Федеральное государственное бюджетное образовательное
учреждение высшего образования
«МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. Н. П. ОГАРЁВА»

Институт электроники и светотехники

Кафедра автоматизированных систем обработки информации и управления

КУРСОВОЙ ПРОЕКТ

РАЗРАБОТКА ПРОГРАММЫ ДЛЯ АВТОМАТИЗИРОВАННОГО ПОСТРОЕНИЯ ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ ПО ПРАВИЛАМ ЗАДАННЫМ В ВИДЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Автор курсового проекта _____ 15.12.2017 О. В. Дыдыкин
(подпись)

Специальность 090301 информатика и вычислительная техника

Обозначение курсового проекта КП-02069964-09.03.01.62-4-17

Руководитель проекта _____ 15.12.2017 С. А. Ямашкин
(подпись)

Оценка _____

Саранск
2017

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. Н. П. ОГАРЁВА»

Институт электроники и светотехники
Кафедра автоматизированных систем обработки информации и управления

ЗАДАНИЕ НА КУРСОВОЙ ПРОЕКТ

Студент Дыдыкин Олег Владимирович

1. Тема: «Разработка программы для автоматизированного построения лексических анализаторов по правилам заданным регулярными выражениями»
2. Срок представления проекта к защите: _____
3. Исходные данные для научного исследования: учебные пособия, интернет-источники
4. Создать программный продукт для облегчения написания лексических анализаторов которые впоследствии можно легко встраивать в программы для обработки текстовых данных таких как исходные коды программ, в том числе подсветки синтаксиса.

Руководитель работы _____ _____
подпись, дата инициалы, фамилия

Задание принял к исполнению _____
дата, подпись

Реферат

Курсовой проект содержит 16 листов, 3 рисунка, 6 источников, 1 приложение.

ПРОГРАММИРОВАНИЕ, АВТОМАТИЗИРОВАННАЯ ИНФОРМАЦИОННАЯ СИСТЕМА, РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ, КОНЕЧНЫЙ АВТОМАТ, ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, C#.

Объектом является программа генератор лексических анализаторов на основе заданных пользователем правил.

Целью разработки является программа способная на основании формальных правил представленных в виде набора регулярных выражений построить лексический анализатор в виде конечного автомата разбирающего текст за один проход. Повысить знания в области ООП. Придерживаться основных понятий: инкапсуляция, наследование, полиморфизм.

В процессе работы проводилась конструктивная проработка методов разработки и усовершенствования приложения для удобства пользователя и большей производительности.

Степень внедрения — полезный инструмент для разработчиков формальных языков.

Требования к оборудованию:

ОС Microsoft Windows Vista/7/10, Процессор Intel Core i3 1.33 GHz или выше 64 MB RAM , 10 MB свободного места на жестком диске, пакет .NET Framework версии 4.0 или выше.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данного курсового проекта является разработка программы для автоматического построения лексического анализатора по набору правил представленных в виде регулярных выражений. В процессе разработки должны быть усвоены некоторые новые возможности Visual Studio 2017.

Для написания генераторов лексических анализаторов обычно не используются языки высокого уровня такие как C#. Однако данный язык позволяет упростить разработку интерфейса пользователя для программного продукта. Благодаря тому, что C# объектно ориентированный язык он позволяет упростить работу с такими структурами как деревья. Так как C# как и Java выполняется на виртуальной машине программу на этом языке можно запустить в любой операционной системе для которой написана соответствующая виртуальная машина.

Актуальность: программы для построения лексических анализаторов часто применяются при разработке компиляторов для языков программирования. Предоставление разработчику удобного инструмента для лексического анализа позволяет существенно сократить время на написание компилятора и потратить освободившееся время на более сложные задачи.

Цель: предоставить программный продукт позволяющий при помощи набора формальных правил представленных регулярными выражениями построить конечный автомат разбирающий входную строку на набор именованных элементов называемых токенами.

Задачи:

1. Анализ предметной области и опыта решения схожих задач
2. Проектирование программного продукта
3. Программная реализация
4. Отладка решения
5. Описание результатов работы

Программа будет разрабатываться на языке C# с применением принципов объектно-ориентированного программирования.

1. Анализ предметной области

Разработка генератора лексических анализаторов подразумевает наличие знаний о базовой структуре лексического анализатора, понятии конечного автомата и способов его описания.

В ходе анализа было определено что одним из самых популярных способов описания лексики являются регулярные выражения благодаря их компактности и выразительности. Также было выявлено несколько аналогов: lex, jlex, flex. Все три являются консольными программами отсюда и проистекает один из главных недостатков — необходимость постоянно переключаться между редактором и консолью для сборки. Намного удобнее работать в одном окне.

Lex и flex создают программу на языке C которая подразумевает обмен данными через каналы или консоль, формат обмена программист определяет сам. Это достаточно универсальное решение однако оно не всегда необходимо. Встроенная функция позволяет более точно контролировать поток токенов.

JLex является узкоспециализированным и применяется только для проектов на Java.

Все три программы имеют весь необходимый функционал и хорошо отлажены однако из-за того что были написаны очень давно сохраняют довольно много кода для обратной совместимости. Язык описания лексических правил тоже необычен и поначалу может запутать пользователя.

2. Проектирование

2.1. Основные задачи

В ходе анализа были выявлены следующие недостатки существующих решений:

1. Необычный синтаксис и как следствие медленное освоение
2. Создают утилиту, код нельзя встроить
3. Код полученной утилиты сложно читать и соответственно разобраться в проблеме если что-то пойдет не так.
4. Имеют консольный интерфейс.

Следовательно, задачами при создании нового инструмента будут:

1. Подобрать более выразительный синтаксис
2. Создать миниатюрную встраиваемую функцию
3. Легко читаемый код функции. Все вспомогательные функции должны иметь осмысленные названия.
4. Оконный интерфейс.

2.2. Синтаксис

В качестве синтаксиса была выбрана концепция схожая с объявлением переменных в C-подобных языках. Пример:

```
token identifier1(\w+)
token identifier2(\d+)
```

Большей читаемости можно добиться ограничив длину строки конечной программы примерно до 40-50 символов и выводя одинаковые операции одна за другой так, что бы они визуально образовывали таблицу. Также программа должна как можно точнее определять место возникновения ошибки, её причину и возможные способы устранения.

Количество вспомогательных функций минимально:

1. Поместить токен в стек

   ```
   void push_token(token_t token, size_t begin, size_t end);
   ```

2. Извлечь токен из стека

   ```
   token_t pop_token();
   ```

3. Границы найденного токена

   ```
   selection_t get_token_str();
   ```

4. Основные классы символов \s, \d. \w соответственно

   ```
   bool ws(char ch);
   bool digit(char ch);
   bool word(char ch);
   ```

5. Сохранить и загрузить состояние автомата

   void push_state(states_t *states, unsigned state, unsigned substate);

   void pop_state(states_t *states, unsigned *state, unsigned *substate);

6. Сама функция:

   size_t lex(const lchar *str, size_t str_lenght);

2.2. Внутреннее представление

Для того что бы программа генерировала оптимизированный конечный автомат применяется дерево похожее по своей организации на префиксное дерево с той лишь только разницей что узлом этого дерева может быть не только конкретный символ, но и некоторый класс символов. Также Дерево может содержать циклические ссылки. Обусловлено это наличием таких операторов как Звезда Клини (*) и Плюс Клини (+) а также оператора условия (?) которые описывают соответственно, повторение заданного набора ноль или более раз, повторение заданного набора один или более раз и повторение заданного набора ноль или один раз.

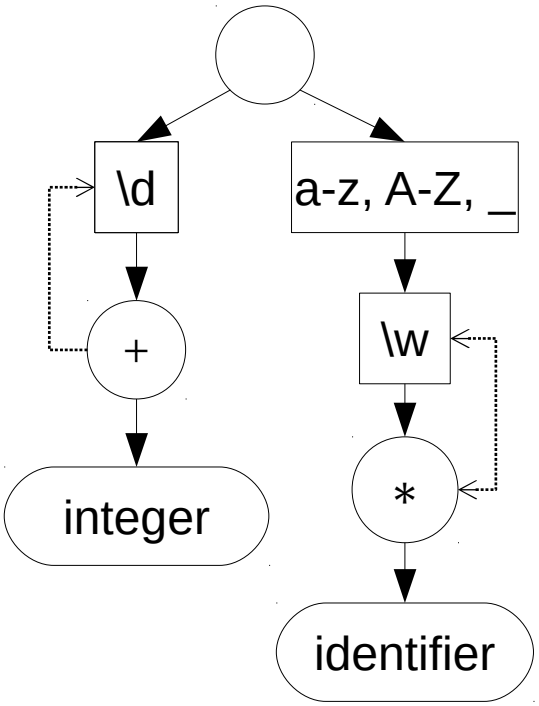

Рисунок 1 — Конечный автомат внутри программы.

На данном этапе программа поддерживает минимально необходимую часть операторов регулярных выражений. Среди них: Замыкания, группировка, альтернация (символ “|”), классы символов (\w – слова, \d – цифры, \s – пробельные символы, [...] – произвольный класс), ленивая квантификация

(выход осуществляется как только появляется такая возможность), некоторые флаги.

Благодаря своей древовидной структуре программа способна определять пересечения двух и более правил, то есть ситуацию когда два разных правила имеют общую цепочку с точностью до стоп символа. При получении такой цепочки состояние будет неопределенно о чем программа немедленно сообщит пользователю. Также программа проверяет корректность правила и уникальность имени.

3. Программная реализация

Программа состоит из одной единственной формы разделенной на два поля. Верхнее содержит в себе код описания лексики. Ниже расположено окно вывода которое появиться при возникновении ошибки.

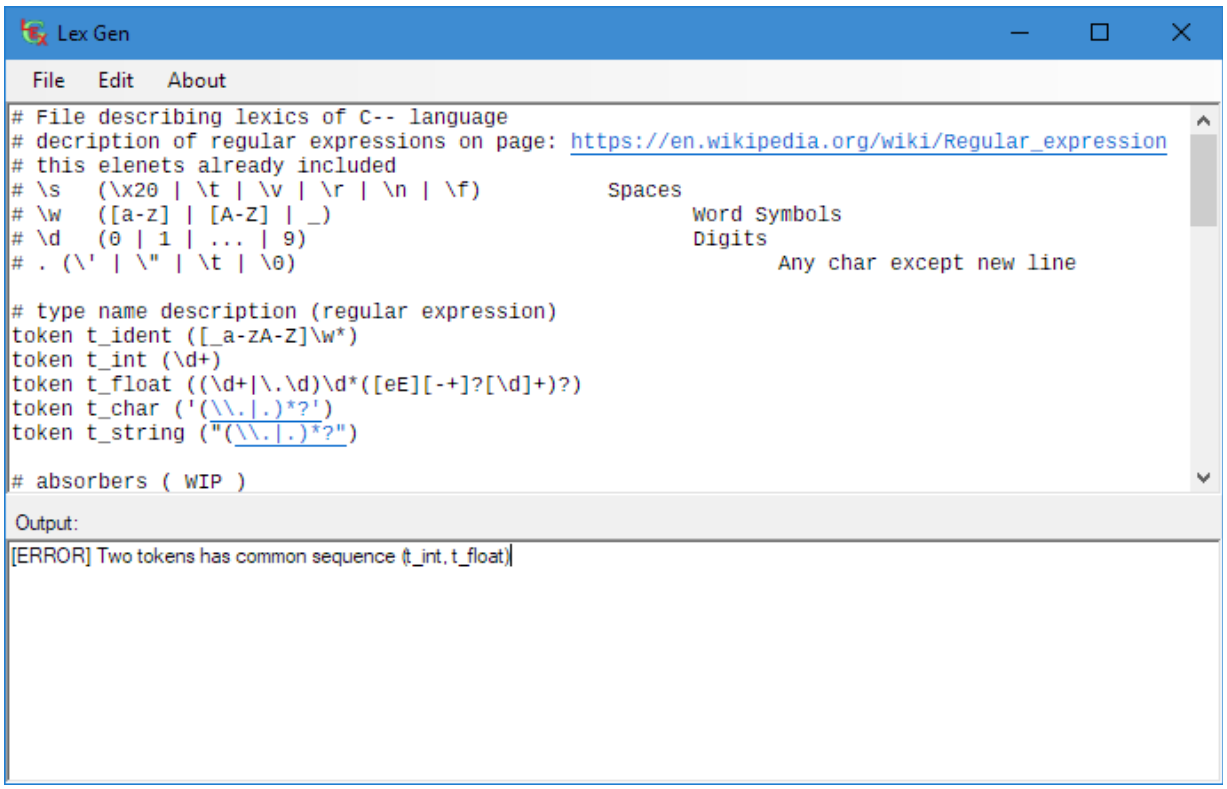

Рисунок 2 — Окно программы.

Главное меню программы содержит пункты:

Файл (File):

Новый (New)— Создать новый набор правил.

Открыть (Open)— Открыть существующий файл.

Сохранить (Save)— Сохранить текущий набор правил.

Сохранить как (Save As)— Сохранить под новым именем.

Выйти (Exit) — закрыть программу.

Правка (Edit):

Отменить (undo)

Повторить (redo)

Вырезать (cur)

Скопировать (copy)

Вставить (paste)

Генерировать (Generate) – Создать анализатор.

Пример кода сгенерированного программой:

```
76  size_t lex(const lchar *str, size_t str_lenght) {
77      states_t states;
78      unsigned depth = 0;
79      unsigned state = 0;
80      unsigned substate = 0;
81      size_t str_begin = 0;
82      size_t i;
83      char ch;
84      states.capacity = 0;
85      states.length = 0;
86      states.state = nullptr;
87      states.substate = nullptr;
88      for (size_t j = 0; j <= str_lenght; j++) {
89          if (j == str_lenght)
90              if (state == 0)
91                  break;
92              else
93                  substate = -1;
94          ch = str[j];
95          i = j;
96      send_to_top:
97          switch (state) {
98          case 0: {
99              switch (substate) {
100             case 0:if ( ch == '=' ) { state = 1; push_state(&states, 0, 0); break; }
101             case 1:if ( ch == '*' ) { state = 2; push_state(&states, 0, 1); break; }
102             case 2:if ( ch == ',' ) { state = 3; push_state(&states, 0, 2); break; }
103             case 3:if ( ch == '/' ) { state = 4; push_state(&states, 0, 3); break; }
104             case 4:if ( ch == '>' ) { state = 6; push_state(&states, 0, 4); break; }
105             case 5:if ( ch == '(' ) { state = 8; push_state(&states, 0, 5); break; }
106             case 6:if ( ch == '<' ) { state = 9; push_state(&states, 0, 6); break; }
107             case 7:if ( ch == '[' ) { state = 11; push_state(&states, 0, 7); break; }
108             case 8:if ( ch == '&' ) { state = 12; push_state(&states, 0, 8); break; }
109             case 9:if ( ch == '|' ) { state = 14; push_state(&states, 0, 9); break; }
```

Рисунок 3 — пример кода.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта была разработана программа-генератор лексических анализаторов позволяющая генерировать довольно сложные и длинные программы анализаторы на основе набора регулярных выражений. Реализация производилась с помощью объектно-ориентированного программирования на языке C#.

Выполнение поставленных задач позволило реализовать программу генерирующую лексические анализаторы которые можно встраивать непосредственно в код. При этом анализаторы являются однопроходными, а значит быстрыми. Тестирование полученного программного продукта ошибок не выявило.

Список использованных источников

1. Шилдт Г. Полный справочник по C# 4.0 / Г. Шилдт. – Лондон: «Osborne», 2010. –112 с.
2. Рихтер Д. Программирование на языке C#/ Д. Рихтер – СПБ: «Пи-тер», 2008. –372 с.
3. Нортроп Т. Основы разработки приложений на платформе Microsoft.NET Framework. Учебный курс Microsoft. Перевод с англ./ Т. Нортроп, Ш. Уилдермьюс, Б. Райан. – Москва: «Русская редакция», 2007. – 864 с.
4. Шилдт Г. C#, учебный курс. / Г. Шилдт. – СПб.: «Питер», 2003. – 512 с.
5. Фридл, Дж. Регулярные выражения = Mastering Regular Expressions. — СПб.: «Питер», 2001. — 352 с. — (Библиотека программиста).
6. Смит, Билл. Методы и алгоритмы вычислений на строках (regexp) = Computing Patterns in Strings. — М.: «Вильямс», 2006. — 496 с.

Приложение А

Исходный код проекта находится по адресу

https://github.com/gHainar/LexGen