МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра автоматизированных систем управления



ОТЧЁТ

по КУРСОВОЙ РАБОТЕ

**«***Объявление комплексного числа с инициализацией на языке Python***»**

по дисциплине: **«***Теория формальных языков и компиляторов***»**

Выполнил:Проверил:

*Киреенко М.А.* *д.т.н., профессор*

Студент гр. «*АВТ-113*» *Шорников Юрий Владимирович*

«\_\_\_» \_\_\_\_\_\_ 20\_\_г.«\_\_\_» \_\_\_\_\_\_ 20\_\_ г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск 2024

**РЕФЕРАТ**

Отчет 38 с., 14 рис., 8 источн., 3 прилож.

ЯЗЫКОВОЙ ПРОЦЕССОР, КОМПИЛЯТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, АВТОМАТНАЯ ГРАММАТИКА, ГРАФ АВТОМАТНОЙ ГРАММАТИКИ, ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК, МЕТОД АЙРОНСА

Цель работы – выполнить программную реализацию алгоритма синтаксического анализа комплексного числа с инициализацией на языке Python.

В результате проектирования был написан синтаксический анализатор (парсер) для комплексного числа с инициализацией на языке Python.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc69940111)

[1 Постановка задачи 5](#_Toc69940112)

[2 Разработка грамматики 6](#_Toc69940113)

[3 Классификация грамматики 7](#_Toc69940114)

[4 Метод анализа 8](#_Toc69940115)

[5 Диагностика и нейтрализация синтаксических ошибок 9](#_Toc69940116)

[5.1 Метод Айронса 9](#_Toc69940117)

[5.2 Метод Айронса для автоматной грамматики 9](#_Toc69940118)

[6 Тестовые примеры 11](#_Toc69940119)

[7 Листинг программы 14](#_Toc69940120)

[ЗАКЛЮЧЕНИЕ 15](#_Toc69940121)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 16](#_Toc69940122)

[Приложение А – Справка (руководство пользователя) 17](#_Toc69940123)

[Меню текстового редактора 17](#_Toc69940125)

[Пункт "Файл" меню текстового редактора 17](#_Toc69940126)

[Пункт "Правка" меню текстового редактора 17](#_Toc69940127)

[Пункт "Текст" меню текстового редактора 18](#_Toc69940128)

[Пункт "Пуск" меню текстового редактора 19](#_Toc69940129)

[Пункт "Справка" меню текстового редактора 19](#_Toc69940130)

[Пункт "Локализация" меню текстового редактора 20](#_Toc69940131)

[Пункт "Вид" меню текстового редактора 20](#_Toc69940132)

[Панель инструментов текстового редактора 20](#_Toc69940133)

[Дополнительные возможности текстового редактора 21](#_Toc69940134)

[Приложение Б – Информация о программе 23](#_Toc69940135)

[Приложение В – Листинг программы 25](#_Toc69940137)

# ВВЕДЕНИЕ

Цель курсовой работы – выполнить программную реализацию алгоритма объявления комплексного числа с инициализацией на языке Python.

Курсовая работа содержит следующие разделы:

* Постановка задачи;
* Грамматика;
* Классификация грамматики;
* Метод анализа;
* Диагностика и нейтрализация ошибок;
* Тестовый пример;
* Список литературы;
* Исходный код программы.

## 1 Постановка задачи

Комплексные числа представляют собой пару значений типа int или float, и имеют вид: <реальная часть>+<мнимая часть>

Для описания комплекса в языке Python используется служебное слово "complex".

Формат записи:

"имя\_идентификатора = complex(значение1,значение2)".

Примеры:

1. Целая десятичная константа – любое десятичное число без десятичной точки со знаком или без него: "z2 = complex(6,8.3)" В этом случае – это число 6.
2. Вещественная константа с фиксированной точкой – действительное число представленное в виде целой и дробной частей разделенных точкой: "z2 = complex(6,8.3)". В этом случае – это число 8.3.

В связи с разработанной автоматной грамматикой G[<КЧ>]  синтаксический анализатор (парсер) комплексных чисел будет считать верными следующие записи констант:

1. " z2 = complex(6,8.3) "
2. "z3 = complex(-1,+2.5) "
3. "z4 = complex(+5,-2.8) "

Справка (руководство пользователя) представлена в Приложении А. Информация о программе представлена в Приложении Б.

## 2 Разработка грамматики

Определим грамматику комплексного числа языка Python G[<КЧ>] в нотации Хомского с продукциями P:

1. <КЧ> -> letter COMPLEXREM
2. COMPLEXREM -> (letter | \_ | digit) COMPLEXREM | = ASSIGNTMENT
3. ASSIGNTMENT -> complex OPEN
4. OPEN -> ( REALPART
5. REALPART -> [+ | -] INTREALPART
6. INTREALPART -> digit INTREALPARTREM
7. INTREALPARTREM -> digit INTREALPARTREM | . DECIMALREALPART | , IMAGINARYPART
8. DECIMALREALPART -> digit DECIMALREALPARTREM
9. DECIMALREALPARTREM -> digit DECIMALREALPARTREM | , IMAGINARYPART
10. IMAGINARYPART -> [+ | -] INTIMAGINARYPART
11. INTIMAGINARYPART -> digit INTIMAGINARYPARTREM
12. INTIMAGINARYPARTREM -> digit INTIMAGINARYPARTREM | . DECIMALIMAGINARYPART |  CLOSE
13. DECIMALIMAGINARYPART -> digit DECIMALIMAGINARYPARTREM
14. DECIMALIMAGINARYPARTREM -> digit DECIMALIMAGINARYPARTREM |  CLOSE
15. CLOSE -> )

* letter → ‘a’ | ‘b’ | … | ‘z’ | ‘A’ | ‘B’ | … | ‘Z’
* digit → ‘0’ | ‘1’ | … | ‘9’

Следуя введенному формальному определению грамматики, представим G[<КЧ>] ее составляющими:

* Z = <КЧ>;
* VT = { ‘a’…’z’, ‘A’…’Z’, '=', 'complex', '(', ')', '+',, '-', '0'...'9', '.', ‘\_’, ‘'‘ }
* VN = { <КЧ>, COMPLEXREM, ASSIGNTMENT, OPEN, REALPART, INTREALPART, INTREALPARTREM, DECIMALREALPART, IMAGINARYPART, DECIMALREALPARTREM, INTIMAGINARYPART, INTIMAGINARYPARTREM, DECIMALIMAGINARYPART, DECIMALIMAGINARYPARTREM, CLOSE}

## 3 Классификация грамматики

Согласно классификации Хомского, грамматика G[<КЧ>]  является автоматной.

Правила (1)-(14) относятся к классу праворекурсивных продукций (A → aB | a | ε):

1. <КЧ> -> letter COMPLEXREM
2. COMPLEXREM -> (letter | \_ | digit) COMPLEXREM | = ASSIGNTMENT
3. ASSIGNTMENT -> complex OPEN
4. OPEN -> ( REALPART
5. REALPART -> [+ | -] INTREALPART
6. INTREALPART -> INTREALPARTREM
7. INTREALPARTREM -> digit INTREALPARTREM | . DECIMALREALPART | , IMAGINARYPART
8. DECIMALREALPART -> digit DECIMALREALPARTREM
9. DECIMALREALPARTREM -> digit DECIMALREALPARTREM | , IMAGINARYPART
10. IMAGINARYPART -> [+ | -] INTIMAGINARYPART
11. INTIMAGINARYPART -> INTIMAGINARYPARTREM
12. INTIMAGINARYPARTREM -> digit INTIMAGINARYPARTREM | . DECIMALIMAGINARYPART |  CLOSE
13. DECIMALIMAGINARYPART -> digit DECIMALIMAGINARYPARTREM
14. DECIMALIMAGINARYPARTREM -> digit DECIMALIMAGINARYPARTREM |  CLOSE

Правило (15) является простым: 15. CLOSE → )

Данная грамматика содержит только праворекурсивные продукции, и, следовательно, является полностью автоматной.

## 4 Метод анализа

Грамматика G[<КЧ>] является автоматной.

Правила (1) – (15) для G[<КЧ>] реализованы на графе (см. рисунок 1).

Сплошные стрелки на графе характеризуют синтаксически верный разбор; пунктирные символизируют состояние ошибки (ERROR); дуга λ и непомеченные дуги предполагают любой терминальный символ, отличный от указанного из соответствующего узла.

Состояние К символизирует успешное завершение разбора.

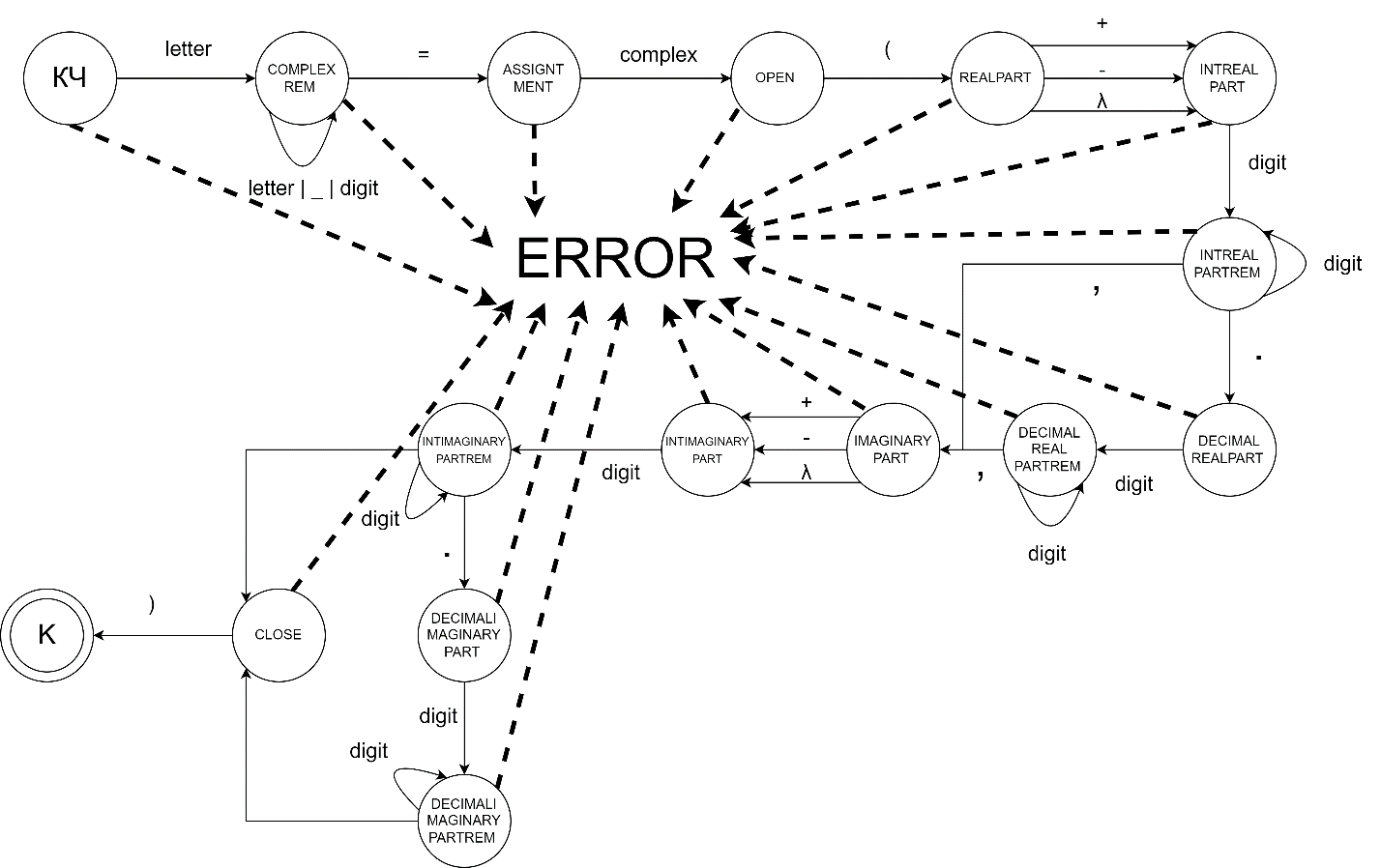


Рисунок 1 – Граф G[<КЧ>]

## 5 Диагностика и нейтрализация синтаксических ошибок

### Метод Айронса — это метод синтаксического анализа, используемый для обработки и восстановления после синтаксических ошибок во время разбора программного кода или других структурированных данных. Основная идея метода заключается в определении и корректировке ошибочных символов во входной последовательности, что позволяет продолжить процесс разбора без остановки.

### 5.1 Метод Айронса

Суть метода Айронса заключается в следующем:

1. **Определение недостроенных кустов дерева разбора:** когда анализатор сталкивается с неожиданным символом, он сначала идентифицирует все текущие недостроенные участки в дереве разбора. Эти участки представляют собой части дерева, которые были начаты, но не завершены из-за встречи с ошибочным символом.
2. **Формирование множества ожидаемых символов:** для каждого недостроенного куста формируется множество символов, которые могли бы быть правильно вставлены в данном контексте. Это множество символов (L) содержит все возможные символы, которые могли бы логически следовать за данным участком дерева согласно правилам грамматики.
3. **Удаление ошибочных символов:** анализатор последовательно удаляет символы из входной цепочки, начиная с ошибочного символа, до тех пор, пока не найдет символ, который может быть выведен из одного из символов в множестве L. Этот процесс помогает синхронизировать входную цепочку с ожиданиями парсера по структуре данных.
4. **Определение причины ошибки:** после того как найден подходящий символ, определяется, к какому из недостроенных кустов он может относиться. Это помогает локализовать ошибку в конкретной части дерева разбора.
5. **Продолжение разбора:** после корректировки входной последовательности и определения подходящего места для продолжения, разбор входных данных продолжается, уже с учётом внесенных исправлений.

### 5.2 Метод Айронса для автоматной грамматики

В контексте автоматной грамматики, где каждое состояние автомата строго определено и ограничено, метод Айронса адаптируется так, чтобы учитывать специфику одного недостроенного куста. Когда возникает синтаксическая ошибка, весь процесс фокусируется на этом единственном недостроенном кусте, что упрощает процесс восстановления:

* **Единый недостроенный куст:** В случае ошибки весь процесс восстановления центрируется вокруг этого куста, упрощая логику поиска и исправления ошибочных символов.
* **Простое удаление символов:** Процесс корректировки ошибок упрощается до удаления каждого следующего символа до тех пор, пока не будет найден допустимый символ, который может продолжить разбор по правилам автоматной грамматики.

Такой подход позволяет эффективно и точно восстанавливать разбор после возникновения синтаксических ошибок в автоматных грамматиках, обеспечивая стабильную работу синтаксического анализатора.

## 6 Тестовые примеры

На рисунках 4-9 представлены тестовые примеры запуска разработанного синтаксического анализатора комплексного числа с инициализацией на языке Python.

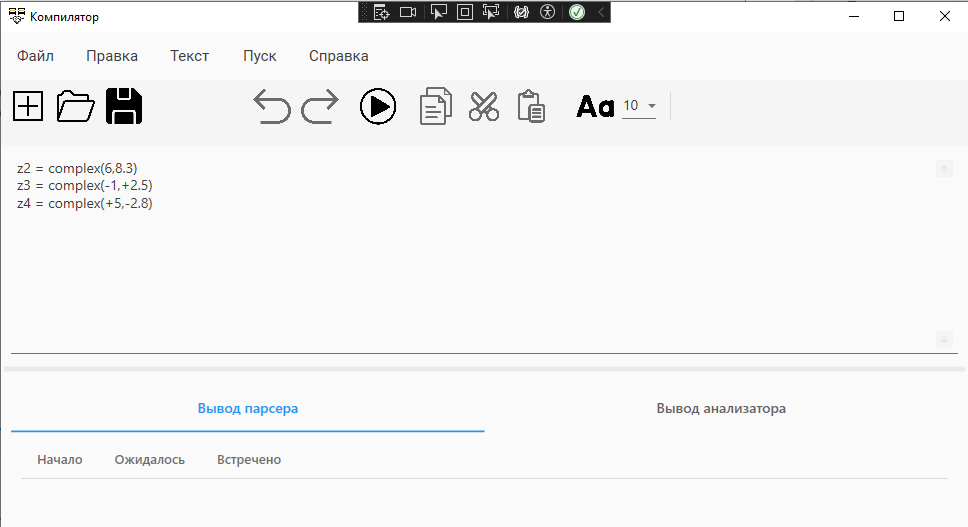


Рисунок 2 – Тестовый пример 1 (Ввод корректен)

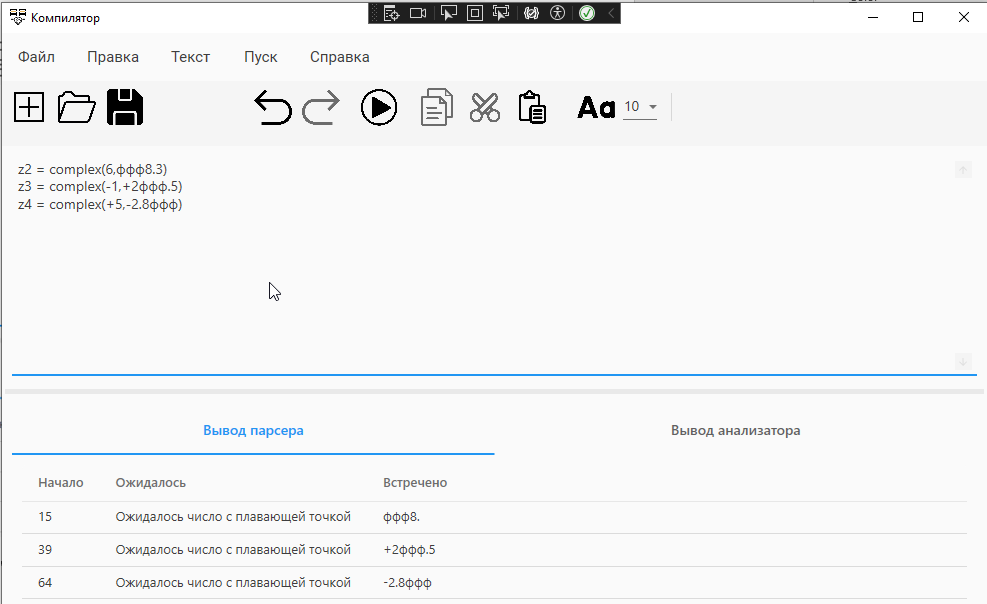


Рисунок 3 – Тестовый пример 2

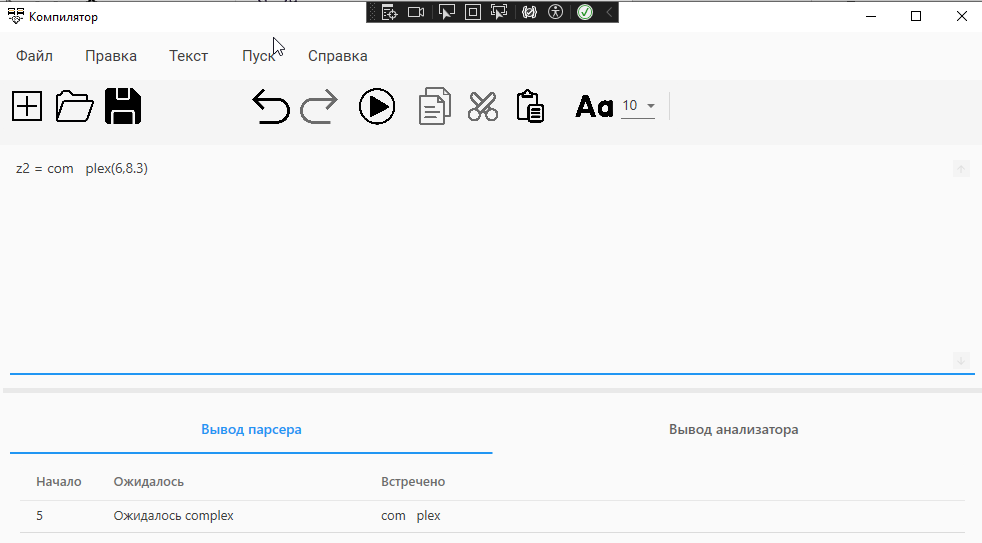


Рисунок 4 – Тестовый пример 3

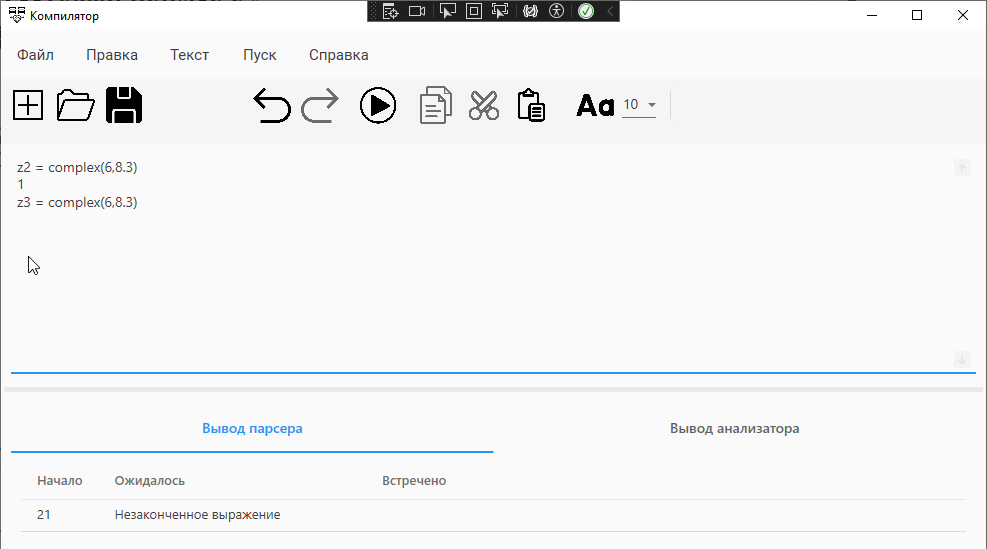


Рисунок 5 – Тестовый пример 4

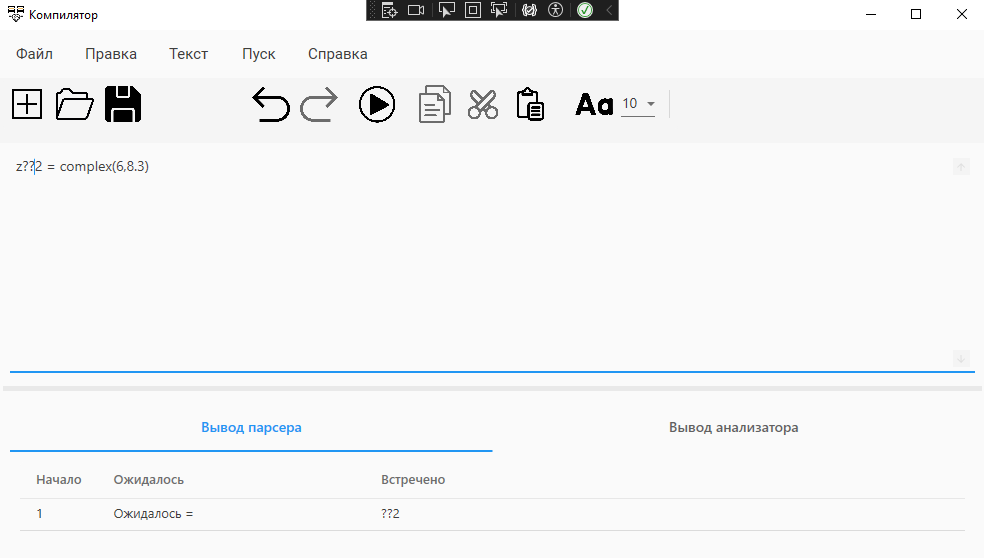


Рисунок 6 – Тестовый пример 5

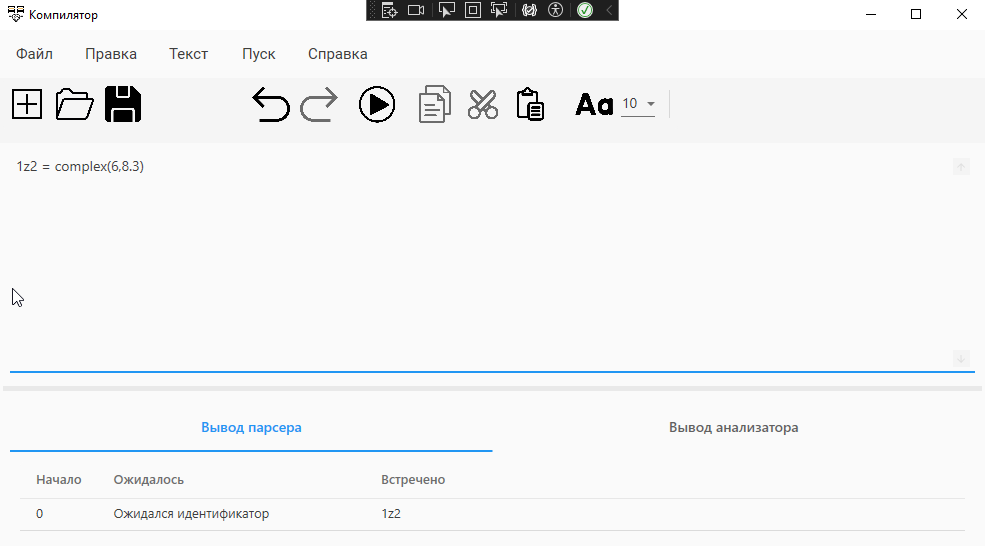


Рисунок 7 – Тестовый пример 6

## 7 Листинг программы

Листинг программной части разработанного синтаксического анализатора комплексного числа с инициализацией на языке Python представлен в приложении В.

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был написан синтаксический анализатор (парсер) для комплексного числа с инициализацией на языке Python. Была определена грамматика комплексного числа с инициализацией на языке Python G[<КЧ>] в нотации Хомского. Согласно классификации Хомского, грамматика G[<КЧ>] является автоматной. Продукции P разработанной грамматики G[<КЧ>] были реализованы на графе. Была реализована нейтрализация синтаксических ошибок методом Айронса. Алгоритм был сведен к последовательному удалению следующего символа во входной цепочке до тех пор, пока следующий символ не окажется одним из допустимых в данный момент разбора.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Шорников Ю.В. Теория и практика языковых процессоров : учеб. пособие / Ю.В. Шорников. – Новосибирск: Изд-во НГТУ, 2004.
2. Gries D. Designing Compilers for Digital Computers. New York, Jhon Wiley, 1971. 493 p.
3. Теория формальных языков и компиляторов [Электронный ресурс] / Электрон. дан. URL: https://dispace.edu.nstu.ru/didesk/course/show/8594, свободный. Яз.рус. (дата обращения 10.03.2024).
4. Грис Д. Конструирование компиляторов для цифровых вычислительных машин / Д. Грис ; пер. с. англ. Е. Б. Докшицкой, Л. А. Зелениной, Л. Б. Морозовой, В. С. Штаркмана, под ред. Ю. М. Баяковского, Вс. С. Штаркмана. - М., 1975. - 544 с. : табл., схемы
5. Ахо А. В. Компиляторы : Принципы, технологии, инструменты / А. Ахо, Р. Сети, Д. Ульман. - М., 2003. - 768 с.
6. Малявко, А. А. Формальные языки и компиляторы : учебник / Малявко А. А. - Новосибирск : Изд-во НГТУ, 2014. - 431 с. (Серия "Учебники НГТУ")
7. Свердлов С. З. Языки программирования и методы трансляции: Учебное пособие. — 2е изд., испр. — СПб.: Издательство «Лань», 2019. — 564 с.: ил. — (Учебники для вузов. Специальная литература).
8. Мозговой М. В. Классика программирования : алгоритмы, языки, автоматы, компиляторы : практический подход / Мозговой М. В. - СПб., 2006. - 320 с. : ил.

# Приложение А

# Справка (руководство пользователя)

## Меню текстового редактора

### Пункт "Файл" меню текстового редактора

В пункте "Файл" меню текстового редактора реализован следующий функционал (см. рисунок А.1):

* Создание документа
* Открытие документа
* Сохранение текущих изменений в документе
* Сохранение документа в новый файл
* Выход из текстового редактора

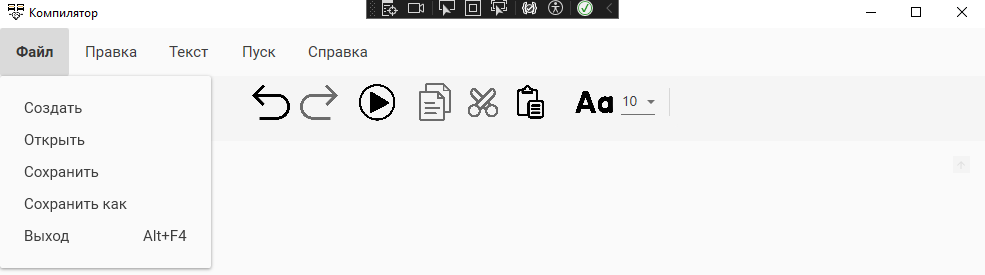
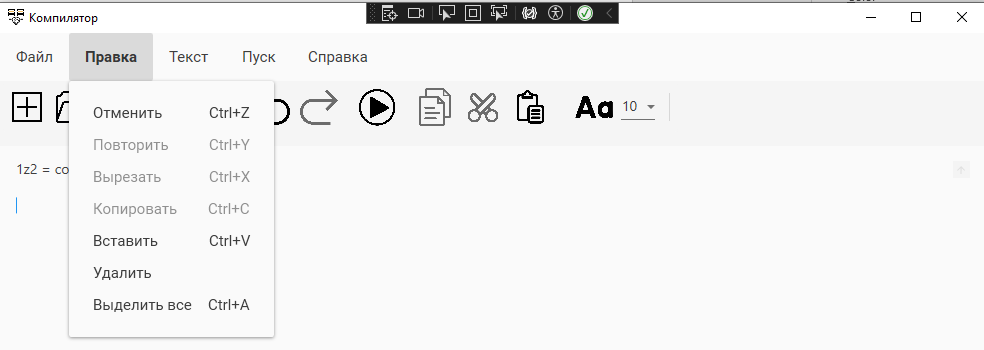


Рисунок А.1 – Пункт "Файл" меню

### Пункт "Правка" меню текстового редактора

В пункте "Правка" меню текстового редактора реализован следующий функционал (см. рисунок А.2):

* Отмена изменений
* Повтор последнего изменения
* Вырезать текстовый фрагмент
* Копировать текстовый фрагмент
* Вставить текстовый фрагмент
* Удалить текстовый фрагмент
* Выделить все содержимое документа

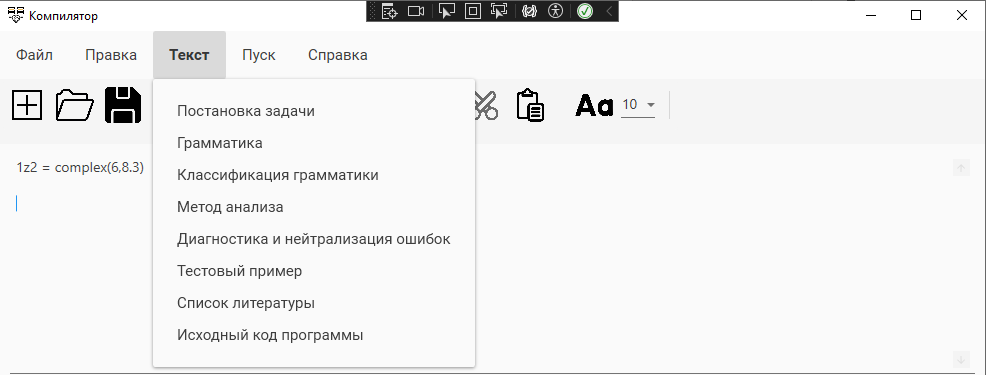
  
Рисунок А.2 – Пункт "Правка" меню

### Пункт "Текст" меню текстового редактора

При вызове команд этого меню должны открываться окна с соответствующей информацией по курсовой работе "Объявление комплексного числа с инициализацией на языке Python".

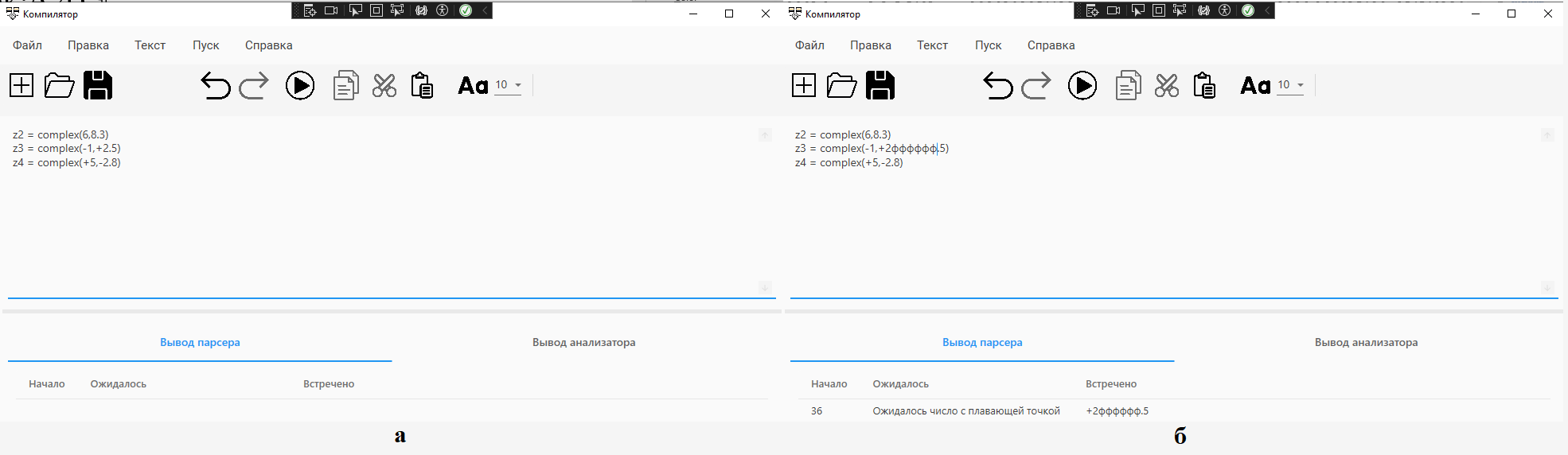
Пункт меню "Текст" содержит следующую информацию (см. рисунок А.3):

* Постановка задачи
* Грамматика
* Классификация грамматики
* Метод анализа
* Диагностика и нейтрализация ошибок
* Тестовый пример
* Список литературы
* Исходный код программы

  
Рисунок А.3 – Пункт "Текст" меню

### Пункт "Пуск" меню текстового редактора

При нажатии на пункт "Пуск" происходит запуск синтаксического анализатора текста (см. рисунок А.4).

  
Рисунок А.4 – Пункт "Пуск" меню

### Пункт "Справка" меню текстового редактора

Приложение имеет справочную систему, запускаемую командой «Вызов справки» (см. рисунок А.5).

Справка содержит описание всех реализованных функций меню.(см. рисунок А.6)

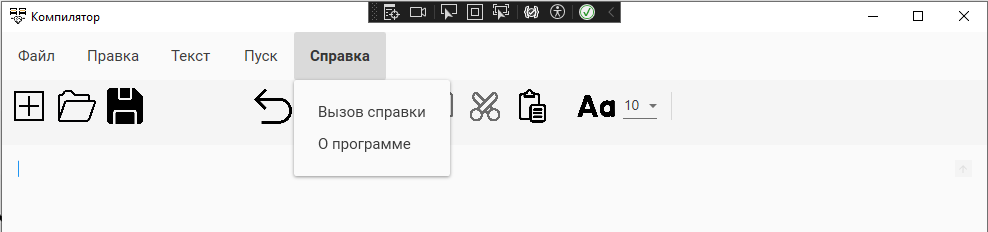
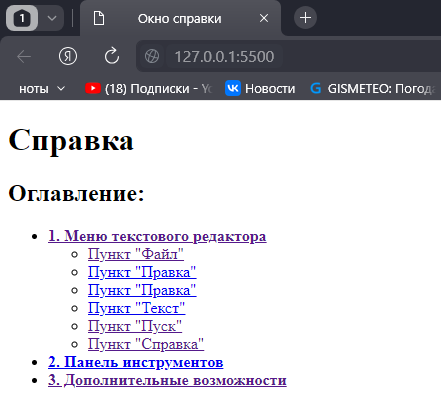
  
Рисунок А.5 – Пункт "Справка" меню  


Рисунок А.6 – Справочная система приложения

## Панель инструментов текстового редактора

Панель инструментов содержит кнопки вызова часто используемых пунктов меню:

* Создание документа (см. рисунок А.7, а)
* Открытие документа (см. рисунок А.7, б)
* Сохранение текущих изменений в документе (см. рисунок А.7, в)
* Отмена изменений (см. рисунок А.7,г)
* Повтор последнего изменения (см. рисунок А.7, д)
* Запуск синтаксического анализатора (см. рисунок А.7, е)
* Копировать текстовый фрагмент (см. рисунок А. 7, ж)
* Вырезать текстовый фрагмент (см. рисунок А.7, з)
* Вставить текстовый фрагмент (см. рисунок А.7, и)
* Изменение размера шрифта (см. рисунок А.7, к)

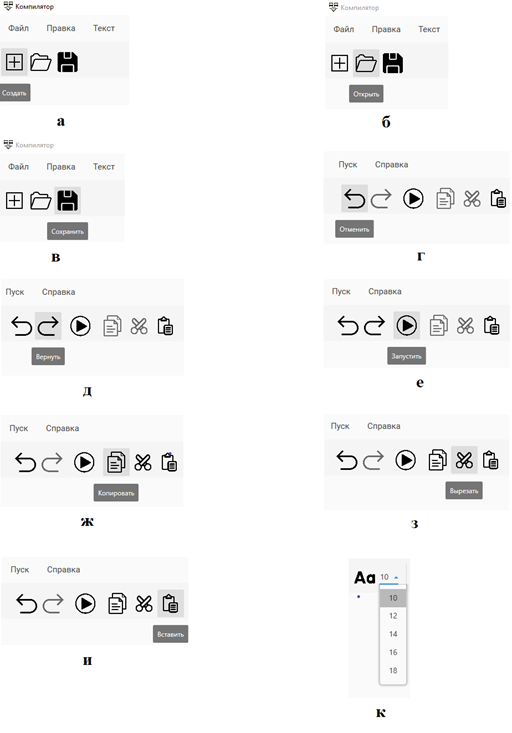


Рисунок А.7 – Панель инструментов

## Дополнительные возможности текстового редактора

1. Интерфейс позволяет открывать файл при перетаскивании его иконки в область редактирования.
2. Изменение размеров текста в окне редактирования и окне вывода результатов.

# Приложение Б

# Информация о программе

Программа написана в рамках лабораторных работы по дисциплине "Теория формальных языков и компиляторов".

Программа доработана в рамках курсовой работы.

Техническое задание:

Разработать приложение – текстовый редактор, дополненный функциями языкового процессора.

Приложение имеет графический интерфейс пользователя.

Язык реализации: C#.

Текстовый редактор имеет следующие элементы:

1. Основное меню программы;
   * Пункт меню "Текст";

При вызове команд этого меню должны открываться окна с соответствующей информацией по курсовой работе «Объявление комплексного числа с инициализацией на языке Python»

1. Панель инструментов;

Панель инструментов содержит кнопки вызова часто используемых пунктов меню:

* + Файл – Создать, Открыть, Сохранить;
  + Правка – Отменить, Повторить, Вырезать, Копировать, Вставить;
  + Пуск;

Команда «Пуск» предназначена для запуска анализатора текста.

Приложение имеет справочную систему, запускаемую командой «Вызов справки».

Cправка содержит описание всех реализованных функций меню.

1. Окно/область ввода/редактирования текста;

Область редактирования представляет текстовый редактор.

Команды меню "Файл" и "Правка" работают с содержимым этой области.

1. Окно/область отображения результатов работы языкового процессора (в этой области ввод текста запрещен).

В область отображения результатов выводятся сообщения и результаты работы языкового процессора.

Интерфейс содержит дополнительные элементы и возможности:

1. Открытие файла при перетаскивании иконки в окно программы.
2. Изменение размеров текста в окне редактирования и окне вывода результатов.

# Приложение В

# Листинг программы

|  |
| --- |
| AssignmentState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class AssignmentState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0 && states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.Complex)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено =");  }  break;  }  if (token.Type != TokenType.Assignment)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось =");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| CloseParenthesisState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class CloseParenthesisState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type != TokenType.CloseParenthesis)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(token);  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось )");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| CommaState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class CommaState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.DoubleLiteral)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено ,");  }  break;  }  if (token.Type != TokenType.Comma)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(token);  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось ,");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| ComplexState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class ComplexState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0 && states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.OpenParenthesis)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено complex");  }  break;  }  if (token.Type != TokenType.Complex)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось complex");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| IdentifierState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class IdentifierState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.Assignment)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущен идентификатор");  }  break;  }  if (token.Type != TokenType.Identifier)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидался идентификатор");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| ImaginaryPartState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class ImaginaryPartState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.CloseParenthesis)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено число с плавающей точкой");  }  break;  }  if (token.Type != TokenType.DoubleLiteral && token.Type != TokenType.IntegerLiteral)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось число с плавающей точкой");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| IParserState.cs |
| using System.Collections.Generic;  namespace CompilerDemo.Model.Parser.States  {  internal interface IParserState  {  void Parse(Parser parser, List<Token> tokens, List<IParserState> states);  }  } |

|  |
| --- |
| OpenParenthesisState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class OpenParenthesisState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.DoubleLiteral)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено (");  }  break;  }  if (token.Type != TokenType.OpenParenthesis)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось (");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| RealPartState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class RealPartState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type == TokenType.Comma)  {  if (token == tokens.First() && errorBuffer.Count == 0)  {  ParserUtils.CreateError(parser, token.StartPos, "Пропущено число с плавающей точкой");  }  break;  }  if (token.Type != TokenType.DoubleLiteral && token.Type != TokenType.IntegerLiteral)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось число с плавающей точкой");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| SemicolonState.cs |
| using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser.States  {  internal class SemicolonState : IParserState  {  public void Parse(Parser parser, List<Token> tokens, List<IParserState> states)  {  if (ParserUtils.TrimWhitespaceTokens(ref tokens) == false || states.Count == 0)  {  return;  }  List<Token> tail = new List<Token>(tokens);  List<Token> errorBuffer = new List<Token>();  foreach (Token token in tail.ToList())  {  if (token.Type != TokenType.Semicolon)  {  errorBuffer.Add(token);  tail.Remove(token);  }  else  {  tail.Remove(tail.First());  break;  }  }  states = states.Skip(1).ToList();  if (tail.Count > 0)  {  ParserUtils.CreateErrorFromBuffer(parser, errorBuffer, "Ожидалось ;");  states.FirstOrDefault()?.Parse(parser, tail, states);  return;  }  states.FirstOrDefault()?.Parse(parser, tokens, states);  }  }  } |

|  |
| --- |
| ParseError.cs |
| using System.Collections.Generic;  namespace CompilerDemo.Model.Parser  {  internal class ParseError  {  private List<Token> errors;  public string Actual { get; private set; }  public string Expected { get; private set; }  public int Pos { get; private set; }  public ParseError(int pos, string actual, string expected)  {  Pos = pos;  Actual = actual;  Expected = expected;  }  public ParseError(List<Token> errors, string expected)  {  this.errors = errors;  Expected = expected;  }  }  } |

|  |
| --- |
| Parser.cs |
| using CompilerDemo.Model.Parser.States;  using System.Collections.Generic;  using System.Linq;  namespace CompilerDemo.Model.Parser  {  internal class Parser  {  private List<ParseError> Errors { get; set; } = new List<ParseError>();    public List<ParseError> Parse(List<Token> tokens)  {  List<IParserState> States = new List<IParserState>()  {  new IdentifierState(),  new AssignmentState(),  new ComplexState(),  new OpenParenthesisState(),  new RealPartState(),  new CommaState(),  new ImaginaryPartState(),  new CloseParenthesisState(),  new SemicolonState(),  };  Errors.Clear();  List<Token> line = tokens.TakeWhile(t => t.Type != TokenType.Newline).ToList();  tokens = tokens.SkipWhile(t => t.Type != TokenType.Newline).ToList();  tokens = tokens.SkipWhile(t => t.Type == TokenType.Newline).ToList();  while (line.Count > 0)  {  States.First().Parse(this, line, States.ToList());  if (line.Last().Type != TokenType.CloseParenthesis || line.Count < 8)  {  ParserUtils.CreateError(this, line.Last().EndPos, "Незаконченное выражение");  }  line = tokens.TakeWhile(t => t.Type != TokenType.Newline).ToList();  tokens = tokens.SkipWhile(t => t.Type != TokenType.Newline).ToList();  tokens = tokens.SkipWhile(t => t.Type == TokenType.Newline).ToList();  }  return Errors;  }  public void AddError(ParseError error)  {  Errors.Add(error);  }  }  } |

|  |
| --- |
| ParserUtils.cs |
| using System.Collections.Generic;  using System.Linq;  using System.Text;  namespace CompilerDemo.Model.Parser  {  internal static class ParserUtils  {  /// <summary>  /// Returns false if tokens size was 0 or became 0 after trim;  /// </summary>  /// <param name="tokens"></param>  /// <returns></returns>  public static bool TrimWhitespaceTokens(ref List<Token> tokens)  {  if (tokens.Count == 0) return false;  foreach (Token token in tokens.ToList())  {  if (string.IsNullOrWhiteSpace(token.RawToken))  {  tokens.Remove(token);  }  else  {  break;  }  }  if (tokens.Count == 0) return false;  return true;  }  public static string ComposeExpectedString(List<Token> errors)  {  StringBuilder sb = new StringBuilder();  foreach (Token token in errors)  {  sb.Append(token.RawToken);  }  return sb.ToString();  }  public static void CreateErrorFromBuffer(Parser parser, List<Token> errorBuffer, string expected)  {  if (errorBuffer.Count > 0)  {  ParseError error = new ParseError(errorBuffer.First().StartPos, ComposeExpectedString(errorBuffer), expected);  parser.AddError(error);  }  }  public static void CreateError(Parser parser, int pos, string expected)  {  ParseError error = new ParseError(pos, "", expected);  parser.AddError(error);  }  }  } |

|  |
| --- |
| Lexer.cs |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  namespace CompilerDemo.Model  {  internal class Lexer  {  public IEnumerable<Token> Scan(string code)  {  if (code.Length == 0)  {  return Enumerable.Empty<Token>();  }  List<Token> tokens = new List<Token>();  int position = 0;  code = code.Replace("\t", "").Replace("\r", "");  do  {  string rawToken = ParseToken(code, position);  tokens.Add(new Token(rawToken, position));  position += rawToken.Length;  } while (position < code.Length);  List<Token> resultTokens = new List<Token>();  for (int i = 0; i < tokens.Count-1; i++)  {  if ((tokens[i].Type == TokenType.Plus || tokens[i].Type == TokenType.Minus)  && (tokens[i+1].Type == TokenType.DoubleLiteral || tokens[i + 1].Type == TokenType.IntegerLiteral))  {  string rawNumber = tokens[i].RawToken + tokens[i+1].RawToken;  Token numberToken = new Token(rawNumber, tokens[i].StartPos);  resultTokens.Add(numberToken);  i++;  }  else  {  resultTokens.Add(tokens[i]);  }  }  resultTokens.Add(tokens.Last());  return resultTokens;  }  private string ParseToken(string code, int position)  {  char symbol = code[position];  string allowedIdentifierSymbols = "\_:<>";  if (char.IsWhiteSpace(symbol))  {  return symbol.ToString();  }  if (char.IsLetter(symbol) || symbol == '\_')  {  return Parse(code, position, (c) => !char.IsLetterOrDigit(c) && !allowedIdentifierSymbols.Contains(c));  }  if (char.IsDigit(symbol))  {  return Parse(code, position, (c) => !char.IsLetterOrDigit(c) && c != '.');  }  return ParseOperator(code, position);  }  private string Parse(string code, int position, Func<char, bool> stopRule)  {  char symbol;  StringBuilder buffer = new StringBuilder();  while (position < code.Length)  {  symbol = code[position];  if (stopRule(symbol))  {  break;  }  buffer.Append(symbol);  position++;  }  return buffer.ToString();  }  private string ParseOperator(string code, int position)  {  string symbol = code[position].ToString();  string firstCharacter = "<>=&!|";  string secondCharacter = "=&|";  if (position < code.Length - 1)  {  if (firstCharacter.Contains(symbol) && secondCharacter.Contains(code[position + 1]))  {  symbol += code[position + 1];  }  }  return symbol;  }  }  } |

|  |
| --- |
| Token.cs |
| using System;  using System.Collections.Generic;  using System.Globalization;  using System.Linq;  using System.Text.RegularExpressions;  namespace CompilerDemo.Model  {  internal enum TokenType  {  Identifier = 0,  Complex,  Integer,  Double,  Whitespace,  Less,  Greater,  OpenParenthesis,  CloseParenthesis,  IntegerLiteral,  DoubleLiteral,  Comma,  Semicolon,  Newline,  Assignment,  Plus,  Minus,  Multiply,  Divide,  Module,  Invalid  }  internal class Token  {  public static Dictionary<string, TokenType> DefaultTypes { get; }  = new Dictionary<string, TokenType>()  {  { "complex", TokenType.Complex },  { "\n", TokenType.Newline },  { " ", TokenType.Whitespace },  { ",", TokenType.Comma },  { ";", TokenType.Semicolon },  { "=", TokenType.Assignment },  { "(", TokenType.OpenParenthesis },  { ")", TokenType.CloseParenthesis },  { "+", TokenType.Plus },  { "-", TokenType.Minus },  { "\*", TokenType.Multiply },  { "/", TokenType.Divide },  { "%", TokenType.Module },  };  public TokenType Type { get; }  public string RawToken { get; }  public int StartPos { get; }  public int EndPos { get => StartPos + RawToken.Length; }  public Token(string rawToken, int startPos)  {  if (rawToken.Length == 0)  throw new ArgumentException("raw token is empty");  RawToken = rawToken;  StartPos = startPos;  Type = GetTokenType(rawToken);  }  public static bool DefaultTokenExists(string rawToken)  => DefaultTypes.ContainsKey(rawToken);  private static bool IsIdentifier(string rawToken)  {  if (rawToken == "std::complex<double>")  return false;  return rawToken.Length != 0 && (char.IsLetter(rawToken.First()) || rawToken.First() == '\_') && Regex.IsMatch(rawToken, "^[a-zA-Z0-9\_]+$");  }  private static bool IsIntegerLiteral(string rawToken)  => int.TryParse(rawToken, out int \_) && !rawToken.StartsWith("0.");  private static bool IsDoubleLiteral(string rawToken)  {  return double.TryParse(rawToken, NumberFormatInfo.InvariantInfo, out double \_)  && (rawToken.StartsWith("0.") != !rawToken.StartsWith('0'))  && !rawToken.EndsWith('.');  }  private static bool IsStringLiteral(string rawToken)  => rawToken.StartsWith("\"") && rawToken.EndsWith("\"") && !rawToken.Contains('\n') && rawToken.Length > 1;  public static TokenType GetTokenType(string rawToken)  {  if (DefaultTokenExists(rawToken))  {  return DefaultTypes[rawToken];  }  if (IsIdentifier(rawToken))  {  return TokenType.Identifier;  }  if (IsIntegerLiteral(rawToken))  {  return TokenType.IntegerLiteral;  }  if (IsDoubleLiteral(rawToken))  {  return TokenType.DoubleLiteral;  }  return TokenType.Invalid;  }  }  } |