Министерство образования и науки РФ
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный технологический институт
(технический университет)»

Факультет информационных технологий и управления
Кафедра систем автоматизированного проектирования и управления

Направление подготовки: 230100 – «Информатика и вычислительная техника»
Уровень подготовки: Специалист 230102 – «Автоматизированные системы обработки информации и управления»

Курсовая работа по дисциплине:
«Лингвистическое и программное обеспечение автоматизированных систем»

Пояснительная записка к курсовой работе

Тема работы: **Таблично-управляемый синтаксический анализатор входных текстов**

Руководитель: _____ Простителенко О.В.
(подпись дата)

Выполнил:
студент гр. 4884 _____ Ануфриев А. О.
(подпись дата)

_____
(оценка)

Санкт-Петербург
2011

Оглавление

Постановка задачи

Целью проектирования является создание работающей программы, выполняющей функции таблично-управляемого синтаксического анализа для произвольных входных данных. Данная программа должна принимать на вход настроечные данные, данные для анализа.

Результатом работы программы является сообщение об синтаксической ошибке в исходных данных, если таковая имеется. А также дерево разбора в виде xml-файла с указанием всей последовательности вывода входной строки из стартового нетерминала.

Исходные данные

Основные

В качестве основных исходных данных используется последовательность текстовых символов, получаемая программой либо из файла (допускает многострочные данные), либо из непосредственного ввода в консольное окно программы (только однострочные последовательности).

Настроечные

Дополнительными исходными данными является текстовый файл с описанием грамматики языка для настройки анализатора. Описание грамматики производится в системе БНФ (Форма Бэкуса-Наура) с учетом того, что заранее определенными терминальными[1] символами являются следующие имена:

- ***id*** – описывает любой идентификатор. Т.е. произвольную последовательность из букв и цифр, начинающуюся с буквы.
- ***num*** – описывает любое десятичное число. Т.е. произвольную последовательность цифр, возможно разделенных одной точкой
- ***empty*** – описывает пустой терминал.

Так же стоит учитывать, что лексический анализатор, считывающий описание грамматики выделяет идентификаторы в отдельную лексему по тому же правилу, что и лексический анализатор входного текста. Таким образом строка в описании грамматики

---

1 *По сути они являются не терминальными, поскольку могут быть разложены на более простые составляющие (буквы и цифры), но с точки зрения описания грамматики это терминалы и обозначаются без обрамления в треугольные скобки.*

<Нетерминал1>id4.8fempty<Нетерминал2>

будет разбита на последовательность из 6 лексем:

Нетерминал1, ***id4***, ***.***, ***8***, ***fempty***, ***Нетерминал2***.

И заранее определенные терминалы (id, empty) не будут найдены в подобной строке. Так что при необходимости допустимо обрамлять терминалы (***id***, ***num*** и ***empty***) в треугольные скобки, как не терминалы. Ошибки в чтении грамматики это не вызовет, если не помещать их в левую часть продукции (до знака «**::=**»).

Важно!

Для корректной работы программы описание грамматики должно обладать следующими свойствами:

- Безлеворекурсивность. Грамматика не должна иметь левых рекурсий. Т.е. не должно быть правил вывода, в правой части которых содержится тот же нетерминал, что и в левой, но при этом не имеет предшествующего терминального символа и не может быть приведен к пустой строке.

  <E> ::= <E> + <T> - неверно

  <E> ::= +<E> | empty - верно

- Однозначность. На основе данной грамматики должно быть возможным построить таблицу анализа, в которой для каждого нетерминала и лексемы входной строки соответствует либо ошибка, либо только ОДНО правило замены. Т.е. для каждой лексемы из входного текста существует лишь единственное правило замены текущего нетерминала на стеке (либо не существует вовсе, что означает ошибку входных данных).

  <E> ::= +<T> | -<T> -неверно

Особые ситуации

В случае отсутствия файлов исходных и настроечных данных программой будет предложено повторно указать пути к ним.

В случае ошибки загрузки описания грамматики из-за ошибок в ее оформлении будет выведено соответствующее сообщение (с ее описанием и указанием строки в исходном файле), потом программа попробует загрузить файл по умолчанию и в случае отсутствия/некорректности этого файла, предложит заного ввести путь к файлу с описанием грамматики.

Теоретический материал

Синтаксический анализ — процесс проверки, может ли последовательность входных токенов (входная строка) быть порождена исходной грамматикой языка (принадлежит генерируемому грамматикой языку). В случае, если входная строка содержит синтаксические ошибки, синтаксический анализатор должен сообщить об этом и по возможности указать на проблемное место и суть проблемы. Так же, будучи в составе транслятора, синтаксический анализатор может формировать дерево разбора и/или синтаксическое дерево, для дальнейшей его обработки семантическим анализатором.

Для своей работы, синтаксический анализатор нуждается в описании грамматики. Существуют несколько типов формальных грамматик согласно иерархии Хомского (неограниченные, контекстно-зависимые, контекстно-свободные и регулярные). В программировании используется контекстно-свободные грамматики (далее КС-грамматики), позволяющие описать грамматическую структуру большинства языков. Поэтому она и использована в программе. Одним из способов описания КС-грамматики является форма Бэкуса-Наура.

КС-грамматики так же делятся на следующие классы:

LR(k). Для разбора используется восходящий синтаксический анализ. Первая «L» говорит о том, что разбор входной строки происходит слева (left) на право. Вторая «R» говорит о получении правого порождения (в каждом предложении выбирается крайний справа нетерминал). Числов *k* говорит о необходимом для принятия решения о дальнейшем поведении анализатора количестве символов на каждом шаге предпросмотра. В виду сложности реализации анализаторов для этого класса грамматик, он не будет рассматриваться в этой работе.

LL(k). Для разбора используется нисходящий синтаксический разбор. Первая «L» говорит о том, что разбор входной строки происходит слева (left) на право. Вторая «L» говорит о получении левого порождения (в каждом предложении выбирается крайний слева нетерминал). Числов *k* говорит о необходимом для принятия решения о дальнейшем поведении анализатора количестве символов на каждом шаге предпросмотра.

В данной работе используется его разновидность, LL(1) класс грамматик. Класс грамматик LL(1) достаточно богат для того, чтобы охватить большинство программных конструкций, хотя при написании грамматики для исходного языка требуется аккуратность. Например, в LL(1)-грамматике не может быть ни левой рекурсии, ни неоднозначности.

Для разбора LL(1)-грамматик применяется либо синтаксический анализ методом рекурсивного спуска, либо нерекурсивный предиктивный синтаксический анализ, использующий таблицу синтаксического анализа. Таблица предиктивного анализа содержит в своих строках нетерминалы грамматики, в столбцах — возможные входные символы(лексемы), а на пересечении — продукция, которой

заменяется нетерминал (соответствующий строке) на стеке при получения из входного потока символа, соответствующего столбцу.

Разработанная программа работает как раз по принципу нерекурсивного предиктивного анализа.

В качестве вспомогательных функций используются функции $FIRST(\alpha)$ и $FOLLOW(A)$. Подробное описание которых, включая алгоритм вычисления, можно найти в книге Ахо, Лам, Сети, Ульман «Компиляторы. Принципы, технологии, инструменты» в разделе «Нисходящий синтаксический анализ».

Вот краткое описание, взятое из книги

Определим $FIRST(\alpha)$, где α — произвольная строка символов грамматики, как множество терминалов, с которых начинаются строки, порождаемые α. Если α за 0 и более шагов порождает пустую строку, то пустую сроку так же вносят в множество $FIRST(\alpha)$.

Определим $FOLLOW(A)$ для не терминала A как множество терминалов a, которые могут располагаться непосредственно справа от A в некоторой сентенциальной форме.

В программе эти функции реализованы в соответствии с описанием в указанной книге и используются при построении таблицы анализа (так же в соответствии с алгоритмом из книги).

Форматы представления данных

В ходе разработки программы были спроектированы несколько пользовательских классов:

Класс CodeLexer

Лексический анализатор для программного кода.

Открытые члены

- **CodeLexer** (QTextStream &strm)

  Создает экземпляр лексического анализатора

- virtual **Token nextToken** ()

Считывает новую лексему из потока, преобразует в токен и возвращает этот токен.

Открытые атрибуты

- QSet< QString > **reservedWords**

  Список зарезервированных слов

Подробное описание

Лексический анализатор для программного кода

Считывает токены из входного потока, объединяя в лексемы целые и дробные числа (токен с псевдонимом *num*) и идентификаторы (токен с псевдонимом *id*). Отдельной лексемой считывает конец потока. Отслеживает позицию считывания во входном потоке.

Конструктор(ы)

- **CodeLexer::CodeLexer (QTextStream & *strm*) `[explicit]`**

Создает экземпляр лексического анализатора

Аргументы:

strm	ссылка на входной поток, из которого будут считываться лексемы.

Методы

- **Token CodeLexer::nextToken () `[virtual]`**

Считывает новую лексему из потока преобразует в токен и возвращает этот токен.

Анализирует символы входного потока.

Если поток содержит последовательность цифр (возможно разделенных точкой), то формируется токен числа с псевдонимом `"num"` , в поле *value* которого значение числа.

Если поток содержит последовательность букв и цифр, начинающуюся с буквы, то формируется токен идентификатора. Далее проверяется наличие имени идентификатора (лексемы) в списке зарезервированных слов, если оно там имеется, то псевдонимом токена является сама лексема, иначе `"id"` , хранящий в поле *value* имя идентификатора (лексему).

Возвращает: сформированный токен.

Замещает **Lexer**.

Объявления и описания членов классов находятся в файлах:

- CodeLexer.h
- CodeLexer.cpp

Класс GrammarLexer

Лексический анализатор для работы с описанием КС-грамматики.

Открытые члены

- **GrammarLexer** (QTextStream &strm)

  Создает лексический анализатор для указанного потока strm.

- virtual **Token nextToken** ()

Считывает новую лексему из потока преобразует в токен и возвращает этот токен.

Статические открытые данные

- static const QString **deffLexem** = "::="

  лексема, соединяющая левую и правую часть правила грамматики.

- static const **SurroundChars surroundChars** = **SurroundChars**('<', '>')

  символы, в которые заключается имя неТерминала.

Подробное описание

Лексический анализатор для работы с описанием КС-грамматики.

Обрабатывает файлы и прочие текстовые потоки построчно. Выделяя в строке отдельные лексемы левую (с одним нетерминалом) и правую (с одним и более символов грамматики) часть правила, символ их разделяющий. Правая часть разбивается на лексемы, соответствующие терминальным, нетерминальным символам и символу "|", разделяющему несколько продукций для единой левой части.

Методы

- **Token GrammarLexer::nextToken () `[virtual]`**

Считывает новую лексему из потока преобразует в токен и возвращает этот токен.

Анализирует символы входного потока построчно. В каждой строке пытается найти последовательность **NsM** , где **N** - нетерминальный символ, окруженный символами из surroundChars (угловыми скобками в системе БНФ); **s** - символ определения порождения (*"::="* в системе БНФ) **M** - множество терминальных, нетерминальных символов и символа "|".

Если строка входного потока не начинается с пары **Ns** , то возвращает токен ошибки (*alias* = `"error"` , *isSystemToken* = `true` .

Каждая строка преобразуется в очередь токенов, откуда и происходит считывание очередного токена до тех пор, пока очередь не опустеет, после чего считывается новая строка и формируется новая очередь.

Возвращает: сформированный токен.

Объявления и описания членов классов находятся в файлах:

- GrammarLexer.h
- GrammarLexer.cpp

Класс GrammarSymbol

Открытые члены

- **GrammarSymbol** ()

  Создает грамматический пустой символ.

- **GrammarSymbol** (const QString &**alias**, const QString &**description**=QString())

  Создает грамматический символ с указаннами псевдонимом и описанием.

- bool **isTerminal** () const

  Проверяет, является ли символ терминальным.

Открытые атрибуты

- QString **alias**

  псевдоним грамматического символа.

- QString **description**

  описание грамматического символа.

- QList< SymbolsSequence > **products**

  продукции грамматического символа.

Подробное описание

Описывает символ грамматики.

Конструктор(ы)

- **GrammarSymbol::GrammarSymbol (const QString &** ***alias*****, const QString &** ***description*** **=** `QString()) [explicit]`

  Создает грамматический символ с указаннами псевдонимом и описанием.

  Аргументы:

alias	псевдоним грамматического символа.
description	описание грамматического символа

Методы

- **bool GrammarSymbol::isTerminal () const**

  Проверяет, является ли символ терминальным.

  Возвращает: `true` , если символ терминальный (не содержит продукций), иначе `false`.

Объявления и описания членов классов находятся в файлах:

- GrammarSymbol.h
- GrammarSymbol.cpp

Класс Lexer

Лексический анализатор.

Открытые члены

- **Lexer** (QTextStream &strm)

  Создает лексический анализатор для обработки потока данных strm.

- **Token token** () const

  Возвращает значение токена на основе последней считанной лексемы.

- virtual **Token nextToken** () = 0

- const QTextStream & **stream** ()

Возвращает ссылку на поток, с которым работает текущий экземпляр анализатора.

Открытые атрибуты

- QSet< QChar > **ignoredChars**

  Список игнорируемых символов.

Защищенные данные

- QTextStream & **m_strm**

  Поток для обработки.

- **Token m_token**

  Текущий, уже считанный токен.

Подробное описание

Лексический анализатор.

Конструктор(ы)

- **Lexer::Lexer (QTextStream & *strm*) `[explicit]`**

  Создает лексический анализатор для обработки потока данных *strm*.

  Аргументы:

strm	анализируемый поток
whiteSpaces	список игнорируемых символов

Методы

- **const QTextStream & Lexer::stream ()**

Возвращает ссылку на поток, с которым работает текущий экземпляр анализатора.

- **Token Lexer::token () const**

  Возвращает значение токена на основе последней считанной лексемы.

Объявления и описания членов классов находятся в файлах:

- Lexer.h
- Lexer.cpp

Структура SurroundChars

Описание символов, окружающих лексему.

Открытые члены

- **SurroundChars** (char l, char r)

Открытые атрибуты

- char **left**

  левый обрамляющий символ

- char **right**

  правый обрамляющий символ

Подробное описание

Описание символов, окружающих лексему.

Объявления и описания членов структур находятся в файлах:

- GrammarLexer.h
- GrammarLexer.cpp

Класс TableDrivenParser

Синтаксический нерекурсивный предиктивный анализатор LL(1)-грамматик.

Открытые члены

- **TableDrivenParser** (QTextStream *errorOut=0)

  Создает экземпляр синтаксического анализатора.

- bool **loadGrammar** (const QString &filePathName)

Загружает описание LL(1) грамматики из файла для конфигурирования парсера.

- void **setGrammar** (const QList< **GrammarSymbol** > &symbols)

  Создает внутреннее описание грамматики на основе переданого списка

- bool **analyse** (**Lexer** &lexer, QDomDocument &result, bool

isParseTreeInResult=true)

Анализирует входной поток stream на соответствие грамматике.

Подробное описание

Синтаксический нерекурсивный предиктивный анализатор LL(1)-грамматик.

По указанному в виде файла описанию LL(1)-грамматики производит синтаксический разбор входного потока с использованием указанного лексического анализатора.

Формирует дерево разбора в виде XML-документа.

Конструктор(ы)

- **TableDrivenParser::TableDrivenParser (QTextStream * *errorOut* = `0`)**

  Создает экземпляр синтаксического анализатора.

  Аргументы:

errorOut	указатель на поток для вывода сообщений об ошибках анализа.

Методы

- **bool TableDrivenParser::analyse (Lexer & *lexer*, QDomDocument & *result*, bool *isParseTreeInResult* = `true`)**

  Анализирует входной поток *stream* на соответствие грамматике.

  Производит анализ входного потока *stream* на соответствие грамматике, установленной для парсера с помощью **loadGrammar()** или **setGrammar()**. Все ошибки, возникающие во время анализа выводятся либо в поток, заданный при создании объекта парсера, либо в stderr, если поток не был задан.

  В случае успешного анализа функция возвращает `true` и заполняет переданную по ссылке конструкцию QDomDocument деревом разбора (при *isParseTreeInResult* = `true` , по умолчанию) или синтаксическим деревом (при *isParseTreeInResult* = `false`).

  Аргументы:

lexer	лексический анализатор, из которого будут получаться лексемы.
result	дерево разбора или синтаксическое дерево, в зависимости от *isSyntaxTreeInResult*
isParseTreeInResult	флаг, указывающий, что нужно строить дерево разбора, а не синтаксическое дерево

- **bool TableDrivenParser::loadGrammar (const QString & *filePathName*)**

Загружает описание LL(1) грамматики из файла для конфигурирования парсера.

Файл должен содержать описание по системе БНФ. При описании разрешено использовать как латиницу, так и кириллицу. Для грамматики предопределены такие терминалы, как `"id"` для идентификаторов и `"num"` для вещественных и целых чисел, а так же `"empty"` для пустого символа.

Аргументы:

filePathName	путь к файлу с описанием грамматики.

Возвращает: `true` , если грамматика успешно загружена, иначе false.

- **void TableDrivenParser::setGrammar (const QList< GrammarSymbol > & *symbols*)**

Создает внутреннее описание грамматики на основе переданного списка.

Аргументы:

symbols	список символов грамматики и их продукций.

Объявления и описания членов классов находятся в файлах:

- TableDrivenParser.h
- TableDrivenParser.cpp

Структура Token

Токен полученный из лексемы входного потока.

Открытые члены

- **Token** (QString a=QString("empty"), QVariant v=QVariant(), int l=-1, int p=-1, bool isSys=false)

  Создает токен с указанными параметрами

Открытые атрибуты

- QString **alias -** псевдоним для лексемы
- QVariant **value -** значение лексемы
- int **line -** номер строки входного потока, содержащей лексему
- int **posInLine -** позиция лексемы в строке входного потока
- bool **isSystemToken -** флаг системного токена (ошибки, конец потока)

Подробное описание

Токен полученный из лексемы входного потока.

Конструктор(ы)

- **Token::Token (QString *a* = `QString("empty")`, QVariant *v* = `QVariant()`, int *l* = `-1`, int *p* = `-1`, bool *isSys* = `false`)**

  Создает токен с указанными параметрами.

  Аргументы:

a	псевдоним токена.
v	значение токена.
l	номер строки в исходном потоке, содержащей токен.
p	позиция токена в строке *l* исходного потока.
isSys	флаг того, является ли токен системным.

Объявления и описания членов структур находятся в файлах:

- Lexer.h
- Lexer.cpp

Структура входных и выходных данных

Файл описания грамматики

Должен состоять из набора строк по системе БНФ, начинающихся с имени стартового нетерминального символа (нетерминала), после которого следует знак определения «::=» и потом набор продукций, разделенных между собой вертикальной чертой «|». Сами продукции представляют собой произвольные последовательности Нетерминальных и терминальных символов грамматики. Все нетереминальные символы, встречающиеся в правой части строк должны хотя бы 1 раз встречаться в левой части строк (т.е. иметь правило вывода). Пример содержания подобного файла приведено в разделе «Работа программы» этого отчета.

Анализируемая последовательность

В качестве исходных данных можно использовать как многострочный текстовый файл, так и однострочные текстовые данные, введенные непосредственно в консольное окно работающей программы после соответствующего запроса. Единственное ограничение на эти данные — они должны быть текстовые.

Файл с деревом разбора

Результатом работы анализатора, помимо сообщений о наличие/отсутствии ошибок во входных данных, является дерево разбора, которое генерируется в том случае, если не было ошибок анализа.

Для отображения древовидных (иерархических) структур лучше всего подходит расширяемый язык разметки, известный как XML. Выходной файл имеет тип «ParsingResult», в качестве корневого элемента выступает узел <ParseTree xmlns =“”> указывающий на семантику структуры (xmlns – пространство имен по умолчанию, программа оставляет его пустым). Следующий, дочерний для ParseTree узел состоит всегда из одного узла, стартового нетерминала, который и разворачивается в дерево, на листьях которого получаются терминалы входной строки. Пример содержания этого файла так же можно наблюдать в разделе отчета «Работа программы».

Работа программы

Для тестирования работы программы была сформированная корректная LL(1)-грамматика, порождающая язык, состоящий из простых математических операций. Данная грамматика была получена преобразованием более наглядной, но не отвечающей требованиям LL(1) класса грамматик:

<Выражение> ::= <Произведение> | <Выражение> <ПлюсМинус> <Произведение>

<Произведение> ::= <Операнд> | <Произведение> <УмножениеДеление> <Операнд>

<Операнд> ::= (<Выражение>) | id | num

<ПлюсМинус> ::= + | -
<УмножениеДеление> ::= * | /

После избавления от неоднозначности и левой рекурсии получилась аналогичная, но видоизмененная грамматика, занесенная в файл grammar.txt:

<Выражение> ::= <Произведение><КонецВыражения>
<КонецВыражения> ::= <ПлюсМинус> <Произведение><КонецВыражения> | empty
<Произведение> ::= <Операнд><КонецПроизведения>
<КонецПроизведения> ::= <УмножениеДеление> <Операнд><КонецПроизведения> | empty
<Операнд> ::= (<Выражение>) | id | num
<ПлюсМинус> ::= + | -
<УмножениеДеление> ::= * | /

В качестве входных данных использовались 2 строки с корректными данными:

- 12 + 34
- var1 + 4 – 8 * (var2 + 4) / var3

В обоих случая синтаксический анализатор сообщил об отсутствии ошибок и предложил сохранить дерево разбора в файл (рисунок 1).

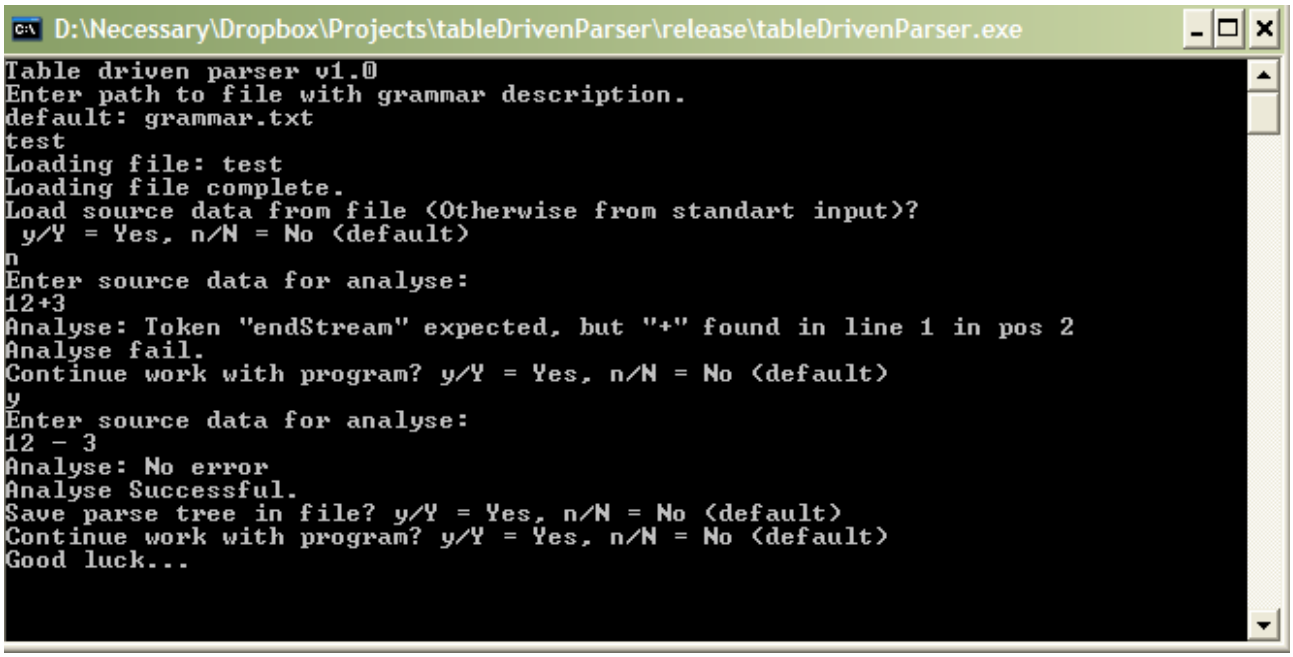

Рисунок 1 — Окно работы программы

Содержание файлов с результатом следующие:

Out1.xml

```
<ParseTree xmlns="">
  <Выражение>
    <Произведение>
      <Операнд>
        <Terminal lineInSrc="1" alias="num" posInLine="0">12</Terminal>
      </Операнд>
      <КонецПроизведения>
        <Terminal lineInSrc="1" alias="empty" posInLine="3">empty</Terminal>
      </КонецПроизведения>
    </Произведение>
    <КонецВыражения>
      <ПлюсМинус>
        <Terminal lineInSrc="1" alias="+" posInLine="3">+</Terminal>
      </ПлюсМинус>
      <Произведение>
        <Операнд>
          <Terminal lineInSrc="1" alias="num" posInLine="5">34</Terminal>
```

```
            </Операнд>
            <КонецПроизведения>
               <Terminal lineInSrc="1" alias="empty" posInLine="7">empty</Terminal>
            </КонецПроизведения>
         </Произведение>
         <КонецВыражения>
            <Terminal lineInSrc="1" alias="empty" posInLine="7">empty</Terminal>
         </КонецВыражения>
      </КонецВыражения>
   </Выражение>
</ParseTree>
```

Out2.xml

```
<ParseTree xmlns="">
   <Выражение>
      <Произведение>
         <Операнд>
            <Terminal lineInSrc="1" alias="id" posInLine="0">var1</Terminal>
         </Операнд>
         <КонецПроизведения>
            <Terminal lineInSrc="1" alias="empty" posInLine="5">empty</Terminal>
         </КонецПроизведения>
      </Произведение>
      <КонецВыражения>
         <ПлюсМинус>
            <Terminal lineInSrc="1" alias="+" posInLine="5">+</Terminal>
         </ПлюсМинус>
         <Произведение>
            <Операнд>
               <Terminal lineInSrc="1" alias="num" posInLine="7">4</Terminal>
            </Операнд>
            <КонецПроизведения>
               <Terminal lineInSrc="1" alias="empty" posInLine="9">empty</Terminal>
            </КонецПроизведения>
         </Произведение>
         <КонецВыражения>
            <ПлюсМинус>
               <Terminal lineInSrc="1" alias="-" posInLine="9">-</Terminal>
            </ПлюсМинус>
            <Произведение>
               <Операнд>
                  <Terminal lineInSrc="1" alias="num" posInLine="11">8</Terminal>
               </Операнд>
               <КонецПроизведения>
                  <УмножениеДеление>
                     <Terminal lineInSrc="1" alias="*" posInLine="13">*</Terminal>
                  </УмножениеДеление>
                  <Операнд>
                     <Terminal lineInSrc="1" alias="(" posInLine="15">(</Terminal>
```

```
                <Выражение>
                  <Произведение>
                    <Операнд>
                      <Terminal lineInSrc="1" alias="id" posInLine="16">var2</Terminal>
                    </Операнд>
                    <КонецПроизведения>
                      <Terminal lineInSrc="1" alias="empty" posInLine="21">empty</Terminal>
                    </КонецПроизведения>
                  </Произведение>
                  <КонецВыражения>
                    <ПлюсМинус>
                      <Terminal lineInSrc="1" alias="+" posInLine="21">+</Terminal>
                    </ПлюсМинус>
                    <Произведение>
                      <Операнд>
                        <Terminal lineInSrc="1" alias="num" posInLine="23">4</Terminal>
                      </Операнд>
                      <КонецПроизведения>
                        <Terminal lineInSrc="1" alias="empty" posInLine="24">empty</Terminal>
                      </КонецПроизведения>
                    </Произведение>
                    <КонецВыражения>
                      <Terminal lineInSrc="1" alias="empty" posInLine="24">empty</Terminal>
                    </КонецВыражения>
                  </КонецВыражения>
                </Выражение>
                <Terminal lineInSrc="1" alias=")" posInLine="24">)</Terminal>
              </Операнд>
              <КонецПроизведения>
                <УмножениеДеление>
                  <Terminal lineInSrc="1" alias="/" posInLine="25">/</Terminal>
                </УмножениеДеление>
                <Операнд>
                  <Terminal lineInSrc="1" alias="id" posInLine="26">var3</Terminal>
                </Операнд>
                <КонецПроизведения>
                  <Terminal lineInSrc="1" alias="empty" posInLine="30">empty</Terminal>
                </КонецПроизведения>
              </КонецПроизведения>
            </КонецПроизведения>
          </Произведение>
          <КонецВыражения>
            <Terminal lineInSrc="1" alias="empty" posInLine="30">empty</Terminal>
          </КонецВыражения>
        </КонецВыражения>
      </КонецВыражения>
    </Выражение>
</ParseTree>
```

Так же, для демонстрации обработки некорректных входных строк были введены следующие данные:

12 + + 23 — 5

43 * (455 — 6

2) — 4

Результаты работы программы отображены на рисунке 2.

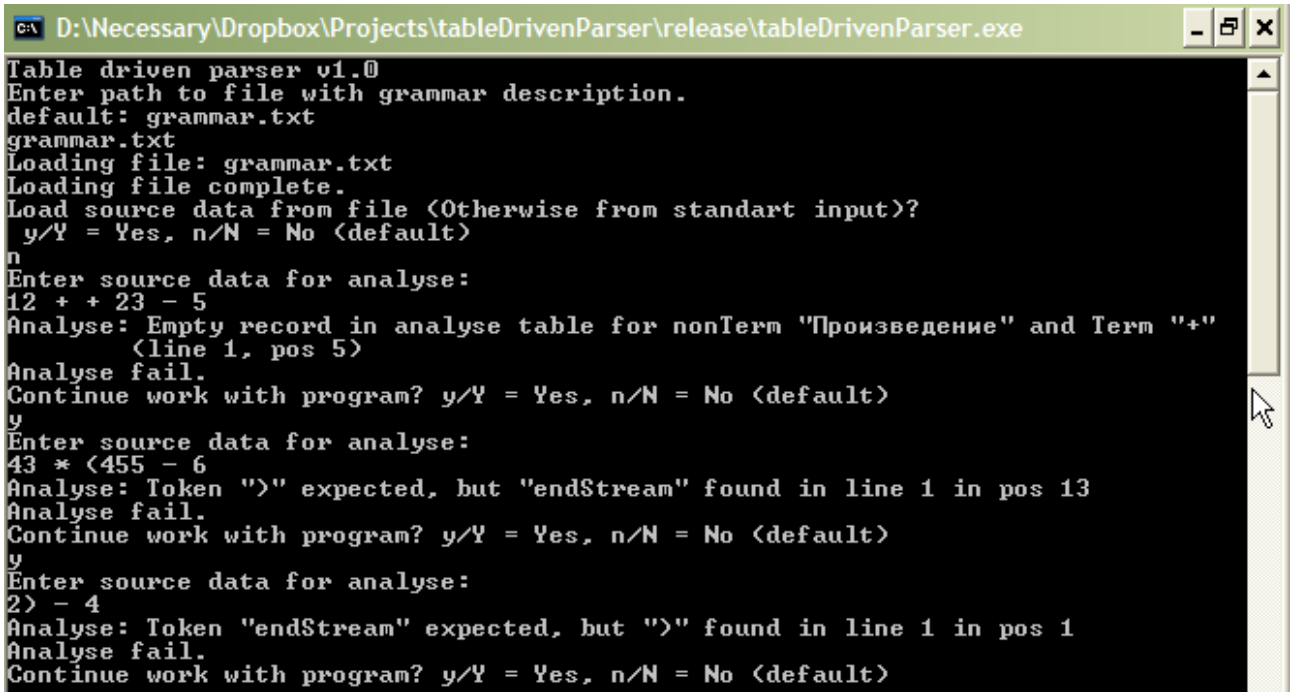

Рисунок 2 — Окно работы программы при обработке ошибочных данных

Исходный текст программы

Lexer.h

```
#ifndef LEXER_H
#define LEXER_H
#include <QString>
#include <QVariant>
#include <QList>
#include <QSet>
#include <QTextStream>
//! Токен полученный из лексемы входного потока
struct Token
{
    QString alias;      //!< псевдоним для лексемы
    QVariant value;     //!< значение лексемы
    int line;           //!< номер строки входного потока, содержащей лексему
    int posInLine;      //!< позиция лексемы в строке входного потока
    bool isSystemToken; //!< флаг системного токена (ошибки, конец потока)
    Token(QString a = QString("empty"),
          QVariant v = QVariant(),
          int l = -1, int p = -1,
          bool isSys = false);
};
//! Абстрактный лексический анализатор
/*!
  Описывает минимальный набор функционала для лексического анализатора
 */
class Lexer
{
  public: //members
    QSet<QChar> ignoredChars; //!< Список игнорируемых символов
  public: //methods
    explicit Lexer(QTextStream &strm);
    Token token() const;
    //! Возвращает следующую лексему в виде токена.
    virtual Token nextToken() = 0;
    const QTextStream& stream();
  protected: //members
    QTextStream & m_strm;   //!< Поток для обработки
    Token m_token;          //!< Текущий, уже считанный токен
  private: //methods
    //Запрет на использование пустого конструктора
    Lexer();
};
#endif // LEXER_H
```

Lexer.cpp

```
/*
  File   : Lexer.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
         автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Абстрактный клас лексического анализатора.
*/
#include "Lexer.h"
//! Создает токен с указанными параметрами
/*!
    \param a псевдоним токена.
    \param v значение токена.
    \param l номер строки в исходном потоке, содержащей токен.
    \param p позиция токена в строке \a l исходного потока.
    \param isSys флаг того, является ли токен системным.
  */
Token::Token(QString a, QVariant v, int l, int p, bool isSys)
    : alias(a), value(v), line(l), posInLine(p), isSystemToken(isSys)
{
    //
}
//! Создает лексический анализатор для обработки потока данных \a strm
/*!
  \param strm анализируемый поток
  */
Lexer::Lexer(QTextStream &strm):
    m_strm(strm)
{
    ignoredChars.insert(' ');
    ignoredChars.insert('\t');
    ignoredChars.insert('\r');
    ignoredChars.insert('\n');
    m_token = Token("start", 0, 0, 0, true);
    //считывание первой лексемы
    //nextToken();
}
//! Возвращает значение токена на основе последней считанной лексемы
/*!
  \return последний считанный токен.
  */
Token Lexer::token() const{
    return m_token;
}
//! Возвращает ссылку на поток, с которым работает текущий экземпляр анализатора
```

```
/*!
  \return ссылка на обрабатываемый поток.
  */
const QTextStream &Lexer::stream(){
    return m_strm;
}
```

GramarSymbol.h

```
#ifndef GRAMMARSYMBOL_H
#define GRAMMARSYMBOL_H
#include <QList>
#include <QString>
class GrammarSymbol;
typedef QList<GrammarSymbol*> SymbolsSequence;
//! Описывает символ грамматики
class GrammarSymbol
{
    public: //members
        QString alias;              //!< псевдоним грамматического символа
        QString description;        //!< описание грамматического символа
        QList<SymbolsSequence> products;    //!< продукции грамматического символа
    public: //methids
        GrammarSymbol();
        explicit GrammarSymbol(const QString &alias,
                               const QString &description  = QString());
        bool isTerminal() const;
};
#endif // GRAMMARSYMBOL_H
```

GramarSymbol.cpp

```
/*
  File   : GrammarSymbol.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
           автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Символ грамматики.
*/
#include "GrammarSymbol.h"
//! Создает грамматический пустой символ
GrammarSymbol::GrammarSymbol(){
    alias = "empty";
    description = QString();
}
//! Создает грамматический символ с указаннами псевдонимом и описанием
/*!
  \param alias псевдоним грамматического символа.
```

```
  \param description описание грамматического символа
  */
GrammarSymbol::GrammarSymbol(const QString &alias, const QString &description){
    this->alias = alias;
    this->description = description;
}
//! Проверяет, является ли символ терминальным
/*!
  \return \c true, если символ терминальный (не содержит продукций), иначе \c false
  */
bool GrammarSymbol::isTerminal() const{
    return this->products.isEmpty();
}
```

GrammarLexer.h

```
#ifndef GRAMMARLEXER_H
#define GRAMMARLEXER_H
#include <Lexer.h>
#include <QQueue>
//! Описание символов, окружающих лексему
struct SurroundChars{
        char left;      //!< левый обрамляющий символ
        char right;     //!< правый обрамляющий символ
        SurroundChars(char l, char r);
};
//! Лексический анализатор для работы с описанием КС-грамматики
/*!
    Обрабатывает файлы и прочие текстовые потоки построчно. Выделяя в строке
    отдельные лексемы левую (с одним нетерминалом) и правую (с одним и более
    символов грамматики) часть правила, символ их разделяющий. Правая часть
    разбивается на лексемы, соответствующие терминальным, нетерминальным символам
    и символу "|", разделяющему несколько продукций для единой левой части.
  */
class GrammarLexer : public Lexer
{
    public: //members
        //! лексема, соединяющая левую и правую часть правила грамматики
        static const QString deffLexem;
        //! символы, в которые заключается имя неТерминала
        static const SurroundChars surroundChars;
    public: //methods
        GrammarLexer(QTextStream &strm);
        virtual Token nextToken();
    private: //members
        int m_currentLine;      //!< хранит номер обрабатываемой строки
        QQueue<Token> m_tokenBuffers; //!< очередь считанный токенов
    private: //methods
        GrammarLexer();
```

```
};
#endif // GRAMMARLEXER_H
```

GrammarLexer.cpp

```
/*
  File   : GrammarLexer.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
         автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Лексического анализатор для файла с описанием КС-грамматики
*/
#include "GrammarLexer.h"
#include <QRegExp>
#include <QString>
const QString GrammarLexer::deffLexem = "::=";
const SurroundChars GrammarLexer::surroundChars = SurroundChars('<', '>');
//! Создает лексический анализатор для указанного потока \a strm
GrammarLexer::GrammarLexer(QTextStream &strm)
    : Lexer(strm)
{
    ignoredChars.remove('\n');
    ignoredChars.remove('\r');
    m_currentLine = 0;
    m_token = Token("startStream", 0, 0, 0, true);
}
//! Считывает новую лексему из потока преобразует в токен и возвращает этот токен
/*!
    Анализирует символы входного потока построчно.
    В каждой строке пытается найти последовательность \b NsM, где
    \b N - нетерминальный символ, окруженный символами из surroundChars (угловыми
    скобками в системе БНФ);
    \b s - символ определения порождения (\a "::=" в системе БНФ)
    \b M - множество терминальных, нетерминальных символов и символа "|".
    Если строка входного потока не начинается с пары \b Ns, то возвращает токен
    ошибки (\a alias = \c "error", \a isSystemToken = \c true.
    Каждая строка преобразуется в очередь токенов, откуда и происходит считывание
    очередного токена до тех пор, пока очередь не опустеет, после чего считывается
    новая строка и формируется новая очередь.
    \return сформированный токен.
  */
Token GrammarLexer::nextToken(){
    if(m_tokenBuffers.isEmpty()){
        if(m_strm.atEnd()){
            m_token = Token("endStream", 0, m_currentLine + 1, -1, true);
            return token();
```

```
}
QString line = m_strm.readLine();
++m_currentLine;
//Удаление из строки всех символов из списка игнорируемых
foreach(QChar c, ignoredChars){
   QRegExp forDel(QString::fromUtf8("(%1*)").arg(c));
   line.remove(forDel);
}
QRegExp finder;
int delPartLen = 0;  // длина удаленной части строки
// Регулярное выражение для символа определения (в БНФ это '::=')
const QString deffSymbRegExp(QString("(%1)").arg(deffLexem));
// Регулярное выражение для нетерминала (в БНФ это <Имя>)
const QString nonTermRegExp = QString::fromUtf8("%1([a-zA-Za-яA-Я]\\w*)%2")
                        .arg(surroundChars.left)
                        .arg(surroundChars.right);
const QString identificatorRegExp = QString::fromUtf8("([a-zA-Za-яA-Я]\\w*)");
const QString otherSymbols = QString::fromUtf8("([^a-zA-Za-яA-Я])");
// Шаблон поиска левой части порождения и знака определения
finder.setPattern(nonTermRegExp + deffSymbRegExp);
finder.indexIn(line);
// Если одна из частей не найдена, то вернуть токен ошибки
if(finder.cap(1).isEmpty() ||  finder.cap(2).isEmpty()){
   m_token = Token("error", 0, m_currentLine, delPartLen, true);
   return token();
}
// Поместить в очередь считанные токены (нетерминал и символ определения)
m_tokenBuffers.enqueue(Token(finder.cap(1), 0,
                        m_currentLine,
                        delPartLen + finder.pos(1)));
m_tokenBuffers.enqueue(Token(finder.cap(2), 0,
                        m_currentLine,
                        delPartLen + finder.pos(2)));
// Удалить найденную подстроку
line.remove(finder);
delPartLen += finder.matchedLength();
finder.setPattern(QString("%1|%2|%3")
               .arg(nonTermRegExp)
               .arg(otherSymbols)
               .arg(identificatorRegExp));
int lexemPos = 0;
// Разбиение строки на состовляющие лексемы
while((lexemPos = finder.indexIn(line)) != -1){
   int capIndex = 0;
   QString lexem;
   //перебор найденных подстрок в поисках ненулевой
   do{
      ++capIndex;
      lexem = finder.cap(capIndex);
```

```
        }while (lexem.isEmpty());
        line.remove(0, finder.matchedLength());
        m_tokenBuffers.enqueue(Token(lexem, 0, m_currentLine, delPartLen));
        delPartLen += finder.matchedLength();
      }
      // Добавление токена конца строки в конец очереди
      m_tokenBuffers.enqueue(Token("endLine", 0, m_currentLine, delPartLen, true));
   }
   m_token = m_tokenBuffers.dequeue();
   return token();
}
SurroundChars::SurroundChars(char l, char r)
   :left(l), right(r)
{
   //nothing;
}
```

CodeLexer.h

```
#ifndef CODELEXER_H
#define CODELEXER_H
#include <Lexer.h>
#include <QStack>
//! Лексический анализатор для программного кода
/*!
   Считывает токены из входного потока, объединяя в лексемы целые и дробные
   числа (токен с псевдонимом \a num) и идентификаторы (токен с псевдонимом
   \a id). Отдельной лексемой считывает конец потока. Отслеживает позицию
   считывания во входном потоке.
 */
class CodeLexer : public Lexer
{
   public: //members
      QSet<QString> reservedWords;    //!< Список зарезервированных слов
   public: //methods
      explicit CodeLexer(QTextStream &strm);
      virtual Token nextToken();
   private: //members
      int m_currentLine;            //!< номер обрабатываемой строки
      int m_pos;                    //!< позиция токена в строке
      QStack<QChar> m_buffer;       //!< хранение считанных символов
};
#endif // CODELEXER_H
```

CodeLexer.cpp

```
/*
  File   : CodeLexer.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
```

```
        автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Лексического анализатор для программного кода
*/
#include "CodeLexer.h"
#ifndef CHECK_LEXER_READ_STATUS
#define CHECK_LEXER_READ_STATUS(stream, retValue) \
   if(stream.status() != QTextStream::Ok){\
      m_token = retValue;\
      return retValue;\
   }
#endif
#ifndef CHECK_FOR_END_STREAM
#define CHECK_FOR_END_STREAM(stream, retValue) \
   if (stream.atEnd()){\
      m_token = retValue;\
      return retValue; \
   }
#endif
//! Создает экземпляр лексического анализатора
/*!
   \param strm ссылка на входной поток, из которого будут считываться лексемы.
 */
CodeLexer::CodeLexer(QTextStream &strm)
   : Lexer(strm), m_currentLine(1), m_pos(0)
{
   m_token = Token("startStream", 0, 0, 0, true);
}
//! Считывает новую лексему из потока, преобразует ее в токен и возвращает этот токен
/*!
   Анализирует символы входного потока.
   Если поток содержит последовательность цифр (возможно разделенных точкой),
   то формируется токен числа с псевдонимом \c "num", в поле \a value которого
   значение числа.
   Если поток содержит последовательность букв и цифр, начинающуюся с буквы, то
   формируется токен идентификатора. Далее проверяется наличие имени
   идентификатора (лексемы) в списке зарезервированных слов, если оно там
   имеется, то псевдонимом токена является сама лексема, иначе \c "id",
   хранящий в поле \a value имя идентификатора (лексему).
   \return сформированный токен.
 */
Token CodeLexer::nextToken(){
   Token errToken("error", "error", m_currentLine, m_pos, true);
   Token endStreamToken("endStream", "endStream", m_currentLine, m_pos, true);
   QChar c;
   if(m_buffer.isEmpty()){
      CHECK_FOR_END_STREAM(m_strm, endStreamToken);
```

```
    m_strm >> c;
    CHECK_LEXER_READ_STATUS(m_strm, errToken);
}else{
    c = m_buffer.pop();
}
// Проверка на ошибки чтения
if(c == '\r' || c == '\n'){
    ++m_currentLine;
    m_pos = 0;
}
while(ignoredChars.contains(c)){
    if(m_buffer.isEmpty()){
        CHECK_FOR_END_STREAM(m_strm, endStreamToken);
        m_strm >> c;
        CHECK_LEXER_READ_STATUS(m_strm, errToken);
    }else{
        c = m_buffer.pop();
    }
    CHECK_LEXER_READ_STATUS(m_strm, errToken);
    ++m_pos;
}
// Если считанный символ - цифра, то делается предположение, что это число
if (c.isDigit()){
    QString numberStr;
    bool isDotFind = false;     //< флаг обнаружения точки в числе
    //Заполнение строки последовательностью цифр с одной точкой из потока
    while(c.isDigit()){
        numberStr.append(c);
        if(m_strm.atEnd()){
            break;
        }
        if(m_buffer.isEmpty()){
            m_strm >> c;
            CHECK_LEXER_READ_STATUS(m_strm, errToken);
        }else{
            c = m_buffer.pop();
        }
        // Если считанный символ - точка и до этого точек считано не было
        if(c == '.' && !isDotFind){
            // обработка точки отдельно, чтобы цикл мог дальше считывать цифры
            isDotFind = true;
            numberStr.append(c);
            if(m_buffer.isEmpty()){
                CHECK_FOR_END_STREAM(m_strm, endStreamToken);
                m_strm >> c;
                CHECK_LEXER_READ_STATUS(m_strm, errToken);
            }else{
                c = m_buffer.pop();
            }
```

```
        }
    }
    // Возвращает обратно в поток последний считанный символ (не цифру)
    if(!c.isDigit()){
        m_buffer.push(c);
    }
    // Если точка была обнаружена
    if(isDotFind){
        // то привести число к вещественному типу
        m_token = Token("num", numberStr.toDouble(), m_currentLine, m_pos);
    }else{
        // иначе, привести число к целму типу
        m_token = Token("num", numberStr.toInt(), m_currentLine, m_pos);
    }
    // сместить позицию на количество считанных/обработанных символов
    m_pos += numberStr.length();
//Если первый символ не цифра, а буквы, то предположить идентификатор
}else if (c.isLetter()){
    QString id;
    // Заполнение лексемы всеми подряд идущими цифро-буквенными символами
    while(c.isLetterOrNumber()){
        id.append(c);
        if (m_strm.atEnd()){
            break;
        }
        if(m_buffer.isEmpty()){
            m_strm >> c;
            CHECK_LEXER_READ_STATUS(m_strm, errToken);
        }else{
            c = m_buffer.pop();
        }
    }
    // Вернуть в исходный поток последний символ, остановивший цикл
    if (!c.isLetterOrNumber()){
        m_buffer.push(c);
    }
    // Если данный идентификатор имеется в списке зарезервированных слов
    if (reservedWords.contains(id)){
        // то использовать в качестве псевдонима токена саму лексему
        m_token = Token(id, id, m_currentLine, m_pos);
    }else{
        // иначе псевдоним "id"
        m_token = Token("id", id, m_currentLine, m_pos);
    }
    m_pos += id.length();
}else{
    m_token = Token(c, c, m_currentLine, m_pos);
    ++m_pos;
}
```

```
    return token();
}
```

TableDrivenParser.h

```
#ifndef TABLEDRIVENPARSER_H
#define TABLEDRIVENPARSER_H
#include <GrammarSymbol.h>
#include <CodeLexer.h>
#include <QList>
#include <QHash>
#include <QStack>
#include <QString>
#include <QTextStream>
#include <QFile>
#include <QtXml/QDomDocument>
typedef QSet<QString> StringSet;
//! Синтаксический нерекурсивный предиктивный анализатор LL(1)-грамматик
/*!
   По указанному в виде файла описанию LL(1)-грамматики производит синтаксический
   разбор входного потока с использованием указанного лексического анализатора.
   Формирует дерево разбора в виде XML-документа.
 */
class TableDrivenParser{
   public: //methods
      TableDrivenParser(QTextStream *errorOut = 0);
      bool loadGrammar(const QString &filePathName);
      void setGrammar(const QList<GrammarSymbol> &symbols);
      bool analyse(Lexer &lexer,
                   QDomDocument &result,
                   bool isParseTreeInResult = true);
#ifdef QT_DEBUG
      void debug();
#endif
   private: //members
      //! Описание грамматики
      QHash<QString, GrammarSymbol> m_symbols;
      //! Стартовый символ грамматики
      GrammarSymbol *m_startSymbol;
      //! Таблица анализа [NonTerminalAlias][TerminalAlias] = SymbolSequence
      QHash<QString, QHash<QString, SymbolsSequence> > m_analyseTable;
      //! Список зарезервированных терминалов
      StringSet m_reservedTerminals;
      //! Поток для вывода сообщений об ошибках анализа
      QTextStream * m_errorOutStream;
   private: //methods
      TableDrivenParser();
      StringSet first(const SymbolsSequence &symbols, bool *isNoError = 0);
      StringSet follow(const GrammarSymbol *symbol);
```

```
        bool fillAnalyseTable();
        void clearSymbos();
        void printError(const QString &errorMsg,
                        const QString &title = QString());
};
#endif // TABLEDRIVENPARSER_H
```

TableDrivenParser.cpp

```
/*
  File   : main.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
           автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Синтаксический таблично-управляемый анализатор LL(1) грамматик.
*/
#include "TableDrivenParser.h"
#include <GrammarLexer.h>
#include <QDebug>
#include <QStringList>
//TODO: Создать константы для символов "empty", "endLine", "error"
//! Создает экземпляр синтаксического анализатора
/*!
  \param errorOut указатель на поток для вывода сообщений об ошибках анализа.
  */
TableDrivenParser::TableDrivenParser(QTextStream *errorOut)
    :m_errorOutStream(errorOut){
    m_reservedTerminals.insert("id");
    m_reservedTerminals.insert("num");
    m_reservedTerminals.insert("empty");
    m_reservedTerminals.insert("endLine");
    m_reservedTerminals.insert("endStream");
}
//! Поиск всех терминальных символов, с которых может начинаться продукция
/*!
    Подробное описание и назначение функции описано в книге
    "Компиляторы: принципы, технологии и инструментарий" Альфред В. Ахо,
    Моника С. Лам и др.
    \param symbols анализируемая последовательность символов грамматики.
    \param isNoError флаг наличия/отсутствия ошибки в работе функции.
    \return множество символов грамматики, с которых может начинаться \a symbols
  */
StringSet TableDrivenParser::first(const SymbolsSequence &symbols, bool *isNoError){
    StringSet result;
    if (isNoError != 0){
        *isNoError = true;
```

```
}
// Если первый символ последовательности - терминал, то вернуть его
// псевдоним в качестве результирующего множества
if (symbols.at(0)->isTerminal()){
   result.insert(symbols.at(0)->alias);
   return result;
}
// Если переданная последовательность состоит из одного символа грамматики
// и этот символ не терминальный
if(symbols.count() == 1){
   // Для каждой продукции нетерминала
   foreach(const SymbolsSequence &product, symbols.at(0)->products){
      StringSet subResult;
      int lenProduct = product.count();
      int index = 0;
      // Добавить множество его возможных начальных терминалов, а так же
      // начальные терминалы всех предшествующих ему символов.
      for (index = 0; index < lenProduct; ++index){
         // Если очередной символ продукции равен символу, чьи продукции
         // просматриваются (левая рекурсия)
         if(product.at(index) == symbols.at(0)){
            if (isNoError != 0){
               *isNoError = false;
            }
            qDebug() << "Left recursion in first() function";
            // выйти из функции во избежании зацикливания
            return result;
         }
         subResult = first(product.mid(index, 1), isNoError);
         result.unite(subResult);
         if(!subResult.contains("empty")){
            break;
         }
      }
      // Если в продукции не все символы порождают пустой терминал
      if(index != lenProduct){
         // Удалить пустой терминал из результата
         result.remove("empty");
      }
   }
// Если последовательность состоит более чем из одного символа
}else{
   StringSet subResult;
   int lenSequnce = symbols.count();
   int index = 0;
   // найти первый символ в продукции, не порождающий пустой терминал
   // Добавить множество его возможных начальных терминалов, а так же
   // начальные терминалы всех предшествующих ему символов.
   for (index = 0; index < lenSequnce; ++index){
```

```
            subResult = first(symbols.mid(index, 1), isNoError);
            result.unite(subResult);
            if(!subResult.contains("empty")){
                break;
            }
        }
        // Если в продукции не все символы порождают пустой терминал
        if(index != lenSequnce - 1){
            // Удалить пустой терминал из результата
            result.remove("empty");
        }
    }
    return result;
}
//! Поиск всех терминальных символов, которые могут следовать сразу за указанным
/*!
    Подробное описание и назначение функции описано в книге
    "Компиляторы: принципы, технологии и инструментарий" Альфред В. Ахо,
    Моника С. Лам и др.
    \param symbol указанный символ.
    \return {множество символов грамматики, которые могут идти непосредственно за
    \a symbol}
 */
StringSet TableDrivenParser::follow(const GrammarSymbol *symbol){
    StringSet result;
    if (symbol->isTerminal()){
        return result;
    }
    if (symbol == m_startSymbol){
        result.insert("endStream");
    }
    // Для каждого символа грамматики
    foreach(const QString &key, m_symbols.keys()){
        // Если он нетерминальный и не равен исследуемому
        if(!m_symbols[key].isTerminal() && (&m_symbols[key] != symbol)){
            // для каждой его продукции
            foreach(const SymbolsSequence &product, m_symbols[key].products){
                // Найти в продукции индекс последнего вхождения
                // исследуемого символа
                int index = product.lastIndexOf(const_cast<GrammarSymbol*>(symbol));
                // Если исследуемый символ имеется в продукции
                if (index != -1){
                    // Если символ стоит последним в продукции
                    if (index == product.count() - 1){
                        // добавить к результату все конечные символы нетерминала
                        // чья продукция просматривается
                        result.unite(follow(&m_symbols[key]));
                    }else{
                        // Если после символа в продукии есть последовательность
```

```
                StringSet subResult;
                // Найти начальные символы этой последовательности
                subResult = first(product.mid(index + 1));
                // Если среди найденых начальных символов есть пустой
                // символ
                if (subResult.contains("empty")){
                    // добавить к результату все конечные символы нетерминала
                    result.unite(follow(&m_symbols[key]));
                    // удалить пустой символ из найденных начальных
                    subResult.remove("empty");
                }
                // добавить к результату найденные начальные символы
                result.unite(subResult);
            }
        }
    }
  }
  return result;
}
//! Заполнение таблицы анализа с использованием вспомогательных функций first/folow
/*!
  Заполнение таблицы происходит по алгоритму, написанному в книге
  "Компиляторы: принципы, технологии и инструментарий" Альфред В. Ахо,
  Моника С. Лам и др.
  Там же описано назначение таблицы.
  \return \c true, если заполнение таблицы прошло успешно, иначе \c false
 */
bool TableDrivenParser::fillAnalyseTable(){
  bool isNoError = true;
  // Заполнение таблицы анализа (M)
  // Для каждого нетерминального символа грамматики (A)
  foreach (const QString &key, m_symbols.keys()){
    GrammarSymbol *symbol = &m_symbols[key];
    if (!symbol->isTerminal()){
      // для каждой продукции нетерминала A (p)
      foreach (const SymbolsSequence &product, symbol->products){
        StringSet startTerminals = first(product, &isNoError);
        // Если была ошибка при вычислении first()
        if (!isNoError){
          return isNoError;
        }
        // для каждого терминала из first (f)
        foreach (const QString &startTermAlias, startTerminals){
          // Добавляем в таблицу продукцию M[A, f] = p
          m_analyseTable[symbol->alias][startTermAlias] = product;
        }
        // Если среди f есть пустой терминал
        if (startTerminals.contains("empty")){
```

```
                // То для каждого терминала (b) из follow(A)
                StringSet endTerminals = follow(symbol);
                foreach (QString startTermAlias, endTerminals){
                    // Добавляем в таблицу продукцию M[A, b] = p
                    m_analyseTable[symbol->alias][startTermAlias] = product;
                }
            }
        }
    }
    return isNoError;
}
//! Очищает список символов грамматики и добавляет в него зарезервированные.
void TableDrivenParser::clearSymbos(){
    m_symbols.clear();
    foreach (QString alias,  m_reservedTerminals){
        m_symbols.insert(alias, GrammarSymbol(alias));
    }
}
//! Выводит ошибку в указанный при создании анализатора поток
/*!
  \param errorMsg текст сообщения об ошибке.
  \param title префикс сообщения об ошибке.
  */
void TableDrivenParser::printError(const QString &errorMsg, const QString &title){
    QTextStream *errorOut = m_errorOutStream;
    QFile stdErrOut;
    if(errorOut == 0 || !errorOut->device()->isOpen()){
        stdErrOut.open(stderr, QFile::WriteOnly);
        errorOut->setDevice(&stdErrOut);
    }
    *errorOut << QString("%1: %2")
                 .arg(title)
                 .arg(errorMsg)
              << endl;
    if (stdErrOut.isOpen()){
        stdErrOut.close();
    }
}
//! Загружает описание LL(1) грамматики из файла для конфигурирования парсера
/*!
    Файл должен содержать описание по системе БНФ. При описании разрешено
    использовать как латиницу, так и кириллицу.
    Для грамматики предопределены такие терминалы, как \c "id" для идентификаторов
    и \c "num" для вещественных и целых чисел, а так же \c "empty" для пустого
    символа.
  \param filePathName путь к файлу с описанием грамматики.
  \return \c true, если грамматика успешно загружена, иначе false.
  */
```

```
bool TableDrivenParser::loadGrammar(const QString &filePathName){
    const QString errorTitle =
            QObject::trUtf8("Loading grammar");
    const QString invalidLeftPartErrorMsg =
            QObject::trUtf8("Invalid left part in line %1");
    const QString invalidLeftPart2ErrorMsg =
            QObject::trUtf8("Contain one of (%1) in left part of line %2");
    const QString errorOpeningFile =
            QObject::trUtf8("Unable to open file %1\n(%2)");
    QFile fInp(filePathName);
    if (!fInp.open(QFile::ReadOnly)){
        printError(errorOpeningFile
                   .arg(filePathName)
                   .arg(fInp.errorString()), errorTitle);
        return false;
    }
    clearSymbos();
    bool isNoError = true;
    QTextStream strm(&fInp);
    GrammarLexer lexer(strm);
    // Считывание определяемого (порождающего) символа
    Token tkn = lexer.nextToken();
    // Проверка на конец потока
    bool isEndStream = tkn.isSystemToken && tkn.alias == "endStream";
    bool isStartSymbolPassed = false;
    while (!isEndStream && isNoError){
        // Если результатом считывания стала ошибка (начало строки не соответствует
        // регламентированному шаблону)
        isNoError = !(tkn.isSystemToken && tkn.alias == "error");
        if (!isNoError){
            printError(invalidLeftPartErrorMsg.arg(tkn.line), errorTitle);
            // то прервать считывание настроек
            break;
        }
        // Проверка, не содержатся ли предопределенные имена в левой части
        if (m_reservedTerminals.contains(tkn.alias)){
            printError(invalidLeftPart2ErrorMsg
                       .arg(QStringList(m_reservedTerminals.toList()).join(", "))
                       .arg(tkn.line),
                       errorTitle);
            isNoError = false;
            break;
        }
        // Проверить, встречается ли токен впервые
        if (!m_symbols.contains(tkn.alias)){
            m_symbols[tkn.alias] = GrammarSymbol(tkn.alias);
            if (!isStartSymbolPassed){
                m_startSymbol =  &m_symbols[tkn.alias];
                isStartSymbolPassed = true;
```

```
            }
        }
        // Сохранения ссылки на текущий нетерминал, для которого будут
        // формироваться продукции
        GrammarSymbol *currentNonTerm = &m_symbols[tkn.alias];
        // Пропуск символа определения
        lexer.nextToken();
        // Формирование продукции
        tkn = lexer.nextToken();
        bool isEndLine = tkn.isSystemToken && tkn.alias == "endLine";
        SymbolsSequence product;
        while (!isEndLine){
            // Если считанный токен не является разделителем продукций
            if(tkn.alias != "|"){
                // Проверить, встречается ли токен впервые
                if (!m_symbols.contains(tkn.alias)){
                    // если да, то добавить его к списку символов грамматики
                    m_symbols[tkn.alias] = GrammarSymbol(tkn.alias);
                }
                // Добавить символ грамматики, соответсвующий считанному токену
                // в последовательность символов текущей продукции
                product.append(&m_symbols[tkn.alias]);
            }else{ // Если считан разделитель продукций
                // добавить продукции к списку продукций текущего нетерминала
                currentNonTerm->products.append(product);
                product.clear();
            }
            // Считать следующий токен
            tkn = lexer.nextToken();
            isEndLine = tkn.isSystemToken && tkn.alias == "endLine";
        }
        // Добавить в список продукций последнюю
        currentNonTerm->products.append(product);
        // Считать первый токен следующей строки
        tkn = lexer.nextToken();
        isEndStream = tkn.isSystemToken && tkn.alias == "endStream";
    }
    fInp.close();
    if(isNoError){
        isNoError = fillAnalyseTable();
    }
    return isNoError;
}
//! Создает внутреннее описание грамматики на основе переданого списка
/*!
  \param symbols список символов грамматики и их продукций.
  */
void TableDrivenParser::setGrammar(const QList<GrammarSymbol> &symbols){
    clearSymbos();
```

```
    foreach(const GrammarSymbol &symbol, symbols){
        m_symbols.insert(symbol.alias, symbol);
    }
    fillAnalyseTable();
}
//! Анализирует входной поток \a stream на соответствие грамматике
/*!
    Производит анализ входного потока \a stream на соответствие грамматике,
    установленной для парсера с помощью loadGrammar() или setGrammar().
    Все ошибки, возникающие во время анализа выводятся либо в поток, заданный при
    создании объекта парсера, либо в stderr, если поток не был задан.
    В случае успешного анализа функция возрващает \c true и заполняет переданную
    по ссылке конструкцию QDomDocument деревом разбора (при
    \a isParseTreeInResult = \c true, по умолчанию) или ситаксическим деревом (при
    \a isParseTreeInResult = \c false).
    \param lexer лексический анализатор, из которого будут получаться лексемы.
    \param result дерево разбора или синтаксическое дерево, в зависимости от
    \a isSyntaxTreeInResult
    \param isParseTreeInResult флаг, указывающий, что нужно строить
    дерево разбора, а не синтаксическое дерево
 */
bool TableDrivenParser::analyse(Lexer &lexer,
                                QDomDocument &result,
                                bool isParseTreeInResult){
    const QString errorTitle =
            QObject::trUtf8("Analyse");
    const QString streamErrorMsg =
            QObject::trUtf8("Error input stream");
    const QString unknownTokenErrorMsg =
            QObject::trUtf8("Unknown token \"%1\" in line %2 in pos %3");
    const QString expectedTokenErrorMsg =
            QObject::trUtf8("Token \"%1\" expected, but \"%2\" found in line %3 in pos %4");
    const QString emptyAnalyseTable =
            QObject::trUtf8("Error. Analyse table is empty. Possible error in loaded grammar.");
    const QString emptyTableRecordErrorMsg =
            QObject::trUtf8("Empty record in analyse table for nonTerm \"%1\" ") +
            QObject::trUtf8("and Term \"%2\"\n\t(line %3, pos %4)");
    const QString noErrorMsg =
            QObject::trUtf8("No error");
    if (lexer.stream().status() != QTextStream::Ok){
        printError(streamErrorMsg);
        return false;
    }
    if (m_analyseTable.isEmpty()){
        fillAnalyseTable();
        if (m_analyseTable.isEmpty()){
            printError(emptyAnalyseTable, errorTitle);
            return false;
        }
```

```
}
QStack<GrammarSymbol*> stack;
stack.push(&m_symbols["endStream"]);
stack.push(m_startSymbol);
QDomImplementation xmlImpl;
QDomDocument resultDoc;
if(isParseTreeInResult){
   resultDoc = xmlImpl.createDocument("", "ParseTree",
                                      result.doctype());
}else{
   resultDoc = xmlImpl.createDocument("", "SyntaxTree",
                                      result.doctype());
}
QStack<QDomElement> parentElementsStack;
parentElementsStack.push(resultDoc.firstChildElement());
lexer.nextToken();
while (!stack.isEmpty()){
   Token currToken = lexer.token();
   if (currToken.isSystemToken && currToken.alias == "error"){
      // Сообщить об ошибке
      printError(streamErrorMsg, errorTitle);
      return false;
   }
   // Если считанного токена нет среди символов грамматики
   if (!m_symbols.contains(currToken.alias)){
      // Сообщить об ошибке
      printError(unknownTokenErrorMsg
            .arg(currToken.alias)
            .arg(currToken.line)
            .arg(currToken.posInLine),
            errorTitle);
      return false;
   }
   GrammarSymbol *stackSymbol = stack.pop();
   QDomElement treeNode;
   GrammarSymbol *inputSymbol = &m_symbols[currToken.alias];
   // Если символ на вершине стека терминальный
   if (stackSymbol->isTerminal()){
      treeNode = resultDoc.createElement("Terminal");
      treeNode.setAttribute("alias", stackSymbol->alias);
      treeNode.setAttribute("lineInSrc", currToken.line);
      treeNode.setAttribute("posInLine", currToken.posInLine);
      QDomText textNode = (stackSymbol->alias == "empty")
                  ? resultDoc.createTextNode(stackSymbol->alias)
                  : resultDoc.createTextNode(currToken.value.toString());
      treeNode.appendChild(textNode);
      if (stackSymbol->alias == "empty"){
         parentElementsStack.pop().appendChild(treeNode);
      } else if (stackSymbol == inputSymbol){
```

```
            if(stackSymbol->alias != "endStream"){
               parentElementsStack.pop().appendChild(treeNode);
            }
            // Считать новый символ
            lexer.nextToken();
         // Если символ на стеке не пустой и не равен символу входного потока
         }else{
            // Вывести ошибку в случае неравенства символов
            printError(expectedTokenErrorMsg
                     .arg(stackSymbol->alias)
                     .arg(currToken.alias)
                     .arg(currToken.line)
                     .arg(currToken.posInLine),
                     errorTitle);
            return false;
         }
      // Если символ на вершине стека не терминальный
      }else{
         treeNode = resultDoc.createElement(stackSymbol->alias);
         parentElementsStack.pop().appendChild(treeNode);
         // То заменить его на последовательность символов согласно таблице
         // или вывести ошибку в случае отсутствия соответствующей последовательности
         if (m_analyseTable[stackSymbol->alias].contains(inputSymbol->alias)){
            SymbolsSequence product =
                  m_analyseTable[stackSymbol->alias][inputSymbol->alias];
            QListIterator<GrammarSymbol*> i(product);
            i.toBack();
            while(i.hasPrevious()){
               stack.push(i.previous());
               parentElementsStack.push(treeNode);
            }
         // Если нет записи в таблице
         }else{
            // сообщить об ошибке
            printError(emptyTableRecordErrorMsg
                     .arg(stackSymbol->alias)
                     .arg(currToken.alias)
                     .arg(currToken.line)
                     .arg(currToken.posInLine),
                     errorTitle);
            return false;
         }
      }
   }
   result.clear();
   result = resultDoc;
   printError(noErrorMsg, errorTitle);
   return true;
}
```

```
#ifdef QT_DEBUG
void TableDrivenParser::debug(){
   foreach (const QString &key, m_symbols.keys()){
      SymbolsSequence seq;
      seq.append(&m_symbols[key]);
      QString str;
      if (!m_symbols[key].isTerminal()){
         qDebug() << QObject::trUtf8("Symbol: %1\n%2")
                  .arg(key)
                  .arg(str);
         qDebug() << QObject::trUtf8("First: ") << first(seq);
         qDebug() << QObject::trUtf8("Follow: ") << follow(&m_symbols[key]);
         qDebug() << "------" << endl;
      }
   }
}
#endif
```

main.cpp

```
/*
  File   : main.cpp
  Created: 20.01.2012
  Reason : Курсовой проект по дисциплине Лингвистическое и программное обеспечение
           автоматизированных систем.
  Product: tableDrivenParser, Таблично-управляемый парсер входных текстов.
  Author : Ануфриев Артем, студент 4884 группы.
  Purpose:
     Основной модуль программы. Реализация диалога с пользователем.
*/
#include <QtCore/QCoreApplication>
#include <QFile>
#include <Lexer.h>
#include <QTextStream>
#include <QStringList>
#include <QTextCodec>
#include <QString>
#include <TableDrivenParser.h>
bool userBinaryRequest(QTextStream &out,
                       QTextStream &in,
                       const QString &message,
                       const QString & trueResponse,
                       Qt::CaseSensitivity caseSens = Qt::CaseInsensitive){
   out << message << endl;
   QString userResponse;
   in >> userResponse;
   // Если пользователь ответил отказом
   if(QString::compare(userResponse, trueResponse, caseSens)){
      return false;
```

```
    }
    return true;
}
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    const QString defaultGrammarFileName = QString::fromUtf8("grammar.txt");
    const QString defaultSourceFileName = QString::fromUtf8("source.txt");
    const QString defaultParseTreeFileName = QString::fromUtf8("parseTree.xml");
    const QString appVersion("1.0");
    const QString appTitle = QObject::trUtf8("Table driven parser v%1")
                        .arg(appVersion);
    const QString yesResponseChar = QObject::trUtf8("y");
    const QString yesResponseWord = QObject::trUtf8("Yes");
    const QString noResponseChar = QObject::trUtf8("n");
    const QString noResponseWord = QObject::trUtf8("No");
    const QString yesNoMsg = QObject::trUtf8("%1/%2 = %3, %4/%5 = %6 (default)")
                        .arg(yesResponseChar.toLower())
                        .arg(yesResponseChar.toUpper())
                        .arg(yesResponseWord)
                        .arg(noResponseChar.toLower())
                        .arg(noResponseChar.toUpper())
                        .arg(noResponseWord);
    const QString promptEnterPath = QObject::trUtf8("Enter path to file %1\ndefault: %2");
    const QString grammarFileDescription =
          QObject::trUtf8("with grammar description.");
    const QString sourceFileDescription =
          QObject::trUtf8("with text for analyse.");
    const QString loadingFileMsg = QObject::trUtf8("Loading file: %1");
    const QString loadingCompleteMsg = QObject::trUtf8("Loading file complete.");
    const QString promptReEnterFileName =
          QObject::trUtf8("re-enter file name? %1")
          .arg(yesNoMsg);
    const QString goodLuckMsg = QObject::trUtf8("Good luck...");
    const QString analyseSuccessfulMsg = QObject::trUtf8("Analyse Successful.");
    const QString analyseFailMsg = QObject::trUtf8("Analyse fail.");
    const QString promptContinueWork =
          QObject::trUtf8("Continue work with program? %1")
          .arg(yesNoMsg);
    const QString promptSourceFromFile =
          QObject::trUtf8("Load source data from file (Otherwise from standart input)?\n %1")
          .arg(yesNoMsg);
    const QString promptEnterSourceData =
          QObject::trUtf8("Enter source data for analyse:");
    const QString promptSaveParseResult =
          QObject::trUtf8("Save parse tree in file? %1")
          .arg(yesNoMsg);
    const QString parseTreeFileDescripion =
```

```
        QObject::trUtf8("for saving parse tree.");
const QString errorFileOpening =
        QObject::trUtf8("Error opepening file %1");
const QString saveParseTreeComplete =
        QObject::trUtf8("Parse tree writed in file %1");
bool isAppExit = false;
bool isSrcFromFile = false;
QTextStream out(stdout);
QTextStream in(stdin);
out.setCodec(QTextCodec::codecForName("IBM 866"));
in.setCodec(QTextCodec::codecForName("IBM 866"));
out << appTitle << endl;
// Создание объекта парсера
TableDrivenParser parser(&out);
// загрузка гармматики
do{
   out <<  promptEnterPath
         .arg(grammarFileDescription)
         .arg(defaultGrammarFileName)
       << endl;
   QString grammarFileName;
   in >> grammarFileName;
   if (grammarFileName.isEmpty()){
      grammarFileName = defaultGrammarFileName;
   }
   out << loadingFileMsg.arg(grammarFileName) << endl;
   // Если произошла ошибка при загрузке грамматики
   if (!parser.loadGrammar(grammarFileName)){
      // Попробовать загрузить грамматику из файла по умолчанию
      out << loadingFileMsg.arg(defaultGrammarFileName) << endl;
      if (!parser.loadGrammar(defaultGrammarFileName)){
         //Вывести предложение ввести новое имя файла
         isAppExit = !userBinaryRequest(out, in, promptReEnterFileName, yesResponseChar);
         if (isAppExit){
            break;
         }
      // Если загрузка грамматики прошла успешно
      }else{
         out << loadingCompleteMsg << endl;
         break;
      }
   // Если загрузка грамматики прошла успешно
   }else{
      out << loadingCompleteMsg << endl;
      // Выйти из цикла запроса имени файла
      break;
   }
}while (true);
if(!isAppExit){
```

```
    isSrcFromFile = userBinaryRequest(out, in, promptSourceFromFile, yesResponseChar);
}
while(!isAppExit){
    QTextStream srcStream;
    // Если пользователь решит вводить данные с файла
    QFile srcFile;
    // Если пользователь решит вводить данные с клавиатуры
    QString srcData;
    if(isSrcFromFile){
        out << promptEnterPath
               .arg(sourceFileDescription)
               .arg(defaultSourceFileName)
            <<endl;
        QString sourceFileName;
        in >> sourceFileName;
        srcFile.setFileName(sourceFileName);
        // Если не удалось открыть файл
        if(!srcFile.open(QFile::ReadOnly)){
            sourceFileName = defaultSourceFileName;
            srcFile.setFileName(sourceFileName);
            if(!srcFile.open(QFile::ReadOnly)){
                //Вывести предложение ввести новое имя файла
                isAppExit = !userBinaryRequest(out, in, promptReEnterFileName, yesResponseChar);
                if (isAppExit){
                    break;
                }
            }
        }
        // Если файл успешно открыт
        if (srcFile.isReadable()){
            //Создать поток на основе файла для передачи его парсеру
            srcStream.setDevice(&srcFile);
        }
    }else{
        out << promptEnterSourceData << endl;
        // Считывание строки из одного символа конца строки (из буфера)
        in.readLine();
        // Считывание обрабатываемое строки
        srcData = in.readLine();
        srcStream.setString(&srcData);
    }
    // --- Анализ входных данных ---
    QDomDocument analyseResult("ParsingResult");
    // Если анализ прошел успешно
    CodeLexer lexer(srcStream);
    if (parser.analyse(lexer, analyseResult)){
        // Вывести сообщение об успехе
        out << analyseSuccessfulMsg << endl;
        if(userBinaryRequest(out, in, promptSaveParseResult, yesResponseChar)){
```

```
            out << promptEnterPath
                   .arg(parseTreeFileDescripion)
                   .arg(defaultParseTreeFileName)
                << endl;
            QString parseTreeFileName;
            in >> parseTreeFileName;
            QFile parseTreeFile(parseTreeFileName);
            out << loadingFileMsg
                   .arg(parseTreeFileName)
                <<endl;
            if(!parseTreeFile.open(QFile::WriteOnly)){
                out << errorFileOpening
                       .arg(parseTreeFileName)
                    << endl;
                parseTreeFile.setFileName(defaultParseTreeFileName);
                out << loadingFileMsg
                       .arg(defaultParseTreeFileName)
                    <<endl;
                if(!parseTreeFile.open(QFile::WriteOnly)){
                    out << errorFileOpening
                           .arg(defaultParseTreeFileName)
                        <<endl;
                }
            }
            if (parseTreeFile.isOpen()){
                QTextStream xmlResultStream(&parseTreeFile);
                analyseResult.save(xmlResultStream, 4);
                parseTreeFile.close();
                out << saveParseTreeComplete
                       .arg(parseTreeFile.fileName())
                    <<endl;
            }
        }
    }else{
        // Вывести сообщение об ошибке
        out << analyseFailMsg << endl;
    }
    if (isSrcFromFile && srcFile.isOpen()){
        srcFile.close();
    }
    // --- Конец анализа ---
    // Спросить у пользователя, не желает ли он продолжить работу
    isAppExit = !userBinaryRequest(out, in, promptContinueWork, yesResponseChar);
  }
  out << goodLuckMsg << endl;
  return app.exec();
}
```

Используемые программные средства

Qt Framework 4.7.4 – набор библиотек для обеспечения дополнительных возможностей. Бесплатный для некоммерческого использования.

Qt Creator 2.4.0 – среда разработки программ под Qt. Бесплатный для некоммерческого использования.

LibreOffice 3.4.0 Writer – текстовый редактор, аналог **MS Word.** Бесплатный.

Используемая литература

- Ахо Альфред Компиляторы. Принципы, технологии, инструменты / Альфред Ахо, Рави Сети, Джеффри Ульман. -М: Вильямс, 2003
- Бланшетт, Жасмин Qt 4: Программирование GUI на C++: пер. с англ. / Жасмин Бланшетт, Марк Саммерфилд. - М.: Кудиц-Пресс, 2008, 718 с.
- Лингвистическое и программное обеспечение САПР: текст лекций / Д. Ш. Тенешев. - СПб.: СПбГТИ(ТУ). - 2010.

Выводы

В результате выполнения работы были изучены теоретические основы по устройству и работе трансляторов, по формальным грамматики. Был получен опыт написания лексического и синтаксического анализатора.