительности. Стоит отметить, что эти недостатки актуальны лишь для разработчиков DSL, конечные пользователи получают чистые преимущества.

SML (Standard ML)

Компилируемый язык программирования общего назначения высшего порядка. Является «в основном» функциональным языком, но предоставляет развитые возможности императивного программирования при необходимости. SML был первым самостоятельным компилируемым языком семейства ML.

Ориентирован на крупномасштабное программирование, предоставляет эффективные средства абстракции и модульности, обеспечивая высокий коэффициент повторного использования кода.

SML гарантирует устойчивость программ и предсказуемое поведение даже при некорректных входных данных.

**Список литературы**

Регулярные выражения и конечные автоматы

<https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions>

Пайплайны

<https://medium.com/@rchang/a-beginners-guide-to-data-engineering-part-i-4227c5c457d7>

<https://medium.com/@rchang/a-beginners-guide-to-data-engineering-part-ii-47c4e7cbda71>

<https://ru.wikipedia.org/wiki/ETL>

<https://ru.wikipedia.org/wiki/Data_mining>

<https://arxiv.org/ftp/arxiv/papers/1810/1810.04570.pdf>

<https://www.loomesoftware.com/elt>

DSL

<https://en.wikipedia.org/wiki/Language-oriented_programming>

<https://en.wikipedia.org/wiki/Domain-specific_language>

<https://en.wikipedia.org/wiki/Visual_programming_language>

<http://mikehadlow.blogspot.com/2018/10/visual-programming-why-its-bad-idea.html>

<https://en.wikipedia.org/wiki/Standard_ML>

<https://www.alibabacloud.com/blog/unlocking-the-power-of-dsls-stateless-state-machines_596467>