МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ

ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра ИиСП

КУРСОВАЯ РАБОТА

По дисциплине: «Операционные системы»

НАПРАВЛЕНИЕ ПОДГОТОВКИ

09.03.04 - ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Выполнил студент 3 курса

специальности «ПС»

группы «ПС-31»

Павловский Алексей Александрович

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:

Нехорошкова Людмила Георгиевна

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Йошкар-Ола

2023 год

# Содержание

[Содержание 2](#_Toc125403811)

[Введение 3](#_Toc125403812)

[Теоретическая часть 4](#_Toc125403813)

[Практическая часть 5](#_Toc125403814)

[**Преобразование конечного автомата** 6](#_Toc125403815)

[**Минимизация конечного автомата** 7](#_Toc125403816)

[**Детерминизация конечных автоматов** 8](#_Toc125403817)

[**Преобразование регулярной грамматики в конечный автомат** 9](#_Toc125403818)

[**Сквозной пример** 10](#_Toc125403819)

[**Лексический анализатор (лексер)** 11](#_Toc125403820)

[**Синтаксический анализатор на алгоритме рекурсивного спуска** 17](#_Toc125403821)

[**Синтаксический анализатор на алгоритме LL(1) разбора** 18](#_Toc125403822)

[Заключение: 19](#_Toc125403823)

[Список использованной литературы: 20](#_Toc125403824)

# Введение

Теория автоматов — область знания, изучающая абстрактные автоматы — вычислительные машины, представленные в виде математических моделей — и задачи, которые они могут решать.

Конечный автомат (КА) - математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно.

Цель работы: изучение теории автоматов, закрепление материала, изученного в ходе прохождения курса.

Задача: разработка программ, выполняющих стандартные операции над конечным автоматом.

# Теоретическая часть

Конечные автоматы - это до предела упрощенная модель компьютера, имеющая конечное число состояний, которая жертвует всеми особенностями компьютеров, такими как ОЗУ, постоянная память, устройства ввода-вывода и процессорными ядрами в обмен на простоту понимания, удобство рас­суждения и легкость программной или аппаратной реализации.

У конечных автоматов имеется таблица переходов, текущее состояние автомата, стартовое состояние и заключительное состояние.  
  
Таблица переходов — структура, хранящая переходы между состояниями в зависимости от входного символа.

Текущее состояние — множество состояний, в котором автомат может находиться в данный момент времени.  
  
Стартовое состояние — состояние, откуда КА начинает свою работу.  
  
Заключительное состояние — множество состояний, в которых работа автомата может быть завершена.

# Практическая часть

В ходе выполнения курсовой работы были разработаны следующие программы:

1. Преобразование конечного автомата
   1. Из автомата Мили в автомат Мура
   2. Из автомата Мура в автомат Мили
2. Минимизация конечного автомата
   1. Автомата Мура
   2. Автомата Мили
3. Детерминирование конечного автомата
4. Преобразование регулярной грамматики в автомат
5. Лексический анализатор (лексер)
6. Синтаксический анализатор на алгоритме рекурсивного спуска
7. Синтаксический анализатор на алгоритме LL(1)

## **Преобразование конечного автомата**

Конечный автомат может быть представлен в форме автомата Мили или автомата Мура.

В автомате Мили сигналы на выходе напрямую зависят от состояния автомата и сигналов на входе.

В автомате Мура сигналы на выходе напрямую зависят только от состояния автомата.

Алгоритм преобразования автомата Мили в автомат Мура:

1. Каждому состоянию вида aX/bY даем уникальное имя, у идентичных состояний имя одинаково
2. Формируем автомат Мура. При этом строки в таблице названы согласно входным символам и обозначают возможные переходы по данным символам, столбцы названы согласно состояниям и выходным сигналам при переходе в это состояние и обозначают возможные переходы из этого состояния

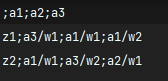
Алгоритм преобразования автомата Мура в автомат Мили:

1. Для каждого состояния добавляем к названию состояния в ячейке перехода его выходной символ

Отображение автоматов:

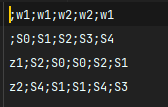
Входные и выходные данные автоматов задаются в формате csv.

При этом автомат Мили будет иметь вид



Где в первой строке заданы состояния, в первом столбце последующих строк входные сигналы, в остальных ячейках переходы и выходные сигналы.

Автомат Мура будет иметь вид



Где в первой строке заданы выходные сигналы состояний, во второй состояния, в первом столбце последующих строк входные сигналы, в остальных ячейках переходы.

Для реализации программы был выбран язык C# как наиболее знакомый мне.

Для отображения автоматов в программе достаточно использовать одномерные массивы для состояний и сигналов, и двумерные для матрицы переходов.

## **Минимизация конечного автомата**

Минимизация конечного автомата – процесс уменьшения числа состояний автомата с сохранением эквивалентности исходному.

Алгоритм минимизации конечного автомата Мили

1. Столбцы делятся на классы эквивалентности по выходным символам
2. Формируем промежуточную таблицу с заменой переходов по группам на переходы по классам эквивалентности, содержащим эти группы
3. Делим промежуточную таблицу на классы эквивалентности по переходам. Повторяем шаги 2-3 пока количество классов эквивалентности не перестанет меняться
4. Из итоговых классов эквивалентности берется по одному элементу, им дается новое название. Переходы идентичны переходам между итоговыми классами эквивалентности, с заменой названий классов на названия взятых элементов. Выходные сигналы берутся у взятых элементов

Алгоритм минимизации конечного автомата Мура

1. Столбцы делятся на классы эквивалентности по выходному символу
2. Формируем промежуточную таблицу аналогично автомату Мили, сохраняя выходные символы для классов эквивалентности
3. Делим промежуточную таблицу на классы эквивалентности по переходам. Повторяем шаги 2-3 пока количество классов эквивалентности не перестанет меняться
4. Из итоговых классов эквивалентности берется по одному элементу, переходы расписываются аналогично автомату Мили, выходные символы для группы были указаны на шаге 2

Представления автоматов Мили и Мура идентичны программе преобразования конечного автомата.

Для реализации программы был выбран язык C# как наиболее знакомый мне.

Для отображения автоматов использовались массивы (линейные структуры типа состояний и сигналов), двумерные массивы (матричные структуры) и словари для описания структур вида «ключ-значение» (например, «класс эквивалентности-массив входящих в него состояний»). Также в местах, где предполагается малое количество значений при большом количестве операций чтения/записи массивы были заменены на хеш-таблицы (выигрывают в быстродействии, но проигрывают в занимаемой памяти).

## **Детерминизация конечных автоматов**

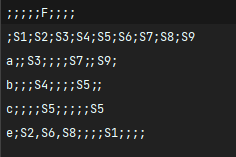
Детерминизация автомата – приведение автомата в состояние детерминированности, т.е. наличия для каждого состояния не более чем одного перехода по каждому символу и отсутствия эпсилон-переходов

Алгоритм детерминизации конечного автомата:

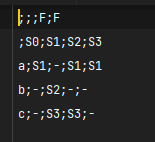
1. Формируем список транзитивных замыканий для эпсилон-переходов для каждой из вершин
2. Формируем таблицу новых вершин. В первую строку пишем транзитивное замыкание начальной вершины, вычисляем переходы из нее по различным выходным сигналам. Новые переходы пишем во временную таблицу. Идем по временной таблице игнорируя уже обработанные переходы, действуем аналогично первой строке
3. Когда во временной таблице кончатся записи, переименовываем вершины в человекочитаемый формат

Входные и выходные данные задаются в csv формате.

Пример входного файла:



Пример выходного файла:



Прочерк обозначает отсутствие перехода.

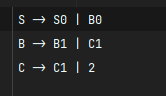
По опыту прошлого задания было принято решение использовать язык Python3, как более удобный для работы с массивами при низких требованиях к абстрактности кода.

Для представления автоматов использовались стандартные массивы (наиболее простой подход) и реализация массивов из библиотеки numpy (большее число полезных функций, в том числе умение считывать двумерные массивы напрямую с csv файлов).

## **Преобразование регулярной грамматики в конечный автомат**

При чтении регулярной грамматики мы можем легко выделить как правила, так и переходы, и на их основе получить недетерминированный автомат. Тогда задача сводится к использованию кода предыдущей программы (детерминизация конечных автоматов), так как мы имеем частные случай ее входных данных (недетерминированный конечный автомат без эпсилон-переходов).

На входе регулярная грамматика представлена в человекочитаемом виде:



На выходе мы имеем csv файл с детерминированным автоматом, аналогичный прошлому заданию.

В целях переиспользования кода был выбран язык Python3.

Для представления автомата так же использовались встроенные массивы и массивы numpy. Также общий код (детерминизация автомата) был вынесен в отдельный файл в целях избежания дублирования кода.

## **Сквозной пример**

Дан сквозной пример: (a\*c\*b\*|c)+a+

Целью данной части работы является получение из него детерминированного минимизированного конечного автомата с использованием описанных выше программ.

После ручного преобразования регулярного выражения мы получаем недетерминированный автомат вида

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | F |
|  | S0 | S1 | S2 | S3 | S4 | S5 | S6 |
| a |  | S1 |  |  |  | S5,S6 |  |
| b |  |  |  | S3 |  |  |  |
| c | S4 |  | S2 |  |  |  |  |
| e | S1 | S2 | S3 | S4 | S0,S5 |  |  |

Где F – конечное состояние.

После работы программы по детерминизации мы получаем детерминированный конечный автомат вида

|  |  |  |
| --- | --- | --- |
|  |  | F |
|  | S0 | S1 |
| a | S1 | S1 |
| b | S0 | S0 |
| c | S0 | S0 |

Запустив программу по минимизации с данным автоматом в качестве входного значения мы получаем неизменный автомат, что означает, что он уже является минимальным.

## **Лексический анализатор (лексер)**

Лексический анализатор – составная часть компилятора, преобразующая листинг кода (лексемы) в набор токенов языка.

В качестве языка будет использоваться упрощенная версия C#:

1. Типы данных ограничены int, float, bool, string
2. Отсутствует поддержка динамической типизации
3. Отсутствует цикл с постусловием
4. Отсутствует оператор безусловного перехода
5. Отсутствует определение сигнатуры методов
6. Отсутствуют лексемы для ООП-кода
7. Поддержка только однострочных комментариев
8. Перед и после сепараторов (кроме ;) необходимо ставить пробелы

Для реализации кода был выбран язык C#, так как он позволяет значительно улучшить читаемость кода за счет элементов ООП-парадигмы (полиморфизм) и упрощенного синтаксиса (методы расширения).

Листинг кода считывается из входного файла. Итоговый массив токенов выводится в стандартный поток вывода. В процессе работы программы листинг кода читается построчно и делится на составляющие части по пробелам. Каждая часть анализируется на предмет того, содержит ли она валидную лексему. Если да, то лексема преобразуется к токену, иначе выводится ошибка. Для упрощения читаемости кода анализаторы лексем подставляются с помощью полиморфизма.

Пример листинга кода:

a = ( a + 1 );

b = "string";

int a;

float a;

float a;

b = a + ( b \* a );

{

a = ( a + 1 );

b = "string";

int a;

float a;

float a;

b = a + ( b \* a );

}

//comment

while ( int i = 0; i = index + 1 )

{

for ( var indexI = 0; index < b )

switch ( indexI )

case : 0

indexI = index + 1;

break;

case : 1

indexI = index + 2;

break;

default

return;

}

Должен порождать выходной массив токенов

1: Identifier a 1:0

2: Assigment = 1:2

3: Separator ( 1:4

4: Identifier a 1:6

5: Addition + 1:8

6: Number 1 1:10

7: Separator ) 1:12

8: Separator ; 1:13

9: Identifier b 2:0

10: Assigment = 2:2

11: String "string" 2:4

12: Separator ; 2:12

13: Keyword int 3:0

14: Identifier a 3:4

15: Separator ; 3:5

16: Keyword float 4:0

17: Identifier a 4:6

18: Separator ; 4:7

19: Keyword float 5:0

20: Identifier a 5:6

21: Separator ; 5:7

22: Identifier b 6:0

23: Assigment = 6:2

24: Identifier a 6:4

25: Addition + 6:6

26: Separator ( 6:8

27: Identifier b 6:10

28: Multiplication \* 6:12

29: Identifier a 6:14

30: Separator ) 6:16

31: Separator ; 6:17

32: Separator { 7:0

33: Identifier a 8:0

34: Assigment = 8:2

35: Separator ( 8:4

36: Identifier a 8:6

37: Addition + 8:8

38: Number 1 8:10

39: Separator ) 8:12

40: Separator ; 8:17

41: Identifier b 9:0

42: Assigment = 9:2

43: String "string" 9:4

44: Separator ; 9:16

45: Keyword int 10:0

46: Identifier a 10:4

47: Separator ; 10:9

48: Keyword float 11:0

49: Identifier a 11:6

50: Separator ; 11:11

51: Keyword float 12:0

52: Identifier a 12:6

53: Separator ; 12:11

54: Identifier b 13:0

55: Assigment = 13:2

56: Identifier a 13:4

57: Addition + 13:6

58: Separator ( 13:8

59: Identifier b 13:10

60: Multiplication \* 13:12

61: Identifier a 13:14

62: Separator ) 13:16

63: Separator ; 13:21

64: Separator } 14:0

65: Keyword while 16:0

66: Separator ( 16:6

67: Keyword int 16:8

68: Identifier i 16:12

69: Assigment = 16:14

70: Number 0 16:16

71: Separator ; 16:33

72: Identifier i 16:19

73: Assigment = 16:21

74: Identifier index 16:23

75: Addition + 16:29

76: Number 1 16:31

77: Separator ) 16:33

78: Separator { 17:0

79: Keyword for 18:0

80: Separator ( 18:4

81: Keyword var 18:6

82: Identifier indexI 18:10

83: Assigment = 18:17

84: Number 0 18:19

85: Separator ; 18:36

86: Identifier index 18:22

87: Comparator < 18:28

88: Identifier b 18:30

89: Separator ) 18:32

90: Keyword switch 19:0

91: Separator ( 19:7

92: Identifier indexI 19:9

93: Separator ) 19:16

94: Keyword case 20:0

95: Separator : 20:5

96: Number 0 20:7

97: Identifier indexI 21:0

98: Assigment = 21:7

99: Identifier index 21:9

100: Addition + 21:15

101: Number 1 21:17

102: Separator ; 21:30

103: Keyword break 22:0

104: Separator ; 22:17

105: Keyword case 23:0

106: Separator : 23:5

107: Number 1 23:7

108: Identifier indexI 24:0

109: Assigment = 24:7

110: Identifier index 24:9

111: Addition + 24:15

112: Number 2 24:17

113: Separator ; 24:30

114: Keyword break 25:0

115: Separator ; 25:17

116: Keyword default 26:0

117: Keyword return 27:0

118: Separator ; 27:18

119: Separator } 28:0

## **Синтаксический анализатор на алгоритме рекурсивного спуска**

Синтаксический анализатор – часть компилятора, отвечающая за выделение лексем в листинге кода.

Алгоритм рекурсивного спуска основан на вложенности правил – он последовательно читает листинг согласно входному правилу, и если далее ожидается вложенное правило – просто рекурсивно передает управление обработчику вложенного правила, работающему по идентичному алгоритму.

В основе данного синтаксического анализатора лежит следующий набор правил:

<PROG> -> PROG id <VAR> begin <LISTST> end

<VAR> -> VAR <IDLIST> : <TYPE>

<IDLIST> -> id | < IDLIST>, id

<LISTST> -> <ST> | <LISTST> <ST>

<TYPE> -> int|float|bool|string

<ST> -> <READ>|<WRITE>|<ASSIGN>

<ASSIGN> -> id := < EXP> ;

<EXP> -> <EXP> + <T> | <T>

<T> -> <T>\*<F> | <F>

<F> -> -<F>| (<EXP>)| id|num

<READ> -> READ (<IDLIST>);

<WRITE> -> WRITE (<IDLIST>);

Для реализации был выбран язык C# для более удобной проброски ошибок за счет механизма исключений. За счет этого также была увеличена читаемость кода за счет уменьшения вложенности.

Проверка для каждого правила реализована в виде метода без возвращаемого значения. В случае ошибки парсинга выбрасывается исключение с информацией об ошибке. Завершение работы метода означает корректность лексемы.

Пример входного листинга:

PROG id

VAR id, id, id : int;

BEGIN

READ(id,id);

id:= id+num\*-(-num+id\*id\*-num\*(id+-num));

WRITE(id,id,id);

END

Результат работы программы – отсутствие ошибок, сообщение об успешном парсинге

## **Синтаксический анализатор на алгоритме LL(1) разбора**

Данный алгоритм разбора основан на построении таблицы грамматики и написания программы, перемещающейся по этой таблице в процессе работы анализатора. При этом программа для работы с таблицей является универсальной, для смены используемой грамматики достаточно заменить таблицу.

Обрабатываемая данной программой грамматика задана следующими правилами:

<EXP> -> <EXP> + <T>|<T>

<T> -> <T> \* <F>|<F>

<F> -> - <F>|(<EXP>)|a|b|8|3

Для реализации был выбран язык C# за счет удобного описания сущностей и механизма исключений.

Пример входных данных:

-8\*-(-3+a\*b\*-8\*(a+-3))

Выходные данные:

Успешный парсинг

# Заключение:

В ходе данной курсовой работы были написаны программы, реализующие стандартные операции над конечными автоматами (конвертация, минимизация, детерминизация). Также были разработаны синтаксические анализаторы на основе алгоритмов рекурсивного спуска и LL(1), и лексический анализатор для подмножества языка C#.

Таким образом можно сделать вывод, что цель и задача курсовой работы выполнены.

# Список использованной литературы:

1. Курс предмета «Операционные системы» (лекции и лабораторные работы)
2. <https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82>
3. <https://habr.com/ru/post/358304/>