МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Рязанский государственный радиотехнический университет имени В.Ф. Уткина»

«К защите»

Заведующий кафедрой ВПМ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Овечкин Г.В.

«09» июня 2023 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ**

**РАБОТА**

**(бакалавриат)**

на тему

**«Разработка программного обеспечения для автоматизации создания распознавателя регулярного языка»**

Направление подготовки: 09.03.04 Программная инженерия

Наименование ОПОП: Программная инженерия

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Никичкин Б. В.)

Обучающийся \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (Гладышев Б. А.)

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Рязанский государственный радиотехнический университет

имени В.Ф. Уткина»

«УТВЕРЖДАЮ»

Заведующий кафедрой ВПМ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Овечкин Г.В.

«\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

**ЗАДАНИЕ**

на выпускную квалификационную работу

Обучающемуся Гладышеву Борису Александровичу 943 группы

(фамилия, имя, отчество, № группы)

1. Тема ВКР: Разработка программного обеспечения для автоматизации создания распознавателя регулярного языка

2. Срок представления законченной ВКР к защите: «\_\_» \_\_\_\_\_\_\_\_\_\_\_\_2023 г.

3. Руководитель: Никичкин Борис Владимирович, РГРТУ, доцент, к.т.н.

(фамилия, имя, отчество полностью, место работы, должность)

4. Исходные данные к ВКР: Операционная система: Windows 10; Язык программирования: С#; Среда разработки: Microsoft Visual Studio 2022

5. Содержание ВКР

Введение

Описание задач разработки

Разработка программного обеспечения

Тестирование программного обеспечения

Программная документация

Заключение

Список использованных источников

Приложение. Листинг наиболее значимых частей программы

6. Перечень графического (демонстрационного) материала

Постановка задачи

Разработка алгоритмов

Интерфейс программной системы

Тестирование программной системы

Дата выдачи задания: «\_\_» \_\_\_\_\_\_\_\_\_ 2023 г.

Руководитель\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Задание принял к исполнению «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г.

Обучающийся \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Реферат

Содержание

[Реферат 3](#_Toc137107835)

[Содержание 4](#_Toc137107836)

[Определения, обозначения и сокращения 6](#_Toc137107837)

[Введение 8](#_Toc137107838)

[1 Определение задач разработки генератора распознавателей 11](#_Toc137107839)

[1.1 Постановка задачи проектирования генератора распознавателей 11](#_Toc137107840)

[1.2 Обзор возможностей существующих инструментов 12](#_Toc137107841)

[1.3 Анализ требований к генератору распознавателей 16](#_Toc137107842)

[2 Проектирование генератора распознавателей 18](#_Toc137107843)

[2.1 Формальное описание конечного автомата и понятия регулярного языка 18](#_Toc137107844)

[2.2 Проектирование компонентов программной системы 20](#_Toc137107845)

[2.3 Разработка алгоритмов, используемых в ПО генератора распознавателей 21](#_Toc137107846)

[2.4 Разработка интерфейса пользователя 22](#_Toc137107847)

[3 Реализация ПО для автоматизации создания распознавателей регулярного языка 24](#_Toc137107848)

[3.1 Описание шаблона проектирования MVVM 24](#_Toc137107849)

[3.2 Реализация алгоритмов генерации исходного кода распознавателей 24](#_Toc137107850)

[3.3 Создание интерфейса программной системы 24](#_Toc137107851)

[4 Разработка программной документации к реализованному программному обеспечению для автоматизации создания распознавателей регулярного языка 25](#_Toc137107852)

[4.1 Описание применения 25](#_Toc137107853)

[4.2 Руководство оператора 25](#_Toc137107854)

[4.3 Руководство программиста 25](#_Toc137107855)

[4.4 Руководство системного программиста 25](#_Toc137107856)

[5 Тестирование ПО для автоматизации создания распознавателей регулярного языка 26](#_Toc137107857)

[5.1 Разработка плана тестирования программной системы генератора распознавателей 26](#_Toc137107858)

[5.2 Выполнение тестирования и анализ результатов 26](#_Toc137107859)

[Заключение 27](#_Toc137107860)

[Список использованных источников 28](#_Toc137107861)

[Приложение А 29](#_Toc137107862)

[A.1 Листинг генератора выходного кода распознавателей регулярного языка (Pascal) 29](#_Toc137107863)

[A.2 Листинг генератора выходного кода распознавателей регулярного языка (Python) 43](#_Toc137107864)

Определения, обозначения и сокращения

В настоящей выпускной квалификационной работе применяются следующие термины со следующими определениями

|  |  |
| --- | --- |
| Генератор распознавателей | – программное обеспечение, позволяющее автоматизировать процесс создания распознавателей регулярного языка. Обозначение введено для облегчения читаемости текста ВКР |
| КА | – конечный автомат |
| Компилятор | – программа-транслятор, преобразующая программу на каком-либо языке программирования в набор машинных кодов |
| Конечный автомат | – математическая модель устройства, которое имеет один вход, один выход и в каждый момент времени находится в одном состоянии из конечного множества возможных. Подробное описание приведено в 2.1 |
| Лексический анализ | – процесс разбора входной последовательности символов на группы (лексемы), с целью получения упорядоченной идентифицированной последовательности блоков (токенов) |
| Парсер | – синтаксический анализатор, программа, реализующая синтаксический разбор конструкций заданного языка |
| ПО | – программное обеспечение |
| РБНФ | – расширенная форма Бэкуса-Наура – система определения грамматик контекстно-свободных формальных языков, в которой одни синтаксические категории последовательно определяются через другие |
| Регулярная грамматика | – формальная грамматика 3-го типа по классификации Хомского |
| Регулярный язык | – язык, задаваемый регулярной грамматикой (язык, который может быть распознан некоторым конечным автоматом. Определения эквивалентны по теореме Клини) |
| Синтаксический анализ | – процесс сопоставления последовательности токенов с заданной грамматикой языка для определения принадлежности этому языку |
| Транслятор | – программа, которая выполняет трансляцию программы |
| Трансляция | – процесс преобразования программы, составленной на одном языке программирования, в эквивалентную программу на другом языке |
| ЯП | – язык программирования |
| DOT | – язык описания графов с форматом, понятным для чтения человека. Описание обычно представляет собой текстовый файл с расширением *.gv* или .*dot* |

Введение

Целью этой выпускной квалификационной работы является проектирование и реализация программного обеспечения для автоматизации создания распознавателей регулярного языка (далее – генератора распознавателей).

Актуальность и практическая значимость ВКР выражается в возможности применения разработанного ПО при проведении учебного процесса студентов по таким дисциплинам как, например, «Разработка компиляторов» или «Теория автоматов и формальных языков». Также сами регулярные языки широко применяются в информационных технологиях и программных системах (например, элементы грамматик языков программирования, языков разметки).

Далее представлено краткое содержание разделов выпускной квалификационной работы. Раздел 1 содержит постановку задач по проектированию и реализации ПО для автоматизации создания распознавателей регулярного языка, обоснование выбранной темы и определение возможностей применения результатов работы. Также приводится описание существующих инструментов с их возможностями и выявляются требования к разрабатываемому программному обеспечению генератора распознавателей.

Раздел 2 содержит описание основных этапов проектирования генератора распознавателей: компонентов программной системы, алгоритмов генерации выходного кода, интерфейсов приложения. В этом же разделе приведены формальные определения модели конечного автомата и понятия регулярного языка.

В разделе 3 описывается процесс реализации программной системы. Приводится определение шаблона проектирования MVVM и его применение в разрабатываемом программном обеспечении генератора распознавателей. Содержится объяснение работы алгоритмов генерации выходного кода. Раздел также включает демонстрацию реализованного интерфейса генератора распознавателей.

Этапы разработки программной документации и представлены в разделе 4. В нём содержится:

* описание применения генератора распознавателей;
* руководство оператора;
* руководство программиста;
* руководство системного программиста.

Процесс тестирования разработанной программной системы описан в разделе 5. Он включает в себя описание плана тестирования программного обеспечения генератора распознавателей, разработку тестовых сценариев, выполнение тестирования и анализ его результатов.

В ходе выполнения ВКР было разработано программное обеспечение для автоматизации создания распознавателей регулярного языка. Программный продукт представляет собой настольное .NET приложение, которое обладает следующей функциональностью:

* возможность установки параметров распознавателя для настройки допустимого регулярного языка;
* генерация выходного кода программы-распознавателя на любом из двух языков программирования: 1) Python; 2) Pascal;
* отображение графа переходов автомата-распознавателя с заданной конфигурацией.

При выполнении ВКР для реализации программной системы была использована интегрированная среда разработки Visual Studio 2022 с установленными пакетами .NET SDK версии 6. Дополнительно были использованы библиотека MS AGL (Microsoft Automatic Graph Layout) и библиотека-расширение Gu.Wpf.DataGrid2D. При проектировании системы и составлении документации для построения части диаграмм использовался инструмент PlantUML, а также сайт drawio.

Результатами ВКР являются:

* разработанное ПО для автоматизации создания распознавателей регулярного языка, отвечающее поставленным на этапе постановки задач требованиям;
* программная документация для реализованного ПО;
* описание процесса проектирования в виде пояснительной записки к ВКР.

# Определение задач разработки генератора распознавателей

## Постановка задачи проектирования генератора распознавателей

При постановке задач на проектирование ПО для автоматизации создания распознавателей регулярного языка основной целью является разработка программного инструмента, способного автоматизировать процесс создания распознавателей на основе заданной конфигурации конечного автомата. Дополнительной целью является использование созданного программного обеспечения в учебном процессе студентов, изучающих теорию формальных языков и автоматов, а также теорию компиляции.

Одна из задач состоит в определении такого интерфейса пользователя, который позволит задавать конфигурацию конечного автомата быстрым и удобным для пользователя способом. При этом стоит минимизировать количество потенциально возможных ошибок, возникающих на этом этапе. Интерфейс должен предоставлять возможность ввода алфавита конечного автомата с определением терминальных и вспомогательных символов, внутренних состояний, переходов, начального и конечных состояний.

Также важно предусмотреть возможность визуализации конфигурации конечного автомата для удобства анализа и проверки. Применительно к учебному процессу, возможность визуализации и интерактивной работы с созданными распознавателями позволит студентам нагляднее изучать принципы работы конечных автоматов и понятие регулярных языков.

Другой задачей является разработка алгоритмов, которые будут преобразовывать заданную конфигурацию конечного автомата в соответствующий выходной код программы-распознавателя регулярного языка. Эти алгоритмы должны обеспечивать корректность и оптимальность создаваемых распознавателей и быть эффективными. Также необходимо учесть различные варианты реализации распознавателей, чтобы дать пользователю возможность выбрать наиболее подходящий вариант в зависимости от требований проекта. Студентам также будет полезна такая возможность, поскольку они смогут выбирать более предпочтительный для себя язык программирования и использовать полученные результаты на практике, создавая работающие распознаватели регулярных языков.

Дополнительно, важной задачей является обеспечение гибкости и расширяемости системы. Генератор должен быть способен поддерживать различные языки программирования для генерации выходного кода и иметь возможности по управлению процессом генерации. Это позволит удовлетворить разнообразные потребности и предпочтения пользователей.

Для успешного внедрения разрабатываемой программной системы в учебный процесс необходимо создать детальную программную документацию, объясняющую принципы работы генератора распознавателей регулярного языка и его возможности.

Таким образом, постановка задач на проектирование генератора распознавателей регулярного языка включает:

* определение требований к интерфейсу пользователя;
* разработку алгоритмов преобразования конфигурации конечного автомата в выходной код программы-распознавателя;
* продумывание возможностей по обеспечению гибкости и расширяемости системы;
* написание детальной программной документации.

Этот список задач нацелен на разработку программного обеспечения для автоматизации создания распознавателей регулярного языка, которое также может быть применено в учебном процессе студентов. Использование генератора распознавателей в учебной среде позволит студентам более эффективно изучать теорию формальных языков и автоматов и применять полученные знания на практике.

## Обзор возможностей существующих инструментов

Существуют несколько инструментов, которые позволяют генерировать распознаватели регулярных языков. Большинство из них обладает значительным количеством функциональных возможностей, поскольку эти инструменты используются для создания лексических (а ANTLR и для синтаксических) анализаторов, предполагающих, в том числе, дополнительную обработку входных данных.

Каждый из этих инструментов обладает своими особенностями и преимуществами, и выбор зависит от конкретных требований проекта. Они позволяют автоматизировать процесс создания языковых инструментов, обеспечивая гибкость процесса разработки. Опишем некоторые из них.

1. **Lex/Flex**:

Lex (в UNIX-системах) и его аналог Flex (в POSIX-совместимых системах) являются одними из наиболее популярных инструментов для генерации лексических анализаторов. Они позволяют определить набор правил для сопоставления регулярных выражений с входной последовательностью символов и выполнения соответствующих действий при совпадении. Flex и Lex генерируют оптимизированный код на C/C++, который можно интегрировать в другие программы [1].

1. **ANTLR**:

ANTLR (ANother Tool for Language Recognition) - это генератор лексических и синтаксических анализаторов, поддерживающий грамматики контекстно-свободных языков [5]. Он позволяет определить грамматику языка в форме контекстно-свободных правил РБНФ и автоматически сгенерировать лексический и синтаксический анализаторы на различных языках программирования: С++, Java, C#, Python и других. ANTLR предоставляет гибкую систему атрибутов и поддерживает создание деревьев разбора для дальнейшей обработки. Инструмент имеет плагины для нескольких интегрированных сред разработки.

1. **JFlex**:

JFlex – это инструмент, аналогичный Flex, но предназначенный для Java. Он позволяет определить лексический анализатор на основе регулярных выражений и сгенерировать соответствующий Java-код. JFlex обладает гибкими возможностями, такими как поддержка Unicode-символов, а также интеграция с другими инструментами и библиотеками на языке Java [6].

1. **Ragel**:

Ragel – инструмент для спецификации и генерации конечных автоматов. Он позволяет описать состояния, переходы и действия автомата в декларативной форме, используя специальный язык. Ragel поддерживает различные типы автоматов, включая конечные автоматы Мили и Мура, и генерирует оптимизированный код на C/C++, Go, Java, Ruby и других языках. Также поддерживает операторы для управления недетерминизмом конечных автоматов, минимизацию с использованием алгоритма Хопкрофта, возможность добавления пользовательского кода в произвольные точки [7].

1. **RE2C**:

RE2C – это генератор лексических анализаторов для языка C/C++, который работает с регулярными выражениями. Он позволяет определить правила для сопоставления регулярных выражений и генерирует эффективный код на C/C++. Разрабатывался с целью генерации быстрых лексических анализаторов, производительность которых была бы не хуже, чем составленных вручную для конкретного формального языка. RE2C поддерживает различные особенности, такие как поддержка Unicode-символов и возможность сохранения состояния [4].

Несмотря на то, что большинство описанных инструментов позволяют автоматизировать процесс создания распознавателей регулярного языка и делают это эффективно, гибко, а также имеют большое количество дополнительных функциональных возможностей, они обладают несколькими недостатками, которые не позволяют внедрять их в учебный процесс.

Будучи созданными с упором на практическое применение при разработке трансляторов, парсеров, эти программные средства обладают высоким порогом вхождения, и их использование предполагает, что пользователь имеет теоретические знания о математических моделях и понятиях, которые лежат в основе работы этих инструментов, понимание интерфейсов взаимодействия с ними:

* знание синтаксиса и семантики регулярных выражений;
* владение специальными языками описания, зависящими от используемого программного средства;
* понимание механизмов работы инструмента и способов взаимодействия с ним (команды, их параметры, дополнительные опции).

То есть, уровень порога вхождения для использования описанных инструментов делает невозможным полноценное их применение (кроме ознакомительного) для теоретического изучения материала теории автоматов и формальных языков, а также связанных с ней дисциплин, поскольку значительная часть времени будет уходить на изучение самого программного средства. Такое использование времени учебного процесса расточительно и малоэффективно, а знакомство с инструментами, описанными выше или им подобными стоит проводить после усвоения теоретической части материала.

Подход к визуализации графа переходов конечного автомата-распознавателя или дерева разбора\*, используемый в описанных выше инструментах, которые обладают такой возможностью (ANTLR, re2c, Ragel), также имеет некоторые трудности для применения его в учебном процессе. При вызове процедуры визуализации они генерируют DOT-описания графов, а сам процесс получения изображений производится с помощью специальных библиотек и средств, таких как Graphviz, ZGRViewer, онлайн-инструментов или других альтернатив. При отсутствии подобного ПО в системе получение изображения из DOT-файла невозможно, но даже при наличии таких средств без использования дополнительных утилит, автоматизирующих процесс получения изображений, визуализация занимает достаточно много времени. Поэтому одним из требований к разрабатываемой программной системе было наличие возможности быстрого получения графического представления конфигурации конечного автомата без использования дополнительных средств.

\*Примечание – визуализация деревьев разбора относится только к ANTLR, который имеет возможность создания синтаксических анализаторов, и упомянута только для указания того, что подход к визуализации является тем же, что и у остальных указанных программ. В этой работе тема создания синтаксических анализаторов не рассматривается.

## Анализ требований к генератору распознавателей

Цель разрабатываемого программного обеспечения заключается в автоматизации процесса создания программ-распознавателей на основе описания конечного автомата. ПО должно генерировать исходный код программы-распознавателя на выбранном языке программирования.

Основной функциональностью сгенерированной программы-распознавателя является приём на вход последовательности символов и определение, принадлежит ли эта последовательность регулярному языку, который задается конечным автоматом. Для этого программа должна реализовывать алгоритм, который эмулирует работу конечного автомата с заданным количеством состояний, входными символами и переходами, обрабатывающего входную последовательность символов. В ходе этого процесса осуществляется проверка на её соответствие регулярному языку.

Описание конечного автомата, на основе которого будет происходить генерация исходного кода, является важным элементом входных данных для ПО. Это описание должно содержать информацию о состояниях, переходах и типах символов, определяющих регулярный язык. Поскольку одному виду символа автоматной грамматики может соответствовать более одного символа из основного алфавита терминальных символов входных последовательностей, с которыми будет работать распознаватель, также должна быть возможность сопоставления множества терминальных символов одному нетерминальному (вспомогательному) символу автоматной грамматики. Разработанное программное обеспечение должно быть способно интерпретировать такое описание и использовать его для генерации соответствующего исходного кода программы-распознавателя.

Таким образом, ПО для автоматизации создания распознавателей регулярного языка должно иметь следующие основные требования.

1. Возможность генерации исходного кода программы-распознавателя на выбранном языке программирования.

2. Поддержка ввода описания конечного автомата, включающего информацию о состояниях, переходах и символах регулярного языка.

3. Обеспечение функциональности сгенерированной программы-распознавателя для проверки принадлежности последовательности символов заданному регулярному языку.

4. Возможность визуализации графа переходов конечного автомата.

На рисунке 1 приведена диаграмма вариантов использования разрабатываемой системы, составленная по описанным ранее требованиям. Диаграммы вариантов использования позволяют более наглядно формулировать функциональные требования к системе [8].



Рисунок 1 – диаграмма вариантов использования

На диаграмме вариантов использования представлены сценарии взаимодействия пользователя с проектируемым программным обеспечением. Варианты использования выделены из требований и отражают задачи, которые пользователь сможет решать с помощью генератора распознавателей, а также действия по подготовке входных данных.

# Проектирование генератора распознавателей

## Формальное описание конечного автомата и понятия регулярного языка

Перед началом проектирования программного обеспечения стоит привести формальное описание конечного автомата, поскольку при реализации системы будет использоваться именно эта математическая модель.

**Конечным автоматом** называется упорядоченная пятёрка элементов следующего вида:

, (1)

где **Q** – конечное множество состояний автомата;

**V** – конечное множество допустимых входных символов (алфавит автомата);

*δ* – функция переходов, отображающая **V**\***Q** во множество подмножеств **Q**, то есть, ;

*q*­0 – начальное состояние автомата;

**F** – непустое множество конечных состояний автомата [1].

Работа конечного автомата представляет собой последовательность шагов, на каждом из которых автомат находится в одном из своих состояний q (текущем состоянии), принадлежащих **Q**. На следующем шаге автомат может остаться в этом же состоянии или перейти в другое под воздействием входного сигнала из алфавита **V**. В начале своей работы автомат находится в начальном состоянии q0. После этого он будет продолжать свою работу до тех пор, пока на его вход будут подаваться символы [1].

Если конечный автомат, получив на вход цепочку символов, может перейти из начального состояния в одно из конечных, то говорят, что автомат принимает цепочку символов. Если это невозможно, то конечный автомат не принимает цепочку [1].

Конечный автомат может быть детерминированным или недетерминированным. Эти два вида отличаются друг от друга возможными значениями функции переходов: в недетерминированных автоматах значением функции переходов может являться сразу несколько состояний из **Q**, то есть, некоторое подмножество. В детерминированных автоматах допустимо только одно значение функции переходов. Доказано, что для любого конечного автомата можно построить детерминированный конечный автомат, эквивалентный ему [1].

Если функция переходов детерминированного конечного автомата определена для каждой пары состояния и входного символа, то его называют полностью определённым детерминированным конечным автоматом. При построении компиляторов чаще всего используют именно такой тип.

Есть несколько способов описать конечный автомат и, в частности, его функцию переходов: перечисление пар аргументов и значений функции переходов для них, граф переходов автомата, таблица переходов. В разрабатываемой системе будет использован последний способ, поскольку он кажется наиболее удобным для описания полностью определённого детерминированного конечного автомата.

Таблица переходов автомата представляет собой таблицу, в которой в столбцах располагаются внутренние состояния автомата, а в строках – входные символы [2]. Также возможен и другой вариант размещения. На пересечении входного символа и состояния располагается ячейка со значением функции переходов при соответствующих аргументах. В полностью определённом детерминированном конечном автомате все ячейки этой таблицы будут заполнены значениями.

Разрабатываемая программная система в первую очередь должна автоматизировать процесс создания распознавателя регулярного языка, поэтому важно объяснить смысл этого понятия.

**Регулярный язык** – третий тип языков в иерархии Хомского. Регулярные языки задаются регулярными грамматиками. Они обладают самой строгой системой правил: возможны только два вида правил: A → Bγ (A → γB) или A → γ, где A, B – элементы алфавита нетерминальных символов грамматики, а γ – элемент алфавита терминальных символов. Это самый простой тип языков, его анализ требует наименьшего количества ресурсов вычислительной машины. Регулярные языки имеют широкое применение в информационных технологиях: ими описываются элементы грамматик языков программирования, мнемонические коды машинных команд, константы и многое другое [1].

По теореме Клини классы регулярных множеств и автоматных языков совпадают, поэтому для распознавания регулярных языков удобно использовать модель конечного автомата.

## Проектирование компонентов программной системы

При проектировании программной системы было принято решение разработать классы, формирующие в программе модель конечного автомата и его основных компонентов. Диаграмма классов, содержащая их представлена на рисунке 2.

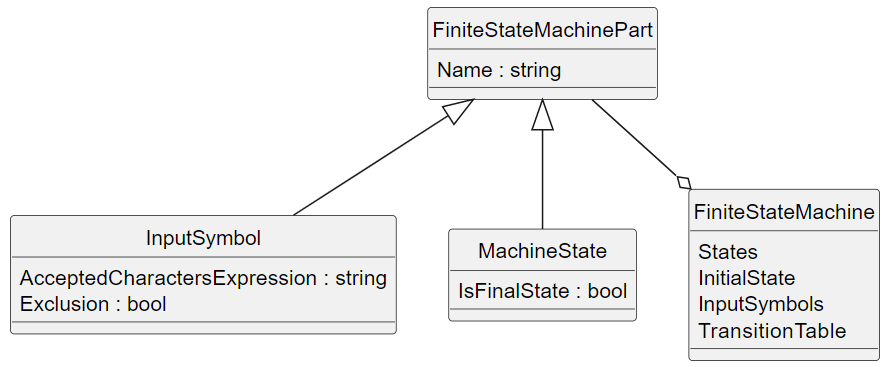


Рисунок 2 – диаграмма классов для реализации конечного автомата

На ней представлены классы для входного символа автомата, его внутреннего состояния и самого конечного автомата. Они позволяют описать, хранить и использовать конфигурацию заданного конечного автомата в программном коде.

Генераторы выходного кода распознавателей на разных языках программирования с позиции использования их в программе имеют одну функцию: создание выходного кода по заданной конфигурации конечного автомата. Конкретный вид выходного кода зависит от генератора, но структурно процесс обращения к ним представляет собой вызов функции GenerateRecognizerCode, возвращающей массив строк выходного кода программы-распознавателя. Для обобщения использования и проектирования различных генераторов кода был создан интерфейс ICodeGenerator, содержащий описанную функцию. Каждый генератор выходного кода в программе должен реализовывать его.

В качестве языков программирования (ЯП), выбор которых первоначально будет доступен в системе в качестве выходного языка, были выбраны Pascal и Python, поскольку эти языки программирования часто применяются при обучении студентов.

## Разработка алгоритмов, используемых в ПО генератора распознавателей

В книге [2] приводится обобщённая схема алгоритма создания универсального распознавателя автоматных языков:

тип tCondition [k] {множество состояний}

тип tAlpha [n]{множество входных символов}

тип (tCondition) tJump [k, n] {таблица переходов}

тип (tCondition) Fin {множество конечных состояний}

S {начальное состояние}

тип (tCondition) Cond =S; {задается текущее состояние}

char Сh; {входной символ}

while ...Есть символы

{

читать (Ch);

Cond = tJump [Cond, Ch]

}

if Cond in Fin

Цепочка принята

else

Цепочка не принята

На приведённой схеме опущен этап определения типа входного символа (сопоставление элемента из алфавита терминальных символов некоторому обобщающему символу из алфавита нетерминалов – вспомогательных символов). При создании распознавателя для конкретного языка этот этап необходимо выполнить.

Задача разрабатываемой системы – генерировать исходные коды программ-распознавателей, которые реализуют представленный выше алгоритм в соответствии с синтаксисом и семантикой выбранного языка программирования для заданных конфигураций конечных автоматов.

## Разработка интерфейса пользователя

Разрабатываемое в этой ВКР программное средство ставит одной из своих задач возможность внедрения в учебный процесс с целью улучшения усвоения теоретического материала. Для выполнения этой задачи интерфейс программного обеспечения должен требовать минимально возможного порога вхождения, необходимого для его использования.

В ходе анализа требований к программной системе было принято решение, что оконный интерфейс па наиболее хорошо подходит на эту роль, поскольку не требует дополнительного изучения специализированных языков описаний или текстовых команд. Прототип интерфейса программной системы, разработанный в соответствии с описанными в 1.3 требованиями, представлен на рисунке 3.



Рисунок 3 – протоип интерфейса

Здесь схематично представлена компоновка основных интерфейсных блоков программы. На каждой из панелей могут размещаться дополнительные специфические поля ввода (например, указание начального и конечных состояний автомата или задание символов из алфавита терминалов).

Визуализация графа переходов автомата для наглядности и более удобного контроля будет производиться в отдельном открывающемся окне, не содержащем дополнительных элементов управления.

# Реализация ПО для автоматизации создания распознавателей регулярного языка

## Описание шаблона проектирования MVVM

Архитектура приложения построена с использованием шаблона проектирования MVVM (Model-View-ViewModel, Модель-Представление-Модель Представления). Он используется для отделения модели от её представления для того, чтобы у разработчиков была возможность изменять их отдельно друг от друга [3].

Этот шаблон архитектурно делит систему на 3 части:

Модель представляет собой логику работы с данными и описание фундаментальных данных (классы, сущности, форматы данных), необходимых для работы приложения [3].

Представление – интерфейс, отображающий состояние системы пользователю. Обычно выступает подписчиком на события изменения значений свойств или команд, которые предоставляются Моделью Представления [3].

Модель Представления — с одной стороны, абстракция Представления, а с другой — обёртка данных из Модели, которая помогает обеспечивать связывание данных. То есть, она содержит Модель, преобразованную к Представлению, а также команды, которыми может пользоваться Представление, чтобы влиять на Модель [3].

В разрабатываемой системе Моделью являются классы, формирующие структуру конечного автомата и генераторы выходного кода, Представлением является интерфейс экранов приложения, описанный в XAML-разметке, а роль Модели Представления выполняет класс ViewModel, который содержит элементы, необходимые для связывания данных Модели с графическим интерфейсом и управляющие процедуры.

## Реализация алгоритмов генерации исходного кода распознавателей

## Создание интерфейса программной системы

# Разработка программной документации к реализованному программному обеспечению для автоматизации создания распознавателей регулярного языка

## Описание применения

## Руководство оператора

## Руководство программиста

## Руководство системного программиста

# Тестирование ПО для автоматизации создания распознавателей регулярного языка

## Разработка плана тестирования программной системы генератора распознавателей

## Выполнение тестирования и анализ результатов

Заключение

Список использованных источников

1. Системное программное обеспечение / А. Ю. Молчанов. - 3-е изд. - Санкт-Петербург: Питер, 2010. - 397 с.: ил. - (Учебник для вузов).; ISBN 978-5-49807-153-4

2. Теория автоматов и формальных языков: учеб. пособие / Т.Э. Шульга. Саратов: Сарат. гос. техн. ун-т, 2015. 104 с. ISBN 987-5-7433-2968-7

3. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern [сайт]. – URL: <https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern> (дата обращения: 17.05.2023)

4. re2c [сайт]. – URL: <https://re2c.org/index.html> (дата обращения: 08.06.2023)

5. Parr, Terence. (2007). The Definitive ANTLR Reference: Building Domain-specific Languages.

6. JFlex – JFlex The Fast Scanner Generator for Java [сайт]. – URL: <https://jflex.de/> (дата обращения 08.06.2023)

7. Ragel State Machine Compiler [сайт]. – URL: <https://www.colm.net/open-source/ragel/> (дата обращения 08.06.2023)

8. Разработка и анализ требований к программному обеспечению: учебник / А.А. Бубнов, С.А. Бубнов, К.А. Майков. — М.: КУРС, 2020. — 176 с. — (Бакалавриат)

Приложение А

A.1 Листинг генератора выходного кода распознавателей регулярного языка (Pascal)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

/\*

\* Тема ВКР: Разработка программного обеспечения для автоматизации создания распознавателя регулярного языка

\*

\* Разработчик: Гладышев Б. А.

\* Номер направления: 09.03.04

\* Номер группы: 943

\*

\* Руководитель ВКР: Никичкин Б. В., доцент, к.т.н.

\*

\* Средства разработки: Microsoft Visual Studio 2022, .NET 6.0

\*

\* Назначение модуля: генерация выходного кода программ-распознавателей на языке Pascal

\*

\* Дата разработки: 18 мая 2023 года

\*/

namespace RecognizerGenerator

{

internal class CodeGeneratorToPascal : ICodeGenerator

{

#region Константы для настройки имён в генерации выходной программы

#region Префиксы для констант состояний и входных символов

public const string OUT\_PREFIX\_STATE = "state\_";

public const string OUT\_PREFIX\_INPUT\_SYMBOL = "input\_symbol\_";

#endregion

#region Имена для "размерных" (количественных) констант

public const string CONSTANT\_NAME\_STATES\_COUNT = "states\_count";

public const string CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT = "input\_symbols\_count";

#endregion

#region Имена типов в секции типов (type)

public const string TYPE\_NAME\_STATE = "TState";

public const string TYPE\_NAME\_INPUT\_SYMBOL = "TInputSymbol";

#endregion

#region Имена префиксов для переменных под множества символов из реального входного потока

public const string REAL\_INPUT\_SYMBOLS\_PREFIX = "InputCharSet\_";

#endregion

#region Имена переменных в секции переменных (var)

public const string VAR\_NAME\_CURRENT\_STATE = "CurrentState";

public const string VAR\_NAME\_INPUT\_STRING\_LENGTH = "InputStringLength";

public const string VAR\_NAME\_COUNTER\_I = "i";

public const string VAR\_NAME\_TRANSITION\_TABLE = "TransitionTable";

public const string VAR\_NAME\_INPUT\_STRING = "InputString";

public const string VAR\_NAME\_SINGLE\_CHAR = "SingleChar";

public const string VAR\_NAME\_SINGLE\_CHAR\_KIND = "InputSymbolKind";

public const string VAR\_NAME\_FINAL\_STATES\_SET = "FinalStates";

#endregion

#region Сообщения программы

public const string MESSAGE\_ACCEPTED = "Допускается";

public const string MESSAGE\_REJECTED = "Не допускается";

#endregion

#endregion

/// <summary>

/// Конечный автомат, на основе которого создаётся программа-распознаватель

/// </summary>

private readonly FiniteStateMachine \_recognizerStateMachine;

/// <summary>

/// Является ли последний входной символ обобщающим для всех неохваченных терминальных символов

/// </summary>

private readonly bool \_isLastCharacterUniversal;

/// <summary>

/// Имя выходной программы

/// </summary>

public string RecognizerProgramName { get; set; } = "recognizer";

/// <summary>

/// Конструктор генератора кода

/// </summary>

/// <param name="recognizerStateMachine">Конечный автомат</param>

/// <param name="isLastCharacterUniversal">Флаг универсальности последнего символа</param>

public CodeGeneratorToPascal(FiniteStateMachine recognizerStateMachine, bool isLastCharacterUniversal)

{

\_recognizerStateMachine = recognizerStateMachine;

\_isLastCharacterUniversal = isLastCharacterUniversal;

}

/// <summary>

/// Создаёт текст программы-распознавателя по заданному конечному автомату

/// </summary>

/// <returns>Текст программы-распознавателя</returns>

public string[] GenerateRecognizerCode()

{

List<string> code = new() { GetRecognizerProgramName() };

code.AddRange(GetConstantSection());

code.Add("");

code.AddRange(GetTypeSection());

code.Add("");

code.AddRange(GetVariableSection());

code.Add("");

code.AddRange(GetProgramStatements());

return code.ToArray();

}

/// <summary>

/// Получение имени программы-распознавателя

/// </summary>

/// <returns></returns>

private string GetRecognizerProgramName()

{

return $"program {RecognizerProgramName};";

}

/// <summary>

/// Получение блока констант

/// </summary>

/// <returns></returns>

private List<string> GetConstantSection()

{

List<string> constantSection = new()

{

"const",

"{Состояния автомата}"

};

constantSection.AddRange(GetStatesConstants());

constantSection.Add("");

constantSection.Add("{Входные символы автомата}");

constantSection.AddRange(GetInputSymbolsConstants());

constantSection.Add("");

constantSection.Add("{Количество строк (состояний) и столбцов (входных символов) таблицы переходов}");

constantSection.AddRange(GetQuantitativeConstants());

return constantSection;

}

/// <summary>

/// Генерация констант под состояния автомата

/// </summary>

/// <returns></returns>

private List<string> GetStatesConstants()

{

int counter = 0;

return \_recognizerStateMachine.States.Select(

s => $"{OUT\_PREFIX\_STATE}{s} = {counter++};")

.ToList();

}

/// <summary>

/// Генерация констант под входные символы автомата

/// </summary>

/// <returns></returns>

private List<string> GetInputSymbolsConstants()

{

int counter = 0;

return \_recognizerStateMachine.InputSymbols.Select(

s => $"{OUT\_PREFIX\_INPUT\_SYMBOL}{s} = {counter++};")

.ToList();

}

/// <summary>

/// Генерация количественных констант

/// </summary>

/// <returns></returns>

private List<string> GetQuantitativeConstants()

{

return new List<string>()

{

$"{CONSTANT\_NAME\_STATES\_COUNT} = {\_recognizerStateMachine.States.Count};",

$"{CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT} = {\_recognizerStateMachine.InputSymbols.Count};"

};

}

/// <summary>

/// Генерация секции типов выходной программы

/// </summary>

/// <returns></returns>

private List<string> GetTypeSection()

{

List<MachineState> states = \_recognizerStateMachine.States;

List<InputSymbol> inputSymbols = \_recognizerStateMachine.InputSymbols;

return new List<string>()

{

"type",

"{Тип состояния автомата}",

$"{TYPE\_NAME\_STATE} = {OUT\_PREFIX\_STATE}{states[0]}..{OUT\_PREFIX\_STATE}{states.Last()};",

"",

"{Тип входного символа автомата}",

$"{TYPE\_NAME\_INPUT\_SYMBOL} = {OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbols[0]}..{OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbols.Last()};"

};

}

/// <summary>

/// Генерация секции переменных

/// </summary>

/// <returns></returns>

private List<string> GetVariableSection()

{

List<string> variableSection = new()

{

"var",

"{Текущее состояние автомата в момент работы}",

$"{VAR\_NAME\_CURRENT\_STATE} : {TYPE\_NAME\_STATE};",

$"{VAR\_NAME\_INPUT\_STRING\_LENGTH} : integer;",

$"{VAR\_NAME\_COUNTER\_I} : integer;",

$"{VAR\_NAME\_TRANSITION\_TABLE} : array[0..{CONSTANT\_NAME\_STATES\_COUNT} - 1, 0..{CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT} - 1] of {TYPE\_NAME\_STATE};",

$"{VAR\_NAME\_INPUT\_STRING} : string;",

"",

"{Обрабатываемый символ из входного потока}",

$"{VAR\_NAME\_SINGLE\_CHAR} : char;",

"",

"{Тип обрабатываемого символа}",

$"{VAR\_NAME\_SINGLE\_CHAR\_KIND} : {TYPE\_NAME\_INPUT\_SYMBOL};",

"",

"{Конечные состояния автомата}",

$"{VAR\_NAME\_FINAL\_STATES\_SET} : set of {TYPE\_NAME\_STATE};",

"",

"{Множества для сопоставления реальных поступающих символов их типам}"

};

variableSection.AddRange(\_recognizerStateMachine.InputSymbols.Select(

s => $"{REAL\_INPUT\_SYMBOLS\_PREFIX}{s} : set of char;"));

return variableSection;

}

/// <summary>

/// Генерация программных блоков (между верхними begin, end)

/// </summary>

/// <returns></returns>

private List<string> GetProgramStatements()

{

List<string> statements = new() { "begin" };

statements.AddRange(GetDataInitializationBlock());

statements.AddRange(GetWhileLoopStatement());

statements.AddRange(GetEndChecking());

statements.Add("end.");

return statements;

}

/// <summary>

/// Генерация блока инициализации в выходной программе

/// </summary>

/// <returns></returns>

private List<string> GetDataInitializationBlock()

{

List<string> dataInitialization = new() { "{Определение соответствия реальных символов входным символам автомата}" };

dataInitialization.AddRange(GetInputSymbolSets());

dataInitialization.Add("");

dataInitialization.Add("{Заполнение таблицы переходов автомата}");

dataInitialization.AddRange(GetTransitionTableInitialization());

dataInitialization.Add("{Считывание входной строки}");

dataInitialization.AddRange(GetInputStringReading());

dataInitialization.Add("");

dataInitialization.AddRange(GetVariableInitialization());

return dataInitialization;

}

/// <summary>

/// Настройка сопоставления терминальных символов нетерминальным (входным, вспомогательным) символам

/// </summary>

/// <returns></returns>

private List<string> GetInputSymbolSets()

{

List<string> symbolSets = new();

// для каждого входного символа формируется соответствующее множество со значениями из пользовательского ввода

foreach (InputSymbol inputSymbol in \_recognizerStateMachine.InputSymbols)

{

StringBuilder symbolsSetInitializationExpression = new(

$"{REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name} := [");

symbolsSetInitializationExpression.Append(GetRealCharactersSetExpression(inputSymbol.AcceptedCharactersExpression));

symbolsSetInitializationExpression.Append("];");

symbolSets.Add(symbolsSetInitializationExpression.ToString());

}

return symbolSets;

}

/// <summary>

/// Получение выражения для инициализации множества терминальных символов при сопоставлении

/// Упрощённая версия концепции скобочных выражений POSIX

/// </summary>

/// <param name="acceptedCharactersExpression">Выражение для определения обрабатываемых символов</param>

/// <returns></returns>

private static StringBuilder GetRealCharactersSetExpression(string acceptedCharactersExpression)

{

// распознаёт и захватывает диапазоны непробельных символов вида <начальный символ>-<конечный символ>

// примеры: A-Z, f-q, !-5,

// а также одиночные символы: s, t, \_, \*, +

Regex charRangeRegex = new(@"(?<rangeStart>\S)\-(?<rangeEnd>\S)|(?<single>\S)", RegexOptions.ExplicitCapture);

StringBuilder setInitializationExpression = new();

MatchCollection matchedCharRanges = charRangeRegex.Matches(acceptedCharactersExpression);

foreach (Match match in matchedCharRanges.Cast<Match>())

{

if (match.Groups["single"].Success)

{

// захват одиночного символа

string single = match.Groups["single"].Value;

setInitializationExpression.Append($"\'{single}\', ");

}

else

{

// захват диапазона символов

string start = match.Groups["rangeStart"].Value;

string end = match.Groups["rangeEnd"].Value;

setInitializationExpression.Append($"\'{start}\'..\'{end}\', ");

}

}

// удаление последней запятой с пробелом

if (setInitializationExpression.Length >= 2)

setInitializationExpression.Remove(setInitializationExpression.Length - 2, 2);

return setInitializationExpression;

}

/// <summary>

/// Генерация заполнения таблицы переходов

/// </summary>

/// <returns></returns>

private List<string> GetTransitionTableInitialization()

{

List<MachineState> states = \_recognizerStateMachine.States;

List<InputSymbol> inputSymbols = \_recognizerStateMachine.InputSymbols;

List<string> transitionTableInitialization = new();

for (int i = 0; i < states.Count; i++)

{

for (int j = 0; j < inputSymbols.Count; j++)

{

transitionTableInitialization.Add(

$"{VAR\_NAME\_TRANSITION\_TABLE}[{OUT\_PREFIX\_STATE}{states[i].Name}, {OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbols[j].Name}]" +

$" := {OUT\_PREFIX\_STATE}{\_recognizerStateMachine.TransitionTable[i][j].Name};");

}

transitionTableInitialization.Add("");

}

return transitionTableInitialization;

}

/// <summary>

/// Генерация блока чтения входной последовательности

/// </summary>

/// <returns></returns>

private static List<string> GetInputStringReading()

{

return new()

{

$"readln({VAR\_NAME\_INPUT\_STRING});",

$"{VAR\_NAME\_INPUT\_STRING\_LENGTH} := Length({VAR\_NAME\_INPUT\_STRING});"

};

}

/// <summary>

/// Генерация блока инициализации переменных

/// </summary>

/// <returns></returns>

private List<string> GetVariableInitialization()

{

StringBuilder finalStates = new();

foreach (MachineState state in \_recognizerStateMachine.States)

{

if (state.IsFinalState)

finalStates.Append($"{OUT\_PREFIX\_STATE}{state}, ");

}

// удаление последней запятой с пробелом

if (finalStates.Length > 2)

finalStates.Remove(finalStates.Length - 2, 2);

return new()

{

"{Определение конечных состояний}",

$"{VAR\_NAME\_FINAL\_STATES\_SET} := [{finalStates}];",

"",

"{Установка автомата в начальное состояние}",

$"{VAR\_NAME\_CURRENT\_STATE} := {OUT\_PREFIX\_STATE}{\_recognizerStateMachine.InitialState.Name};",

$"{VAR\_NAME\_COUNTER\_I} := 1;"

};

}

/// <summary>

/// Генерация блока цикла обработки входной строки

/// </summary>

/// <returns></returns>

private List<string> GetWhileLoopStatement()

{

List<string> loopStatement = new()

{

"",

"{Пока в строке не кончились символы}",

$"while {VAR\_NAME\_COUNTER\_I} <= {VAR\_NAME\_INPUT\_STRING\_LENGTH} do",

"begin",

"{Взятие очередного символа и определение его типа}",

$"{VAR\_NAME\_SINGLE\_CHAR} := {VAR\_NAME\_INPUT\_STRING}[{VAR\_NAME\_COUNTER\_I}];"

};

loopStatement.AddRange(GetIfStatements());

loopStatement.Add("");

loopStatement.Add("{Переход автомата в новое состояние и смещение на следующий символ}");

loopStatement.Add($"{VAR\_NAME\_CURRENT\_STATE} := {VAR\_NAME\_TRANSITION\_TABLE}[{VAR\_NAME\_CURRENT\_STATE}, {VAR\_NAME\_SINGLE\_CHAR\_KIND}];");

loopStatement.Add($"{VAR\_NAME\_COUNTER\_I} := {VAR\_NAME\_COUNTER\_I} + 1;");

loopStatement.Add("end;");

return loopStatement;

}

/// <summary>

/// Получение блока определения типов символов

/// </summary>

/// <returns></returns>

private List<string> GetIfStatements()

{

List<string> ifStatements = new();

List<InputSymbol> symbolsInConditions = \_isLastCharacterUniversal

? \_recognizerStateMachine.InputSymbols.Take(\_recognizerStateMachine.InputSymbols.Count - 1).ToList()

: \_recognizerStateMachine.InputSymbols;

foreach (InputSymbol inputSymbol in symbolsInConditions)

{

if (inputSymbol.Excusion)

ifStatements.Add($"if not ({VAR\_NAME\_SINGLE\_CHAR} in {REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name}) then");

else

ifStatements.Add($"if {VAR\_NAME\_SINGLE\_CHAR} in {REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name} then");

ifStatements.Add($"{VAR\_NAME\_SINGLE\_CHAR\_KIND} := {OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbol.Name}");

ifStatements.Add("else");

}

if (\_isLastCharacterUniversal)

ifStatements.Add($"{VAR\_NAME\_SINGLE\_CHAR\_KIND} := {OUT\_PREFIX\_INPUT\_SYMBOL}{\_recognizerStateMachine.InputSymbols.Last().Name};");

else

{

ifStatements.Add("begin");

ifStatements.Add($"writeln(\'{MESSAGE\_REJECTED}\');");

ifStatements.Add("exit;");

ifStatements.Add("end;");

}

return ifStatements;

}

/// <summary>

/// Получение блока генерации ответа распознавателя

/// </summary>

/// <returns></returns>

private static List<string> GetEndChecking()

{

return new()

{

"",

"{Вывод результата}",

$"if {VAR\_NAME\_CURRENT\_STATE} in {VAR\_NAME\_FINAL\_STATES\_SET} then",

$"writeln(\'{MESSAGE\_ACCEPTED}\')",

$"else",

$"writeln(\'{MESSAGE\_REJECTED}\')"

};

}

}

}

A.2 Листинг генератора выходного кода распознавателей регулярного языка (Python)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

/\*

\* Тема ВКР: Разработка программного обеспечения для автоматизации создания распознавателя регулярного языка

\*

\* Разработчик: Гладышев Б. А.

\* Номер направления: 09.03.04

\* Номер группы: 943

\*

\* Руководитель ВКР: Никичкин Б. В., доцент, к.т.н.

\*

\* Средства разработки: Microsoft Visual Studio 2022, .NET 6.0

\*

\* Назначение модуля: генерация выходного кода программ-распознавателей на языке Python

\*

\* Дата разработки: 6 июня 2023 года

\*/

namespace RecognizerGenerator

{

internal class CodeGeneratorToPython3 : ICodeGenerator

{

#region Константы для настройки имён в генерации выходной программы

#region Префиксы для констант состояний и входных символов

public const string OUT\_PREFIX\_STATE = "state\_";

public const string OUT\_PREFIX\_INPUT\_SYMBOL = "input\_symbol\_";

#endregion

#region Имена для "размерных" (количественных) констант

public const string CONSTANT\_NAME\_STATES\_COUNT = "states\_count";

public const string CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT = "input\_symbols\_count";

#endregion

#region Имена префиксов для переменных под множества символов из реального входного потока

public const string REAL\_INPUT\_SYMBOLS\_PREFIX = "input\_char\_set\_";

#endregion

#region Название функции для получения диапазона символов

private const string FUNCTION\_CHAR\_RANGE\_NAME = "char\_range";

#endregion

#region Имена переменных в секции переменных

public const string VAR\_NAME\_CURRENT\_STATE = "current\_state";

public const string VAR\_NAME\_TRANSITION\_TABLE = "transition\_table";

public const string VAR\_NAME\_INPUT\_STRING = "input\_string";

public const string VAR\_NAME\_SINGLE\_CHAR = "single\_char";

public const string VAR\_NAME\_SINGLE\_CHAR\_KIND = "input\_symbol\_kind";

public const string VAR\_NAME\_FINAL\_STATES\_SET = "final\_states";

#endregion

#region Сообщения программы

public const string MESSAGE\_ACCEPTED = "Допускается";

public const string MESSAGE\_REJECTED = "Не допускается";

#endregion

#endregion

/// <summary>

/// Конечный автомат, на основе которого создаётся программа-распознаватель

/// </summary>

private readonly FiniteStateMachine \_recognizerStateMachine;

/// <summary>

/// Является ли последний входной символ обобщающим для всех неохваченных терминальных символов

/// </summary>

private readonly bool \_isLastCharacterUniversal;

/// <summary>

/// Свойство для генерации согласованных отступов

/// </summary>

private static string Tab { get; } = " ";

/// <summary>

/// Конструктор генератора кода

/// </summary>

/// <param name="recognizerStateMachine">Конечный автомат</param>

/// <param name="isLastCharacterUniversal">Флаг универсальности последнего символа</param>

public CodeGeneratorToPython3(FiniteStateMachine recognizerStateMachine, bool isLastCharacterUniversal)

{

\_recognizerStateMachine = recognizerStateMachine;

\_isLastCharacterUniversal = isLastCharacterUniversal;

}

/// <summary>

/// Создаёт текст программы-распознавателя по заданному конечному автомату

/// </summary>

/// <returns>Текст программы-распознавателя</returns>

public string[] GenerateRecognizerCode()

{

List<string> code = new();

code.AddRange(GetConstantSection());

code.Add("");

code.Add("");

code.AddRange(GetCharRangeFunctionSection());

code.Add("");

code.Add("");

code.AddRange(GetProgramStatements());

return code.ToArray();

}

/// <summary>

/// Получение блока констант

/// </summary>

/// <returns></returns>

private List<string> GetConstantSection()

{

List<string> constantSection = new()

{

"# Состояния автомата"

};

constantSection.AddRange(GetStatesConstants());

constantSection.Add("");

constantSection.Add("# Входные символы автомата");

constantSection.AddRange(GetInputSymbolsConstants());

constantSection.Add("");

constantSection.Add("# Количество строк (состояний) и столбцов (входных символов) таблицы переходов");

constantSection.AddRange(GetQuantitativeConstants());

return constantSection;

}

/// <summary>

/// Генерация констант под состояния автомата

/// </summary>

/// <returns></returns>

private List<string> GetStatesConstants()

{

int counter = 0;

return \_recognizerStateMachine.States.Select(

s => $"{OUT\_PREFIX\_STATE}{s} = {counter++}")

.ToList();

}

/// <summary>

/// Генерация констант под входные символы автомата

/// </summary>

/// <returns></returns>

private List<string> GetInputSymbolsConstants()

{

int counter = 0;

return \_recognizerStateMachine.InputSymbols.Select(

s => $"{OUT\_PREFIX\_INPUT\_SYMBOL}{s} = {counter++}")

.ToList();

}

/// <summary>

/// Генерация количественных констант

/// </summary>

/// <returns></returns>

private List<string> GetQuantitativeConstants()

{

return new List<string>()

{

$"{CONSTANT\_NAME\_STATES\_COUNT} = {\_recognizerStateMachine.States.Count}",

$"{CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT} = {\_recognizerStateMachine.InputSymbols.Count}"

};

}

/// <summary>

/// Получение текста функции создания диапазона символов. Необходима

/// для инициализации множеств символов алфавита терминальных символов

/// входных последовательностей

/// </summary>

/// <returns></returns>

private static List<string> GetCharRangeFunctionSection()

{

return new List<string>()

{

$"def {FUNCTION\_CHAR\_RANGE\_NAME}(c1: str, c2: str):",

$"{Tab}return {{chr(c) for c in range(ord(c1), ord(c2) + 1)}}"

};

}

/// <summary>

/// Генерация программных блоков (определение функции распознавателя и её вызов)

/// </summary>

/// <returns></returns>

private List<string> GetProgramStatements()

{

List<string> statements = new() { "def recognizer():" };

statements.AddRange(GetDataInitializationBlock());

statements.AddRange(GetLoopStatement());

statements.AddRange(GetEndChecking());

statements.Add("");

statements.Add("");

statements.Add("recognizer()");

return statements;

}

/// <summary>

/// Генерация блока инициализации в выходной программе

/// </summary>

/// <returns></returns>

private List<string> GetDataInitializationBlock()

{

List<string> dataInitialization = new() { $"{Tab}# Определение соответствия реальных символов входным символам автомата" };

dataInitialization.AddRange(GetInputSymbolSets());

dataInitialization.Add("");

dataInitialization.Add($"{Tab}# Заполнение таблицы переходов автомата");

dataInitialization.AddRange(GetTransitionTableInitialization());

dataInitialization.Add($"{Tab}# Считывание входной строки");

dataInitialization.AddRange(GetInputStringReading());

dataInitialization.Add("");

dataInitialization.AddRange(GetVariableInitialization());

return dataInitialization;

}

/// <summary>

/// Настройка сопоставления терминальных символов нетерминальным (входным, вспомогательным) символам

/// </summary>

/// <returns></returns>

private List<string> GetInputSymbolSets()

{

List<string> symbolSets = new();

// для каждого входного символа формируется соответствующее множество со значениями из пользовательского ввода

foreach (InputSymbol inputSymbol in \_recognizerStateMachine.InputSymbols)

{

StringBuilder symbolsSetInitializationExpression = new(

$"{Tab}{REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name} = {{");

symbolsSetInitializationExpression.Append(GetRealCharactersSetExpression(inputSymbol.AcceptedCharactersExpression));

symbolsSetInitializationExpression.Append('}');

symbolSets.Add(symbolsSetInitializationExpression.ToString());

}

return symbolSets;

}

/// <summary>

/// Получение выражения для инициализации множества терминальных символов при сопоставлении

/// Упрощённая версия концепции скобочных выражений POSIX

/// </summary>

/// <param name="acceptedCharactersExpression">Выражение для определения обрабатываемых символов</param>

/// <returns></returns>

private static StringBuilder GetRealCharactersSetExpression(string acceptedCharactersExpression)

{

// распознаёт и захватывает диапазоны непробельных символов вида <начальный символ>-<конечный символ>

// примеры: A-Z, f-q, !-5,

// а также одиночные символы: s, t, \_, \*, +

Regex charRangeRegex = new(@"(?<rangeStart>\S)\-(?<rangeEnd>\S)|(?<single>\S)", RegexOptions.ExplicitCapture);

StringBuilder setInitializationExpression = new();

MatchCollection matchedCharRanges = charRangeRegex.Matches(acceptedCharactersExpression);

foreach (Match match in matchedCharRanges.Cast<Match>())

{

if (match.Groups["single"].Success)

{

// захват одиночного символа

string single = match.Groups["single"].Value;

setInitializationExpression.Append($"\'{single}\', ");

}

else

{

// захват диапазона символов

string start = match.Groups["rangeStart"].Value;

string end = match.Groups["rangeEnd"].Value;

setInitializationExpression.Append($"\*{FUNCTION\_CHAR\_RANGE\_NAME}(\'{start}\', \'{end}\'), ");

}

}

// удаление последней запятой с пробелом

if (setInitializationExpression.Length >= 2)

setInitializationExpression.Remove(setInitializationExpression.Length - 2, 2);

return setInitializationExpression;

}

/// <summary>

/// Генерация заполнения таблицы переходов

/// </summary>

/// <returns></returns>

private List<string> GetTransitionTableInitialization()

{

List<MachineState> states = \_recognizerStateMachine.States;

List<InputSymbol> inputSymbols = \_recognizerStateMachine.InputSymbols;

List<string> transitionTableInitialization = new()

{

$"{Tab}{VAR\_NAME\_TRANSITION\_TABLE} = tuple([0 for \_ in range({CONSTANT\_NAME\_INPUT\_SYMBOLS\_COUNT})] for \_ in range({CONSTANT\_NAME\_STATES\_COUNT}))"

};

for (int i = 0; i < states.Count; i++)

{

for (int j = 0; j < inputSymbols.Count; j++)

{

transitionTableInitialization.Add(

$"{Tab}{VAR\_NAME\_TRANSITION\_TABLE}[{OUT\_PREFIX\_STATE}{states[i].Name}][{OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbols[j].Name}]" +

$" = {OUT\_PREFIX\_STATE}{\_recognizerStateMachine.TransitionTable[i][j].Name}");

}

transitionTableInitialization.Add("");

}

return transitionTableInitialization;

}

/// <summary>

/// Генерация блока чтения входной последовательности

/// </summary>

/// <returns></returns>

private static List<string> GetInputStringReading()

{

return new()

{

$"{Tab}{VAR\_NAME\_INPUT\_STRING} = input()"

};

}

/// <summary>

/// Генерация блока инициализации переменных

/// </summary>

/// <returns></returns>

private List<string> GetVariableInitialization()

{

StringBuilder finalStates = new();

foreach (MachineState state in \_recognizerStateMachine.States)

{

if (state.IsFinalState)

finalStates.Append($"{OUT\_PREFIX\_STATE}{state}, ");

}

// удаление последней запятой с пробелом

if (finalStates.Length > 2)

finalStates.Remove(finalStates.Length - 2, 2);

return new()

{

$"{Tab}# Определение конечных состояний",

$"{Tab}{VAR\_NAME\_FINAL\_STATES\_SET} = {{{finalStates}}}",

"",

$"{Tab}# Установка автомата в начальное состояние",

$"{Tab}{VAR\_NAME\_CURRENT\_STATE} = {OUT\_PREFIX\_STATE}{\_recognizerStateMachine.InitialState.Name}"

};

}

/// <summary>

/// Генерация блока цикла обработки входной строки

/// </summary>

/// <returns></returns>

private List<string> GetLoopStatement()

{

List<string> loopStatement = new()

{

"",

$"{Tab}for {VAR\_NAME\_SINGLE\_CHAR} in {VAR\_NAME\_INPUT\_STRING}:",

$"{Tab}{Tab}# Взятие очередного символа и определение его типа"

};

loopStatement.AddRange(GetIfStatements());

loopStatement.Add("");

loopStatement.Add($"{Tab}{Tab}# Переход автомата в новое состояние и смещение на следующий символ");

loopStatement.Add($"{Tab}{Tab}{VAR\_NAME\_CURRENT\_STATE} = {VAR\_NAME\_TRANSITION\_TABLE}[{VAR\_NAME\_CURRENT\_STATE}][{VAR\_NAME\_SINGLE\_CHAR\_KIND}]");

return loopStatement;

}

/// <summary>

/// Получение блока определения типов символов

/// </summary>

/// <returns></returns>

private List<string> GetIfStatements()

{

List<string> ifStatements = new();

List<InputSymbol> symbolsInConditions = \_isLastCharacterUniversal

? \_recognizerStateMachine.InputSymbols.Take(\_recognizerStateMachine.InputSymbols.Count - 1).ToList()

: \_recognizerStateMachine.InputSymbols;

int counter = 0;

foreach (InputSymbol inputSymbol in symbolsInConditions)

{

StringBuilder expr = counter++ > 0 ? new($"{Tab}{Tab}elif ") : new($"{Tab}{Tab}if ");

if (inputSymbol.Excusion)

expr.Append($"not {VAR\_NAME\_SINGLE\_CHAR} in {REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name}:");

else

expr.Append($"{VAR\_NAME\_SINGLE\_CHAR} in {REAL\_INPUT\_SYMBOLS\_PREFIX}{inputSymbol.Name}:");

ifStatements.Add(expr.ToString());

ifStatements.Add($"{Tab}{Tab}{Tab}{VAR\_NAME\_SINGLE\_CHAR\_KIND} = {OUT\_PREFIX\_INPUT\_SYMBOL}{inputSymbol.Name}");

}

ifStatements.Add($"{Tab}{Tab}else:");

if (\_isLastCharacterUniversal)

ifStatements.Add($"{Tab}{Tab}{Tab}{VAR\_NAME\_SINGLE\_CHAR\_KIND} = {OUT\_PREFIX\_INPUT\_SYMBOL}{\_recognizerStateMachine.InputSymbols.Last().Name}");

else

{

ifStatements.Add($"{Tab}{Tab}{Tab}print(\'{MESSAGE\_REJECTED}\')");

ifStatements.Add($"{Tab}{Tab}{Tab}return");

}

return ifStatements;

}

/// <summary>

/// Получение блока генерации ответа распознавателя

/// </summary>

/// <returns></returns>

private static List<string> GetEndChecking()

{

return new()

{

"",

$"{Tab}# Вывод результата",

$"{Tab}if {VAR\_NAME\_CURRENT\_STATE} in {VAR\_NAME\_FINAL\_STATES\_SET}:",

$"{Tab}{Tab}print(\'{MESSAGE\_ACCEPTED}\')",

$"{Tab}else:",

$"{Tab}{Tab}print(\'{MESSAGE\_REJECTED}\')"

};

}

}

}