Министерство образования Российской Федерации

#### ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

##### Кафедра «Математическое обеспечение и применение ЭВМ»

«Утверждаю»

## Зав. кафедрой "МО и ПЭВМ"

"\_\_" \_\_\_\_\_\_ 2017 г.

### Пояснительная записка

к курсовой работе по дисциплине

«Теория языков программирования и методы трансляции»

на тему: "Разработка транслятора"

Автор работы: Юрченко С. А.

Специальность 09.03.04 («Программная инженерия»)

Обозначение курсовой работы ПГУ 09.03.04 - 05КР151.01 ПЗ

Группа 15ВП1

Руководитель работы Дорофеева О.С., к.т.н., доцент

Работа защищена «\_\_» \_\_\_\_ 2017 г. Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Пенза 2017 г.

Реферат

Изм.

Лист

№ докум.

Подпись

Дата

Лист

3

*05КР151.01 ПЗ*

Разраб.

*Юрченко С. А.*

Провер.

*Дорофеева О.С.*

Т. Контр.

Н. Контр.

Утверд.

Разработка отдельных фаз компиляции для заданного входного языка

Пояснительная записка

Лит.

Листов

53

15ВП1

Реценз.

Масса

Масштаб

Пояснительная записка содержит 53 листов, 6 рисунков, 3 таблиц, 8 использованных источников, 2 приложения.

ТРАНСЛЯТОР, КОМПИЛЯТОР, ИНТЕРПРЕТАТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, ГЕНЕРАТОР КОДА, МАГАЗИННЫЙ АВТОМАТ, ГРАММАТИЧЕСКИЙ РАЗБОР

Целью курсового проектирования является разработка учебного транслятора с заданного языка.

Разработка проводилась на базовом языке программирования C# в среде объектно-ориентированного программирования Visual Studio 2017.

Разработка проведена с использованием операционной системы Windows 10.

Осуществлено функциональное тестирование разработанного транслятора, которое показало корректность его работы.

Мною был разработан парсер, лексер.

Содержание

Введение…………………………………………………………………………………………………………….……5

# 1 Методы грамматического разбора…………………………………...…………………………..…..6

1.1 Разбор сверху-вниз……………………………………………...……………………….…...6

1.2 LL(k) - языки и грамматики……………………………………………...…………....…7

1.3 Метод рекурсивного спуска……………...…………………………………….………11

## 2 Разработка транслятора……………………………………………………..……………...….…12

2.1 Анализ требований……………………………………………………...……....…………..12

2.1.2 Требования к программным средствам……………………………...…..12

2.2 Проектирование………………………………………………………………..………….…13

2.2.1 Анализ предметной области……………………………………………...…...13

2.2.2 Разработка генератора кода или модуля интерпретации…...…..14

2.2.3 Постфиксная запись……………………………………………………...………..14

2.2.4 Проектирование интерфейса пользователя………………………...….18

2.2.5 Проектирование модуля интерпретации………………………………..19

2.2.6 Проектирование модуля Lexer…………………………………………..22

2.2.7 Проектирование модуля Parser…………………………………………….. 25

2.2.8 Проектирование модуля Form1……………………………………………. 28

2.3 Кодирование…………………………………………………………………………………. 28

2.4 Тестирование………………………………………………………………………………... 29

# Заключение……………………………………………………………………………………………… 29

Список использованных источников………………………………………………………. 30

Приложение А. Листинг программного текста транслятора…………………... 31

Приложение Б. Результаты тестирования……………………………………………… 51

Изм.

Лист

№ докум.

Подпись

Дата

Лист

4

**Введение**

Программа, написанная на машинном языке какой-либо конкретно взятой ЭВМ, может быть выполнена на ней непосредственно. Однако программирование на машинном языке является очень трудоемким, и поэтому программы пишутся на языках, имеющих более символическую и стилизованную форму, которая легче для человеческого понимания. Отсюда возникает потребность в переводе текста программы из одной формы представления в другую. Эту функцию выполняет транслятор – программа, которая переводит входную программу на исходном языке в эквивалентную ей выходную программу на результирующем языке. Чтобы создать транслятор, необходимо выбрать входной и выходной языки. Одной из разновидностей транслятора является компилятор – программа, которая осуществляет перевод исходной программы в эквивалентную ей объектную программу на языке машинных команд или языке ассемблера. Компиляторы являются самым распространенным видом трансляторов. Они имеют самое широкое практическое применение благодаря широкому распространению всевозможных языков программирования.

В ходе данной курсовой работы производится разработка компилятора с заданного языка программирования.

Разработка компилятора осуществляется в несколько этапов:

1. разработка лексического анализатора;
2. разработка синтаксического анализатора;
3. разработка модуля интерпретации;
4. разработка графической части

На этапе лексического анализа производится выделение лексем из исходной программы и построение цепочки символов, которая необходима на этапе синтаксического анализа.

На этапе синтаксического анализа происходит анализ синтаксиса исходной программы, выявление ошибок. Для реализации этой задачи производится построение соответствующей грамматики, использование ее для получения необходимых правил и применение этих правил в процессе анализа.

На этапе выполнения модуля интерпретации производится постфиксная запись выражений, которые посылаются на выполнение.

1. **Методы разбора**
   1. **Разбор сверху-вниз**

Грамматический разбор – процедура построения синтаксического дерева для конкретного предложения языка. Построение такого дерева позволяет однозначно доказать, что анализируемая строка языка является допустимой, т.е. принадлежит конкретному языку.

Грамматический разбор может осуществляться в разной последовательности, причем дерево можно строить как «сверху» – от аксиомы, так и «снизу» – от предложения. Соответственно существуют нисходящий и восходящий методы разбора. При этом предложения языка можно рассматривать слева направо и наоборот. Соответственно различают левосторонний и правосторонний разбор. Левосторонний разбор используют чаще, т. к. мы читаем слева направо.

Метод разбора «сверху-вниз» применим, если грамматика не содержит правил с левосторонней рекурсией. Метод предполагает последовательное выдвижение гипотез, начиная с аксиомы, их доказательство посредством подбора правил, которым удовлетворяет разбираемый фрагмент, и выдвижения новых гипотез относительно не разобранной части предложения. Правила подстановки также должны проверяться, начиная с самого сложного, иначе цель разбора не будет достигнута. При наличии правил с левосторонней рекурсией процедура разбора становится бесконечной. В общем случае при разборе возможны возвраты, поскольку может быть выбрано неподходящее правило подстановки.

**Пример**. Разобрать строку «-45»

Правила грамматики:

а) ::= |

б) ::= |,

в) ::= 0|1|2|3|4|5|6|7|8|9,

г) ::= +| - .

Последовательность разбора представлена на рисунке 1

Недостаток метода заключается в том, что в программе разбора следует хранить всю информацию о каждом его шаге.

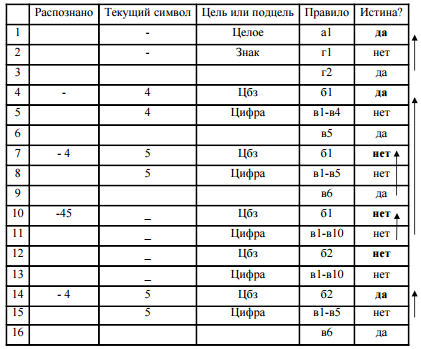


Рисунок 1. Последовательность разбора

## LL(k) языки и грамматики

Продолжением идеи, положенной в основу метода рекурсивного спус­ка, является предложение использовать для выбора одной из множества альтер­натив не один, а несколько символов входной цепочки. Однако напрямую переложить алгоритм выбора альтернативы для одного символа на такой же алгоритм для цепочки символов не удастся — два соседних символа в цепочке на самом деле могут быть выведены с использованием различных правил грамматики, по­этому неверным будет напрямую искать их в одном правиле. Тем не менее суще­ствует класс грамматик, основанный именно на этом принципе — выборе одной альтернативы из множества возможных на основе нескольких очередных симво­лов в цепочке. Это так называемые LL(k)-грамматики. Правда, алгоритм работы распознавателя для них не так очевидно прост, как рассмотренный выше алго­ритм рекурсивного спуска.

Грамматика обладает свойством LL(k), k > 0, если на каждом шаге вывода для однозначного выбора очередной альтернативы МП-автомату достаточно знать символ на верхушке стека и рассмотреть первые k символов от текущего поло­жения считывающей головки во входной цепочке символов.

Грамматика называется *LL(k)-грамматикой,* если она обладает свойством LL(k) для некоторого k > 0.

Название «LL(k)» несет определенный смысл. Первая литера «L» происходит от слова «left» и означает, что входная цепочка символов читается в направлении слева направо. Вторая литера «L» также происходит от слова «left» и означает, что при работе распознавателя используется левосторонний вывод. Вместо «k» в названии класса грамматики стоит некоторое число, которое показывает, сколь­ко символов надо рассмотреть, чтобы однозначно выбрать одну из множества аль­тернатив. Так, существуют LL(1) – грамматики, LL(2) – грамматики и другие классы. Очевидно, что LL(0) – грамматика бессмысленна, т.к. этом случае разбор вообще не зависит от входной цепочки.

В совокупности все LL(k) – грамматики для всех k>0 образуют класс LL – грамматик.

На рисунке 2 схематично показано частичное дерево вывода для некоторой LL(k) – грамматики. В нем ω обозначает уже разобранную часть входной цепочки α, ко­торая построена на основе левой части дерева y. Правая часть дерева х — это еще не разобранная часть, а А — текущий нетерминальный символ на верхушке стека МП-автомата. Цепочка х представляет собой незавершенную часть цепочки вы­вода, содержащую как терминальные, так и нетерминальные символы. После за­вершения вывода символ А раскрывается в часть входной цепочки υ, а правая часть дерева х преобразуется в часть входной цепочки τ. Свойство LL(k) предпо­лагает, что однозначный выбор альтернативы для символа А может быть сделан на основе k первых символов цепочки υτ, являющейся частью входной цепоч­ки α.

Алгоритм разбора входных цепочек для LL(k)-грамматики носит название «k-предсказывающего алгоритма». Принципы его выполнения во многом соответст­вуют функционированию МП-автомата с той разницей, что на каждом шаге работы этот алгоритм может просматривать k символов вперед от текущего поло­жения считывающей головки автомата.



Рисунок 2. Схема построения дерева вывода для LL(k)-грамматики

Для LL(k)-грамматик известны следующие полезные свойства:

* Всякая LL(k)-грамматика для любого k>0 является однозначной;
* Существует алгоритм, позволяющий проверить, является ли заданная грамма­тика LL(k)-грамматикой для строго определенного числа k.

Кроме того, известно, что все грамматики, допускающие разбор по методу рекур­сивного спуска, являются подклассом LL(1)-грамматик. То есть любая грамма­тика, допускающая разбор по методу рекурсивного спуска, является LL(1)-грамматикой (но не наоборот!).

Есть, однако, неразрешимые проблемы для произвольных КС-грамматик:

* Не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика LL(k)-грамматикой для некоторого произвольного числа k;
* Не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику к виду LL(k)-грамматики для некоторого k (или доказать, что преобразование невозможно).

Это несколько ограничивает применимость LL(k)-грамматик, поскольку не все­гда для произвольной КС-грамматики можно очевидно найти число k, для кото­рого она является LL(k)-грамматикой, или узнать, существует ли вообще для нее такое число k.

Для LL(k)-грамматики при k>1 совсем не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с k различ­ных терминальных символов.

Для LL(1)-грамматики, очевидно, для каждого нетерминального символа не мо­жет быть двух правил, начинающихся с одного и того же терминального симво­ла. Однако это менее жесткое условие, чем то, которое накладывает распознава­тель по методу рекурсивного спуска, поскольку в принципе LL(1)-грамматика допускает в правой части правил цепочки, начинающиеся с нетерминальных символов, а также λ-правила.

Поскольку все LL(k)-грамматики используют левосторонний нисходящий рас­познаватель, основанный на алгоритме с подбором альтернатив, очевидно, что они не могут допускать левую рекурсию. Поэтому никакая леворекурсивная грам­матика не может быть LL-грамматикой. Следовательно, первым делом при по­пытке преобразовать грамматику к виду LL-грамматики необходимо устранить в ней левую рекурсию (соответствующий алгоритм был рассмотрен выше).

Класс LL-грамматик широк, но все же он недостаточен для того, чтобы покрыть все возможные синтаксические конструкции в языках программирования (к ним относим все детерминированные КС-языки). Известно, что существуют детер­минированные КС-языки, которые не могут быть заданы LL(k)-грамматикой ни для каких k. Однако LL-грамматики удобны для использования, поскольку по­зволяют построить распознаватели с линейными характеристиками (линейной зависимостью требуемых для работы алгоритма распознавания вычислительных ресурсов от длины входной цепочки символов).

**1.3 Метод рекурсивного спуска**

Метод рекурсивного спуска основывается на синтаксических диаграммах языка. Согласно этому методу для каждого нетерминала разрабатывают рекурсивную процедуру. Основная программа вызывает процедуру аксиомы, которая вызывает процедуры нетерминалов, упомянутые в правой части аксиомы и т. д. В эти же процедуры встраивают семантическую обработку распознанных конструкций.

Для объяснения принципов, лежащих в основе метода рекурсивного спуска, рассмотрим задачу вычисления значения арифметической формулы. Будем рассматривать формулы, состоящие из целочисленных значений, бинарных операций сложения (+), вычитания (–), умножения (\*) и деления нацело (/), а также круглых скобок. Как обычно, приоритеты операций умножения и деления равны и их приоритет больше, чем приоритеты операций сложения и вычитания, причем приоритеты этих операций также равны. Будем называть операции + и – операциями типа сложения, а операции \* и / – операциями типа умножения. Круглые скобки используются для изменения стандартного порядка выполнения операций. Наша задача заключается в написании программы, вычисляющей значение формулы.

Изучаемые нами формулы можно представить следующим образом: T1+T2+…+Tn, где Ti – это формула вида F1i\*F2i\*…\*Fmi. В свою очередь, Fji – это либо число, либо произвольная формула, заключенная в круглые скобки. Представим себе процесс вычисления значения формулы. Вначале вычисляется F11, далее мы выясняем, какая операция следует за F11. Если это операция типа умножения, то мы, зная ее левый операнд, вычисляем правый операнд и выполняем операцию. Тем самым, мы получим левый операнд для возможных следующих операций типа умножения. Когда мы закончим вычисление формулы F1\*F2\*…\*Fn, то, увидим операцию типа сложения, и процесс вычисления такой формулы будет аналогичен только что описанному процессу.

**2. Разработка транслятора**

**2.1 Анализ требований**

### 2.1.1 Требования к интерфейсу пользователя

Транслятор должен обладать простым и удобным интерфейсом, который должен включать в себя:

* Поле ввода текста программы
* Поле вывода значений программы
* Кнопку открытия сохраненной программы
* Кнопку сохранения текста программы
* Кнопку выполнения введенной программы

### 2.1.2 Требования к программным средствам

Назначение: трансляция программы и её выполнение.

Функции:

1. Ввод текста программы

2. Лексический анализ программы

3. Синтаксический анализ программы

4. Выдача сообщений об ошибках

5. Вывод результатов выполнения программы

Технические средства: IBM PC

Операционная система: Windows 10

Среда программирования: Microsoft Visual Studio 2017

Язык программирования: C#.

**2.2 Проектирование**

## 2.2.1 Анализ предметной области

Разработать учебный транслятор в форме интерпретатора с языка, определенного соответствующей формальной грамматикой. Базовое описание языка имеет следующий вид:

<Программа> ::= <Объявление переменных> <Описание вычислений>

<Описание вычислений> ::= BEGIN <cписок присваиваний> END

<Объявление переменных> ::= VAR <список переменных> : тип ;

<Список переменных> ::= <Идент>|<Идент>,<Список переменных>

<Список присваиваний> ::= <Присваивание>|<Присваивание> <Список присваиваний>

<Присваивание> ::= <Идент> = <Выражение>;

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>

<Подвыражение> ::= (<Выражение>) | <Операнд> | <Подвыражение>

<Бин.оп.><Подвыражение>

<Ун.оп> ::= вид

<Бин.оп.> ::= вид

<Операнд> ::= <Идент>|<Конст>

<Идент> ::= <Буква><Идент>|<Буква>

<Конст> ::= вид

Необходимо реализовать операторы:

READ<список переменных>,

WRITE<список переменных>,

CASE <Выражение> OF <Список выбора> END\_CASE,

Тип переменных – INTEGER.

2.2.2 Разработка генератора кода или модуля интерпретации

Данный компонент транслятора, в основном и определяет его форму: компилятор или интерпретатор, поскольку основное различие между компилятором и интерпретатором состоит в том, что интерпретатор, в отличие от компилятора, не порождает объектную программу, которая должна выполняться при необходимости впоследствии, а непосредственно выполняет ее сам.

При разработке генератора кода целесообразно, чтобы на выходе генератора формировалась последовательность команд языка ассемблера с тем, чтобы впоследствии можно было задействовать соответствующее ассемблирование.

При разработке модуля интерпретации в качестве промежуточной формы исходной программы наиболее часто используется постфиксная форма записи, которая позволяет достаточно легко реализовывать процесс выполнения (интерпретации) транслируемой программы.

2.2.3 Постфиксная запись

Постфиксная запись представляет собой такую запись арифметического выражения, в которой сначала записываются операнды, а затем – знак операции. Например, для выражения a + b \* c постфиксная запись будет a b c \* +. Здесь операндами операции \* будут b и c (два ближайших операнда), а операндами операции + будут а и составной операнд b c \*. Эта запись удобна тем, что она не требует скобок. Например, для выражения (a + b) \* c постфиксная запись будет a b + c \*. В этой записи не требуется ставить скобки для того, чтобы изменить порядок вычисления, зависящий от приоритета операций, как в исходном выражении.

Алгоритм перевода в постфиксную запись обрабатывает исходный массив лексем и строит новый массив из тех же лексем, расположенных в другом порядке. Кроме того, необходим еще стек – аналогичный массив, используемый для временного хранения операций.

Для унарного минуса в арифметическом выражении существуют несколько различных способов его записи и идентификации:

1. Унарный минус записывается как бинарная операция, т.е. вместо, например -В записывается 0-В;
2. Для обозначения унарного минуса используется новый знак, например @;
3. Унарный минус может определяться по контексту: он может стоять либо в начале выражения, либо сразу после открывающей скобки, например

-х+а(-в+с)/(-d+f)

Основные правила преобразования инфиксной записи выражения в постфиксную заключаются в следующем.

Считанные операнды добавляются к постфиксной записи, операции записываются в стек.

Если операция в вершине стека имеет больший (или равный) приоритет, чем текущая считанная операция, то операция из стека добавляется к постфиксной записи, а текущая операция заносится в стек. В противном случае (при низшем приоритете) происходит только занесение текущей операции в стек.

Считанная открывающая скобка заносится в стек.

После считывания закрывающей скобки все операции до первой открывающей скобки извлекаются из стека и добавляются к постфиксной записи, после чего и открывающая, и закрывающая скобки отбрасываются, т.е. не помещаются ни в постфиксную запись, ни в стек.

После считывания всего выражения, оставшиеся в стеке операции добавляются к постфиксной записи.

Рассмотрим пример преобразования для следующей инфиксной записи:

A+B\*C

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг | Текущий символ (лексема) | Постфиксная запись | Стек |
| 1. | A | A |  |
| 2. | + | A | + |
| 3. | B | AB | + |
| 4. | \* | AB | \*+ |
| 5. | C | ABC | \*+ |
| 6. |  | ABC\*+ |  |

Рассмотрим пример преобразования для следующей инфиксной записи:

(A+B) \*C

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг | Текущий символ (лексема) | Постфиксная запись | Стек |
|  | ( |  | ( |
|  | A | A | ( |
|  | + | A | +( |
|  | B | AB | +( |
|  | ) | AB+ |  |
|  | \* | AB+ | \* |
|  | C | AB+C | \* |
|  |  | AB+C\* |  |

Рассмотрим пример преобразования для следующей инфиксной записи:

-A\*((-B+C)/D-F)

|  |  |  |  |
| --- | --- | --- | --- |
| Шаг | Текущий символ (лексема) | Постфиксная запись | Стек |
|  | - |  | - |
|  | A | A | - |
|  | \* | A- | \* |
|  | ( | A- | (\* |
|  | ( | A- | ((\* |
|  | - | A- | -((\* |
|  | B | A-B | -((\* |
|  | + | A-B- | +((\* |
|  | C | A-B-C | +((\* |
|  | ) | A-B-C+ | (\* |
|  | / | A-B-C+ | /(\* |
|  | D | A-B-C+D | /(\* |
|  | - | A-B-C+D/ | -(\* |
|  | F | A-B-C+D/F | -(\* |
|  | ) | A-B-C+D/F- | \* |
|  |  | A-B-C+D/F-\* |  |

Постфиксная запись выражения позволяет производить его вычисление следующим образом.

Если лексема является операндом, то она записывается в стек. Если лексема является операцией, то указанная операция выполняется над последними элементами (последним элементом), записанными в стек, и эти элементы (элемент) заменяются в стеке результатом операции.

## 2.2.4 Проектирование интерфейса пользователя

При запуске программы на экране появляется окно, представленное на рисунке 3.

Сообщение об ошибках выводится в нижней левой части главного окна программы. При нажатии на кнопку «Run» выполняются лексический анализ, синтаксический анализ, составляется постфиксная запись выражений. При отсутствии ошибок, запускается выполнение программы.

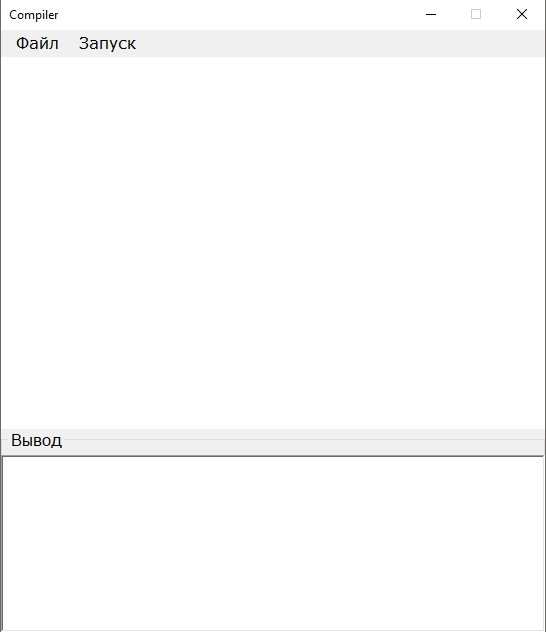


Рисунок 3. Интерфейс программы

Кнопка «Файл/Создать проект» очищает окно ввода кода

Кнопка «Файл/Открыть файл» открывает диалоговое окно открытия программы

Кнопка «Файл/Сохранить файл» открывает диалоговое окно сохранения программы

Кнопка «Файл/Выход» выход из программы

Кнопка «Запуск –Run» запускает выполнение программы

## 2.2.5 Проектирование модуля интерпретации

На начальном этапе реализации константы (CONSTANTS) и переменные (IDENTIFIES) помещаются в формируемую запись в порядке их появления в исходном массиве.

s1.Push(CONSTANTS[CounterConstants++]);

s1.Push(IDENTIFIES[CounterIdentifies++]);

При появлении операции в исходном массиве:

* 1. Если в стеке нет операций или верхним элементом стека является открывающая скобка, операция кладётся в стек

if (operations.Peek() == "16" || operations.Peek() == "00")

{

operations.Push(TokensStack.Pop());

break;

}

if (TokensStack.Peek() == "16") // (

{

operations.Push(TokensStack.Pop());

break;

}

* 1. если новая операция имеет больший приоритет, чем верхняя операция в стеке, то новая операции кладётся в стек

if (Convert.ToInt32(TokensStack.Peek()) > Convert.ToInt32(operations.Peek()))

{

operations.Push(TokensStack.Pop());

}

* 1. Если новая операция имеет меньший или равный приоритет, чем верхняя операция в стеке, то операции, находящиеся в стеке, до ближайшей открывающей скобки или до операции с приоритетом меньшим, чем у новой операции, перекладываются в формируемую запись, а новая операции кладётся в стек

while (Convert.ToInt32(TokensStack.Peek()) > Convert.ToInt32(operations.Peek()) || operations.Peek() != "16")

{

if (operations.Peek() == "00")

break;

s1.Push(BinOp(operations.Pop()));

}

Открывающая скобка кладётся в стек.

if (operations.Peek() == "16" || operations.Peek() == "00")

{

operations.Push(TokensStack.Pop());

break;

}

Закрывающая скобка выталкивает из стека в формируемую запись все операции до ближайшей открывающей скобки, открывающая скобка удаляется из стека

if (TokensStack.Peek() == "17") // ) => +/- to stack

{

while (operations.Peek() != "16")

{

s1.Push(BinOp(operations.Pop()));

}

TokensStack.Pop();

operations.Pop();

break;

}

После того, как мы добрались до конца исходного выражения, операции, оставшиеся в стеке, перекладываются в формируемое выражение.

while (operations.Peek() != "00")

s1.Push(BinOp(operations.Pop()));

Краткая спецификация модуля интерпретации:

Модуль Shell

Название процедуры: Equal

Назначение процедуры: Реализует постфиксную запись.

Название процедуры: BinOp

Назначение процедуры: переводит число в знак

Название процедуры: FindOperations

Назначение процедуры: проход по коду в поисках функций

2.2.6 Проектирование модуля Lexer

Ключевым элементом модуля является структура «Ключевое слово»

private struct KeyWords

{

public string Words;

public string WordKey;

public KeyWords(string a, string b)

{

Words = a;

WordKey = b;

}

}

А так же регулярное выражение вида

@"(?<char>[^:;\(\)\=\+\-\/\s\,]\*)(?<separator>[:;\(\)\=\+\-\/\s\,]{1})?";

Регулярное выражение состоит из двух групп – char и separator

?<char>[^:;\(\)\=\+\-\/\s\,]\*)  - группа Char

(?<separator>[:;\(\)\=\+\-\/\s\,]{1})?";- группа separator

Группа Char исключает все символы :;()=+- \s

Группа separator включает следующие символы - :;()=+- \s

{1} - указывает точное число вхождений в строку, например шаблон (@"не{2}") найдет слово “нее” в строке.

\s - Пробельный символ (пробел, табуляция, перевод строки и т. п.).

Модуль наполняет стек TOKENS списком полученных токенов.

Основные методы модуля Lexer представлены в таблице 1.

Таблица 1. Основные методы модуля Lexer

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Основные функции |
| public Lexer(string Code) | Конструктор | Преобразует код в удобный для работы формат – верхний регистр, удаляет лишний пробелы |
| private void LexerOut() | При помощи регулярного выражения разбивает текст | Разбивает строку на лексиомы и делает проверку на их принадлежность определенной группе |
| private bool IsKeyword(string word, KeyWords[] kwd) | Метод делает проверку является ли строка ключевым словом |  |
| private bool IsIdenOrConst(string word, string regular, string ID, List<string> lst, int len) | Метод делает проверку является ли строка переменной |  |
| private void AddError(string word) | Проверка строки на наличие ошибок | Возвращает следующие ошибки: Недопустимый символ,  лишком длинное имя переменной, Повторное объявление переменной, Необъявленная переменная, Неверное имя переменной |

Порядок работы:

1. Вызываем конструктор. В качестве входного параметра передаем код программы. В конструкторе вызывается метод LexerOut

Lexer lex = new Lexer(txtCode.Text);

1. Метод LexerOut используя регулярные выражение, парсит исходный текст программы.

Regex rgx = new Regex(pattern);

this.Code = rgx.Replace(this.Code, replacement);

this.Code = this.Code.ToUpper();

Регулярные выражения – это один из способов поиска подстрок (соответствий) в строках. Осуществляется это с помощью просмотра строки в поисках некоторого шаблона.  
Для того чтобы использовать Regex в своих программах необходимо в список используемых пространств имен добавить:  
**using System.Text.RegularExpressions;**  
Далее, в коде самой программы необходимо создать экземпляр Regex:  
**Match newReg = new Regex(pattern,options);**  
Все найденные соответствия в тексте помещаются в тип MatchCollection  
**MatchCollection matches;**

**Match** - Представляет результаты из отдельного совпадения регулярного выражения.  
И далее в этот объект поместить текст, в котором необходимо произвести поиск:  
**matches = optionRegex.Matches(text);**  
В результате в matches появляются все результаты парсинга. Мы можем посмотреть сколько их (matches.Count), можем узнать значение конкретного элемента (matches[N].Value) pattern – образец или условие для поиска, например, если необходимо найти слово “не” в строке “не может быть”, то pattern для поиска будет выглядеть так: @"не". В простейшем случае, можно обойтись без условия поиска, тогда найдено будет значение точно повторяющее pattern.

1. Метод работает в цикле до тех пор, пока находим совпадения (удовлетворяющие регулярному выражению)

while (match.Success)

1. В цикле делается проверка к какой группе относится символ.

if (match.Groups["char"].Length > 0)

if ((match.Groups["separator"].ToString() != " ") && (match.Groups["separator"].ToString() != "\n"))

1. В зависимости от группы происходит проверка – это ключевое слово или переменная. (методы IsIdenOrConst, IsKeyword – описание в таблице 2.2.6)
2. В результате работы модуля получаем стек, со списком полученных токенов.  
   2.2.7 Проектирование модуля Parser

Модуль предназначен для заполнения списка полученных токенов, идентификаторов, констант.

Ключевым элементом данного модуля является таблица состояний

string[,] States = { { "04", "53", "03", "11", "02", "10", "51", "00" },

{ "54", "11", "17", "51", "16", "00", "00", "00" },

{ "54", "11", "09", "56", "08", "58", "00", "00" },

{ "54", "11", "58", "12", "00", "00", "00", "00" },

{ "11", "58", "12", "00", "00", "00", "00", "00" },

{ "57", "55", "10", "00", "00", "00", "00", "00" },

{ "60", "17", "58", "00", "00", "00", "00", "00" } };

Основные методы модуля Parser представлены в таблице 2.

Таблица 2. Основные методы модуля Parser

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Основные функции |
| private void ToStack() | Кладет на вершину стека данные | Заполняются два стека TokensStack и TokensShell.  TokensShell используется в модуле Shell. TokensStack заполнятся данными полученными в модуле lexer |
| private int Parsing() | Метод предназначен для парсинга стека | В цикле, пока стек не опустеет, разбирает стек и делает проверку согласно таблице. |

Порядок работы:

1. Вызывается конструктор. В качестве параметра передаем стек лекскем, полученных в результате выполнения метода Lexer

Parser pars = new Parser(lex.TOKENS);

1. Вызывается метод Parsing (описание метода в таблице 2);

public Parser(Stack<string> Tokens)

{

this.Tokens = Tokens;

Parsing();

}

1. Метод Parsing вызывает метод ToStack (описание метода в таблице 2.2.7)
2. Далее в цикле

while ((StatesStack.Count != 0) || (TokensStack.Count != 0))

пока стеки не пустые, выполняем слудеющие действия:

1. «снимаем» значение с вершины стека StatesStack

StatesStack.Pop()

1. «снимаем» значение с вершины стека TokensStack

TokensStack.Pop()

1. Проверям по таблице состояний и наполняем стек StatesStack значениями из таблицы или возвращаем ошибку.

case "01": { int i = 0; while (States[0, i] != "00") StatesStack.Push(States[0, i++]); break; }

default: { DebugMsg = "error: expected \"VAR\""; return 0; }

Порядок работы с таблицей состояний –

1.Снимаем значение с вершины стека StatesStack

2.Проверяется значение, в зависимости от этого значения выполняется «ветка» работы

3. Снимаем значение с вершины стека TokenStack

4. Проверяем значение, в зависимости от токена должен выполняться определенный порядок действий. Порядок этих действий указан в таблице States

5. Заполняем StatesStack стека значениями из таблицы States или выводим ошибку

Начальное состоянием стека StatesStack = 50. Проверем, что лежит на вершине стека TokenStack, если «01» - это означает, что код программы начинается с ключевого слова VAR- заполним стек значениями из первой строки таблицы States, в противно случае вернем ошибку

switch (StatesStack.Pop())

{

case "50":

{

switch (TokensStack.Pop())

{

case "01": { int i = 0; while (States[0, i] != "00") StatesStack.Push(States[0, i++]); break; }

default: { DebugMsg = "error: expected \"VAR\""; return 0; }

}

break;

}

"04", "53", "03", "11", "02", "10", "51" –

04 - var

53 – READ/ WRITE/ CASE

03 - begin

11 - ;

02 – integer

10 - :

51 - переменная

2.2.8 Проектирование модуля Form1

Главная форма приложения.

Кнопка – «Запуск –Run» последовательно выполняет следующие функции:

1. Вызов лексического анализатора (описание в пункте 2.2.6)
2. Парсинг (описание в пункте 2.2.7)
3. Интерпретация (описание в пункте 2.2.5)
4. Вывод ошибок если такие есть.

Основные методы модуля Form1 представлены в таблице 3.

Таблица 3. Основные методы модуля Form1

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | Основные функции |
| private static void Run(object INI) | Выполнение введенного текста. |  |
| private static string Operation(Stack<string> a, List<ConstIdent> CI) |  | Выполнение математических операций |
| private void сохранитьФайлToolStripMenuItem\_Click(object sender, EventArgs e) | Сохранение текста программы |  |
| private void открытьФайлToolStripMenuItem\_Click(object sender, EventArgs e) | Загрузка текста программы |  |
| private void runToolStripMenuItem\_Click(object sender, EventArgs e) | Запуск программа на выполнение |  |
| private static string TryFind(string a, List<ConstIdent> ci) | Поиск переменной |  |

# 2.3 Кодирование

Программа была написана на языке программирования C# в среде Microsoft Visual Studio 2017. Исходный текст программы представлен в приложении А.

## 2.4 Тестирование

Проверка разработанной программы заключалась в её активном использовании с целью выявления ошибок и некорректной работы. В результате этого ошибок обнаружено не было, что позволяет сделать вывод о работоспособности программы. Этапы тестирования представлены в таблице 2.4. Скриншоты результатов тестирования представлены в приложении Б.

Таблица 2.4 – план тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № теста | Цель теста | Ожидаемый результат | Полученный результат |
| 1 | Проверка вывода ошибок при вводе не корректного кода. | Вывод сообщения об ошибки | Полученный результат совпадает с ожидаемым (рис. 4,5,6). |
| 2 | Вычисление необъявленной переменной | Вывод сообщения об ошибки | Полученный результат совпадает с ожидаемым (рис. 6). |
| 3 | Ввод корректного кода программы | Программа выводит результат работы программы | Полученный результат совпадает с ожидаемым (рис. 7). |

# Заключение

В данной курсовой работе была поставлена задача разработки компилятора к заданной грамматике.

В ходе решения задачи был разработан интерфейс программы, алгоритмического анализатора. Программа производит анализ исходного текста и выдает сообщения об ошибках, если они есть. Было произведено кодирование на языке программирования C# в среде объектно-ориентированного программирования Visual Studio 2017 и тестирование, в ходе которого было выявлено, что программа решает задачу корректно и устойчиво работает на тестовом наборе данных.

**Список использованных источников**

1. Д. Креншоу «Пишем компилятор»
2. Р. Хантер «Проектирование и конструирование компиляторов»
3. Компаниец Р.И., Маньков Е.В., Филатов Н.Е. «Системное программирование. Основы построения трансляторов»
4. Костельцев А.В. «Построение интерпретаторов и компиляторов»
5. Касьянов В.Н., Поттосин И.В. «Методы построения трансляторов»
6. Д. Грис «Конструирование компиляторов для цифровых вычислительных машин»
7. Ахо А., Ульман Дж. «Теория синтаксического анализа, перевода и компиляции»
8. Льюис Ф., Розенкранц Д., Смирну Д. «Теоретические основы проектирования компиляторов»

**Приложение А.**

**Листинг программного текста транслятора**

Form1.cs

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading;

using System.Windows.Forms;

namespace WindowsFormsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

int len = 0;

static bool IsEnter = false;

static string ReadString = "";

delegate void AddText(string msg);

delegate void IsVisible(bool msg);

struct snd

{

public List<Stack<string>> IN;

public RichTextBox rtb;

public snd(List<Stack<string>> a, RichTextBox b)

{

IN = a;

rtb = b;

}

};

struct ConstIdent

{

public string Ident;

public string Const;

public ConstIdent(string a, string b)

{

Ident = a;

Const = b;

}

};

struct CaseStruct

{

public int one;

public string two;

public string three;

public CaseStruct(int a, string b, string c)

{

one = a;

two = b;

three = c;

}

};

private void выходToolStripMenuItem\_Click(object sender, EventArgs e)

{

this.Close();

}

private void сохранитьФайлToolStripMenuItem\_Click(object sender, EventArgs e)

{

SaveFileDialog sd = new SaveFileDialog();

sd.DefaultExt = "\*.rt";

sd.Filter = "RT files(\*.rt)|\*.rt";

if (sd.ShowDialog() == DialogResult.OK && sd.FileName.Length > 0)

{

txtCode.SaveFile(sd.FileName);

}

}

private void создатьПроектToolStripMenuItem\_Click(object sender, EventArgs e)

{

txtCode.Text = "";

}

private void открытьФайлToolStripMenuItem\_Click(object sender, EventArgs e)

{

using(OpenFileDialog OpnFDlg = new OpenFileDialog())

{

try

{

OpnFDlg.Filter = "RT files(\*.rt)|\*.rt|All files(\*.\*)|\*.\*";

OpnFDlg.FilterIndex = 1;

OpnFDlg.InitialDirectory = "D:";

if(OpnFDlg.ShowDialog() == DialogResult.OK)

{

StreamReader sr = new StreamReader(OpnFDlg.FileName, Encoding.Default);

string str = sr.ReadToEnd();

sr.Close();

txtCode.Text = str;

}

}

catch(Exception msg)

{

MessageBox.Show(msg.Message);

}

}

}

private void runToolStripMenuItem\_Click(object sender, EventArgs e)

{

Lexer lex = new Lexer(txtCode.Text);

Parser pars = new Parser(lex.TOKENS);

if (pars.DebugMsg == "debug successful")

{

txtMsg.Focus();

Shell shell = new Shell(pars.TokensShell, lex.IDENTIFIES, lex.CONSTANTS);

snd[] snd1 = { new snd(shell.OUT, txtMsg) };

Thread RunThread = new Thread(Run);

RunThread.Start(snd1);

}

txtMsg.Text = pars.DebugMsg+"\n";

}

static void Run(object INI)

{

snd[] snd1 = (snd[])INI;

List<ConstIdent> CI = new List<ConstIdent>();

for (int i = 0; i < snd1[0].IN.Count(); i++)

{

while (snd1[0].IN[i].Count() != 0)

{

switch (snd1[0].IN[i].Peek())

{

case "05":

{

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new IsVisible((s) => snd1[0].rtb.ReadOnly = s), false);

snd1[0].IN[i].Pop();

while (snd1[0].IN[i].Count != 0)

{

int rr = 0;

string b = "";

string a = snd1[0].IN[i].Pop();

while (rr == 0)

{

while (IsEnter == false) { Thread.Sleep(100); }

b = ReadString;

try { int k = Convert.ToInt32(b); IsEnter = false; rr++; }

catch

{

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), "Incorrect type\n");

IsEnter = false;

}

}

CI.Add(new ConstIdent(a, b));

}

IsEnter = false;

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new IsVisible((s) => snd1[0].rtb.ReadOnly = s), true);

break;

}

case "06":

{

snd1[0].IN[i].Pop();

while (snd1[0].IN[i].Count != 0)

{

for (int j = CI.Count - 1; j >= 0; j--)

{

if (snd1[0].IN[i].Peek() == CI[j].Ident)

{

try

{

Convert.ToInt32(CI[j].Const);

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), Convert.ToString(CI[j].Const + "\n"));

}

catch

{

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), "");

}

snd1[0].IN[i].Pop();

break;

}

}

}

break;

}

case "07":

{

Stack<string> s1 = new Stack<string>();

List<CaseStruct> CS = new List<CaseStruct>();

string three;

int case1 = 0;

snd1[0].IN[i].Pop();

snd1[0].IN[i].Pop();

string case2 = snd1[0].IN[i].Pop();

if (snd1[0].IN[i].Peek() == "$")

{

try

{

case1 = Convert.ToInt32(case2);

}

catch

{

try { case1 = Convert.ToInt32(TryFind(case2, CI)); }

catch

{

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), TryFind(case2, CI) + "\n");

}

}

}

else

{

snd1[0].IN[i].Push(case2);

try

{

case1 = Convert.ToInt32(Operation(snd1[0].IN[i], CI));

}

catch

{

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), Operation(snd1[0].IN[i], CI) + "\n");

}

}

snd1[0].IN[i].Pop();

while (snd1[0].IN[i].Count != 0)

{

int one = Convert.ToInt32(snd1[0].IN[i].Pop());

string two = snd1[0].IN[i].Pop();

snd1[0].IN[i].Pop();

three = Operation(snd1[0].IN[i], CI);

CS.Add(new CaseStruct(one, two, three));

snd1[0].IN[i].Pop();

}

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

{

if (case1 == CS[j].one)

{

CI.Add(new ConstIdent(CS[j].two, CS[j].three));

break;

}

}

break;

}

default:

{

if (snd1[0].IN[i].Count() == 4)

{

string a = snd1[0].IN[i].Pop();

snd1[0].IN[i].Pop();

string err = TryFind(snd1[0].IN[i].Pop(), CI);

CI.Add(new ConstIdent(a, err));

snd1[0].IN[i].Pop();

break;

}

else

{

string a = snd1[0].IN[i].Pop();

snd1[0].IN[i].Pop();

string ww = Operation(snd1[0].IN[i], CI);

if (ww == "variable is not set" || ww == "zero division")

if (snd1[0].rtb.InvokeRequired)

snd1[0].rtb.Invoke(new AddText((s) => snd1[0].rtb.AppendText(s)), ww + "\n");

CI.Add(new ConstIdent(a, ww));

snd1[0].IN[i].Pop();

}

break;

}

}

}

}

}

static string TryFind(string a, List<ConstIdent> ci)

{

int ret;

int kol = 0;

while (true)

{

try { ret = Convert.ToInt32(a); return a; }

catch

{

kol = 0;

for (int j = ci.Count - 1; j >= 0; j--)

{

if (ci[j].Ident == a)

{

a = ci[j].Const;

kol++;

break;

}

}

if (kol == 0)

return "variable is not set";

}

}

}

static string Operation(Stack<string> a, List<ConstIdent> CI)

{

Stack<string> s1 = new Stack<string>();

while (a.Peek() != "$")

{

switch (a.Peek())

{

case "+":

{

a.Pop();

int b1, c1;

try

{

b1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

c1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

}

catch

{

return "variable is not set";

}

if (a.Peek() != "$")

s1.Push(Convert.ToString(b1 + c1));

else

return Convert.ToString(b1 + c1);

break;

}

case "-":

{

a.Pop();

int b1, c1;

try

{

b1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

c1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

}

catch

{

return "variable is not set";

}

if (a.Peek() != "$")

s1.Push(Convert.ToString(c1 - b1));

else

return Convert.ToString(c1 - b1);

break;

}

case "/":

{

a.Pop();

int b1, c1;

try

{

b1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

c1 = Convert.ToInt32(TryFind(s1.Pop(), CI));

if (b1 == 0)

return "zero division";

}

catch

{

return "variable is not set";

}

if (a.Peek() != "$")

s1.Push(Convert.ToString(c1 / b1));

else

return Convert.ToString(c1 / b1);

break;

}

default:

{

s1.Push(a.Pop());

if (a.Peek() == "$" && s1.Count == 1)

{

return s1.Pop();

}

break;

}

}

}

return "error";

}

}

}

Shell.cs

using System;

using System.Collections.Generic;

namespace WindowsFormsApplication1

{

class Shell

{

private List<string> IDENTIFIES = new List<string>();

private List<string> CONSTANTS = new List<string>();

private Stack<string> TokensStack = new Stack<string>();

private int CounterIdentifies = 0;

private int CounterConstants = 0;

public List<Stack<string>> OUT = new List<Stack<string>>();

private string OutAdd = "";

private string[] priority = { "14", "13", "15" };

public Shell(Stack<string> TOKENS, List<string> IDENTIFIES, List<string> CONSTANTS)

{

this.TokensStack = TOKENS;

this.IDENTIFIES = IDENTIFIES;

this.CONSTANTS = CONSTANTS;

FindOperations();

}

private void FindOperations()

{

while (TokensStack.Peek() != "03")

{

if (TokensStack.Pop() == "20")

{

CounterIdentifies++;

}

}

while (TokensStack.Count != 0)

{

switch (TokensStack.Peek())

{

case "05":

{

Stack<string> s1 = new Stack<string>();

Stack<string> s2 = new Stack<string>();

s1.Push(TokensStack.Pop());

while (TokensStack.Peek() != "11")

{

if (TokensStack.Pop() == "20")

{

s1.Push(IDENTIFIES[CounterIdentifies++]);

}

}

while (s1.Count != 0)

s2.Push(s1.Pop());

OUT.Add(s2);

break;

}

case "06":

{

Stack<string> s1 = new Stack<string>();

Stack<string> s2 = new Stack<string>();

s1.Push(TokensStack.Pop());

while (TokensStack.Peek() != "11")

{

if (TokensStack.Pop() == "20")

{

s1.Push(IDENTIFIES[CounterIdentifies++]);

}

}

while (s1.Count != 0)

s2.Push(s1.Pop());

OUT.Add(s2);

break;

}

case "07":

{

Stack<string> s1 = new Stack<string>();

Stack<string> s2 = new Stack<string>();

s1.Push(TokensStack.Pop());

s2 = Equal("08");

while (s2.Count != 0)

s1.Push(s2.Pop());

while (TokensStack.Pop() != "09")

{

if (TokensStack.Peek() == "09")

break;

s1.Push(CONSTANTS[CounterConstants++]);

for (int i = 0; i < 4; TokensStack.Pop(), i++) ;

s1.Push(IDENTIFIES[CounterIdentifies++]);

s2 = Equal("11");

while (s2.Count != 0)

s1.Push(s2.Pop());

}

while (s1.Count != 0)

s2.Push(s1.Pop());

OUT.Add(s2);

break;

}

case "12":

{

Stack<string> s1 = new Stack<string>();

Stack<string> s2 = new Stack<string>();

s1.Push(IDENTIFIES[CounterIdentifies++]);

TokensStack.Pop();

s2 = Equal("11");

while (s2.Count != 0)

s1.Push(s2.Pop());

while (s1.Count != 0)

s2.Push(s1.Pop());

OUT.Add(s2);

break;

}

default:

{

TokensStack.Pop();

break;

}

}

}

}

private Stack<string> Equal(string end)

{

Stack<string> s1 = new Stack<string>();

Stack<string> s2 = new Stack<string>();

s1.Push("$");

Stack<string> operations = new Stack<string>();

operations.Push("00");

while (TokensStack.Peek() != end)

{

switch (TokensStack.Peek())

{

case "20": { s1.Push(IDENTIFIES[CounterIdentifies++]); TokensStack.Pop(); break; }

case "21": { s1.Push(CONSTANTS[CounterConstants++]); TokensStack.Pop(); break; }

default:

{

//для унарного минуса

//если стек пуст и мы считываем минус

if (s1.Peek() == "$" && TokensStack.Peek() == "14")

s1.Push("0");

if (operations.Peek() == "16" && TokensStack.Peek() == "14")

s1.Push("0");

if (operations.Peek() == "16" || operations.Peek() == "00")

{

operations.Push(TokensStack.Pop());

break;

}

if (TokensStack.Peek() == "16")

{

operations.Push(TokensStack.Pop());

break;

}

if (TokensStack.Peek() == "17")

{

while (operations.Peek() != "16")

{

s1.Push(BinOp(operations.Pop()));

}

TokensStack.Pop();

operations.Pop();

break;

}

if (Convert.ToInt32(TokensStack.Peek()) > Convert.ToInt32(operations.Peek()))

{

operations.Push(TokensStack.Pop());

}

else

{

while (Convert.ToInt32(TokensStack.Peek()) > Convert.ToInt32(operations.Peek()) || operations.Peek() != "16")

{

if (operations.Peek() == "00")

break;

s1.Push(BinOp(operations.Pop()));

}

}

break;

}

}

}

while (operations.Peek() != "00")

s1.Push(BinOp(operations.Pop()));

s1.Push("$");

while (s1.Count != 0)

s2.Push(s1.Pop());

return s2;

}

private string BinOp(string str)

{

if (str == "13")

return "+";

if (str == "14")

return "-";

return "/";

}

}

}

Parser.cs

using System.Collections.Generic;

namespace WindowsFormsApplication1

{

class Parser

{

private Stack<string> Tokens = new Stack<string>();

private Stack<string> TokensStack = new Stack<string>();

private Stack<string> StatesStack = new Stack<string>();

public string DebugMsg = "";

public Stack<string> TokensShell = new Stack<string>();

public Parser(Stack<string> Tokens)

{

this.Tokens = Tokens;

Parsing();

}

//создаем список состояний

string[,] States = { { "04", "53", "03", "11", "02", "10", "51", "00" },

{ "54", "11", "17", "51", "16", "00", "00", "00" },

{ "54", "11", "09", "56", "08", "58", "00", "00" },

{ "54", "11", "58", "12", "00", "00", "00", "00" },

{ "11", "58", "12", "00", "00", "00", "00", "00" },

{ "57", "55", "10", "00", "00", "00", "00", "00" },

{ "60", "17", "58", "00", "00", "00", "00", "00" } };

//записываем лексемы в стек

private void ToStack()

{

TokensStack.Push("$");

TokensShell.Push("$");

while (Tokens.Count != 0)

{

TokensStack.Push(Tokens.Peek());

TokensShell.Push(Tokens.Pop());

}

}

//парсер

private int Parsing()

{

ToStack();

StatesStack.Push("$");

StatesStack.Push("50");

while ((StatesStack.Count != 0) || (TokensStack.Count != 0))

{

switch (StatesStack.Pop())

{

case "50":

{

switch (TokensStack.Pop())

{

case "01": { int i = 0; while (States[0, i] != "00") StatesStack.Push(States[0, i++]); break; }

default: { DebugMsg = "error: expected \"VAR\""; return 0; }

}

break;

}

case "51":

{

switch (TokensStack.Pop())

{

case "20": { StatesStack.Push("52"); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: expected variable"; return 0; }

}

break;

}

case "52":

{

switch (TokensStack.Pop())

{

case "18": { StatesStack.Push("51"); break; }

case "10": { StatesStack.Pop(); break; }

case "17": { StatesStack.Pop(); break; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "53":

{

switch (TokensStack.Pop())

{

case "05": { int i = 0; while (States[1, i] != "00") StatesStack.Push(States[1, i++]); break; }

case "06": { int i = 0; while (States[1, i] != "00") StatesStack.Push(States[1, i++]); break; }

case "07": { int i = 0; while (States[2, i] != "00") StatesStack.Push(States[2, i++]); break; }

case "20": { int i = 0; while (States[3, i] != "00") StatesStack.Push(States[3, i++]); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "54":

{

switch (TokensStack.Peek())

{

case "05": { StatesStack.Push("53"); break; }

case "06": { StatesStack.Push("53"); break; }

case "07": { StatesStack.Push("53"); break; }

case "20": { StatesStack.Push("53"); break; }

case "04": { StatesStack.Pop(); TokensStack.Pop(); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "55":

{

switch (TokensStack.Pop())

{

case "20": { int i = 0; while (States[4, i] != "00") StatesStack.Push(States[4, i++]); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "56":

{

switch (TokensStack.Pop())

{

case "21": { int i = 0; while (States[5, i] != "00") StatesStack.Push(States[5, i++]); break; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "57":

{

switch (TokensStack.Peek())

{

case "21": { StatesStack.Push("56"); break; }

case "09": { TokensStack.Pop(); break; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "58":

{

switch (TokensStack.Peek())

{

case "14": { TokensStack.Pop(); StatesStack.Push("59"); break; }

case "16": { StatesStack.Push("59"); break; }

case "20": { StatesStack.Push("59"); break; }

case "21": { StatesStack.Push("59"); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: incorrect expression"; return 0; }

}

break;

}

case "59":

{

switch (TokensStack.Pop())

{

case "16": { int i = 0; while (States[6, i] != "00") StatesStack.Push(States[6, i++]); break; }

case "20": { StatesStack.Push("60"); break; }

case "21": { StatesStack.Push("60"); break; }

case "91": { DebugMsg = "error: invalid code"; return 0; }

case "92": { DebugMsg = "error: long variable name"; return 0; }

case "93": { DebugMsg = "error: re variable declaration"; return 0; }

case "94": { DebugMsg = "error: undeclared variable"; return 0; }

case "95": { DebugMsg = "error: incorrect variable name"; return 0; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "60":

{

switch (TokensStack.Pop())

{

case "13": { StatesStack.Push("59"); break; }

case "14": { StatesStack.Push("59"); break; }

case "15": { StatesStack.Push("59"); break; }

case "08": { StatesStack.Pop(); break; }

case "11": { StatesStack.Pop(); break; }

case "17": { StatesStack.Pop(); break; }

default: { DebugMsg = "error: Syntax error"; return 0; }

}

break;

}

case "02":

{

switch (TokensStack.Pop())

{

case "02": { break; }

default: { DebugMsg = "error: expected \"integer\""; return 0; }

}

break;

}

case "03":

{

switch (TokensStack.Pop())

{

case "03": { break; }

default: { DebugMsg = "error: expected \"begin\""; return 0; }

}

break;

}

case "11":

{

switch (TokensStack.Pop())

{

case "11": { break; }

default: { DebugMsg = "error: expected \";\""; return 0; }

}

break;

}

case "12":

{

switch (TokensStack.Pop())

{

case "12": { break; }

default: { DebugMsg = "error: expected \"=\""; return 0; }

}

break;

}

case "10":

{

switch (TokensStack.Pop())

{

case "10": { break; }

default: { DebugMsg = "error: expected \":\""; return 0; }

}

break;

}

case "16":

{

switch (TokensStack.Pop())

{

case "16": { break; }

default: { DebugMsg = "error: expected \"(\""; return 0; }

}

break;

}

case "17":

{

switch (TokensStack.Pop())

{

case "17": { break; }

default: { DebugMsg = "error: expected \")\""; return 0; }

}

break;

}

case "$":

{

switch (TokensStack.Pop())

{

case "$": { break; }

default: { DebugMsg = "error: incorrect completion"; return 0; }

}

break;

}

default: { break; }

}

}

DebugMsg = "debug successful";

return 0;

}

}

}

Lexer.cs

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

namespace WindowsFormsApplication1

{

internal class Lexer

{

private const int IdentLength = 12;

private string Code;

private int FindInt = 0;

public string Error = "";

public int kolError = 0;

private struct KeyWords

{

public string Words;

public string WordKey;

public KeyWords(string a, string b)

{

Words = a;

WordKey = b;

}

}

private KeyWords[] KEYS =

{

new KeyWords("VAR", "01"), new KeyWords("INTEGER", "02"), new KeyWords("BEGIN", "03"),

new KeyWords("END", "04"), new KeyWords("READ", "05"), new KeyWords("WRITE", "06"),

new KeyWords("CASE", "07"), new KeyWords("OF", "08"), new KeyWords("END\_CASE", "09")

};

private KeyWords[] CHARS =

{

new KeyWords(":", "10"), new KeyWords(";", "11"), new KeyWords("=", "12"),

new KeyWords("+", "13"), new KeyWords("-", "14"), new KeyWords("/", "15"),

new KeyWords("(", "16"), new KeyWords(")", "17"), new KeyWords(",", "18")

};

/\*

Недопустимый символ 91

Слишком длинное имя переменной 92

Повторное объявление переменной 93

Необъявленная переменная 94

Неверное имя переменной 95

\*/

//регулярное выражение

private string rgz = @"(?<char>[^:;\(\)\=\+\-\/\s\,]\*)(?<separator>[:;\(\)\=\+\-\/\s\,]{1})?";

//список полученных токенов, идентификаторов, констант

public Stack<string> TOKENS = new Stack<string>();

public List<string> IDENTIFIES = new List<string>();

public List<string> CONSTANTS = new List<string>();

public Lexer()

{

this.Error = "Lexer error.\n";

}

public Lexer(string Code)

{

this.Code = Code;

string pattern = "( )+";

string replacement = " ";

if (this.Code.Length == 0)

{

this.Error += "Error: No entry.\n";

}

else

{

//удаление лишних пробелов, перевод в вверхний регистр

Regex rgx = new Regex(pattern);

this.Code = rgx.Replace(this.Code, replacement);

this.Code = this.Code.ToUpper();

}

LexerOut();

}

private void LexerOut()

{

Match match = Regex.Match(this.Code, this.rgz);

while (match.Success)

{

if (match.Groups["char"].Length > 0)

if (!IsKeyword(match.Groups["char"].ToString(), KEYS))

if (!IsIdenOrConst(match.Groups["char"].ToString(),

"[^0-9]+", "21", CONSTANTS, int.MaxValue.ToString().Length))

{

IDENTIFIES.Add(match.Groups["char"].ToString());

AddError(match.Groups["char"].ToString());

}

if ((match.Groups["separator"].ToString() != " ") && (match.Groups["separator"].ToString() != "\n"))

IsKeyword(match.Groups["separator"].ToString(), CHARS);

match = match.NextMatch();

}

}

private bool IsKeyword(string word, KeyWords[] kwd)

{

bool TOF = false;

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

{

if (kwd[i].Words == word)

{

TOKENS.Push(kwd[i].WordKey);

TOF = true;

if (word == "BEGIN")

FindInt = IDENTIFIES.Count;

break;

}

}

return TOF;

}

private bool IsIdenOrConst(string word, string regular, string ID, List<string> lst, int len)

{

bool TOF = true;

if (Regex.Match(word, regular).Success)

TOF = false;

else

{

TOKENS.Push(ID);

lst.Add(word);

}

return TOF;

}

private void AddError(string word)

{

int kol = 0;

for (int i = 0; i < 5; i++)

{

switch (i)

{

case 0: { if (Regex.Match(word, @"[^A-Z0-9:;\(\)\=\+\-\/\s\,]").Success) { kol++; TOKENS.Push("91"); } break; }

case 1: { if (word.Length > IdentLength) { kol++; TOKENS.Push("92"); } break; }

case 2: { int k2 = 0; if (FindInt == 0) for (int j = 0; j < IDENTIFIES.Count; j++) if (word == IDENTIFIES[j]) k2++; if (k2 > 1) { kol++; TOKENS.Push("93"); } break; }

case 3: { int k2 = 1; if (FindInt != 0) { k2 = 0; for (int j = 0; j < FindInt; j++) if (word == IDENTIFIES[j]) k2++; } if (k2 == 0) { kol++; TOKENS.Push("94"); } break; }

case 4: { if (Regex.Match(word, "[^A-Z]").Success) { kol++; TOKENS.Push("95"); } break; }

}

}

if (kol == 0)

TOKENS.Push("20");

}

}

}

**Приложение Б.**

**Результаты тестирования**

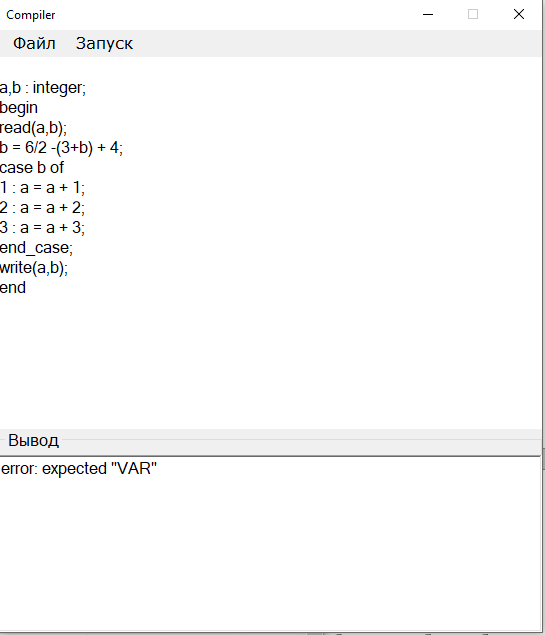


Рисунок 3. Вывод сообщения об ошибке

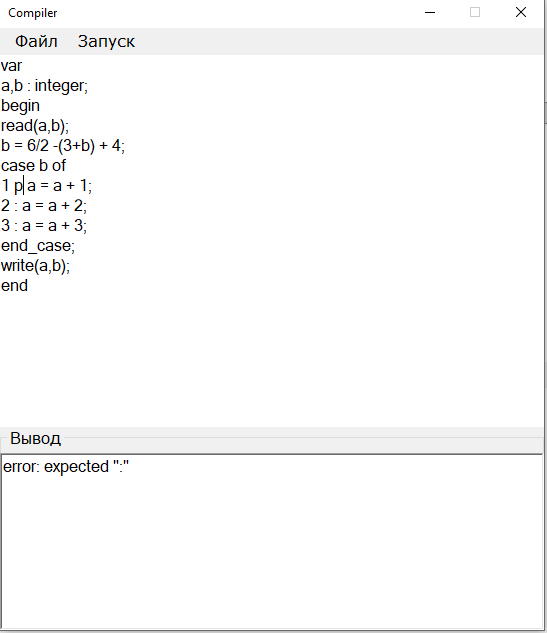


Рисунок 4. Вывод сообщения об ошибке

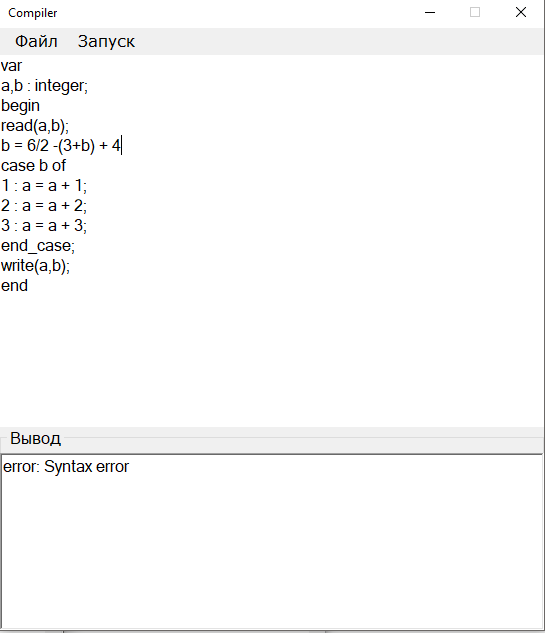


Рисунок 5. Вывод сообщения об ошибке

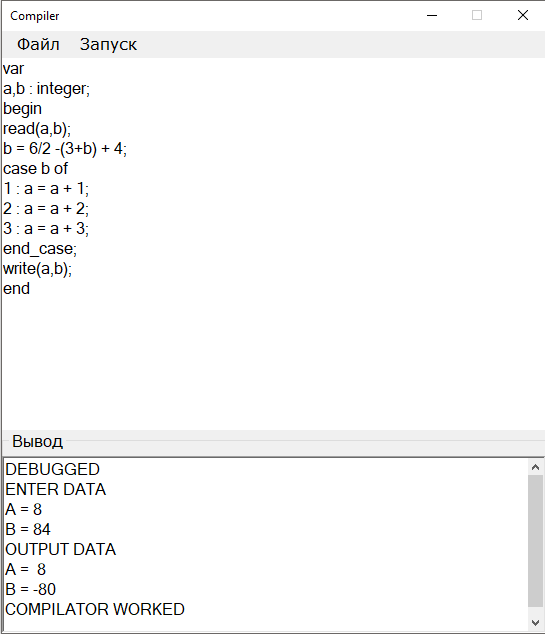


Рисунок 6. Рабочая программа