МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
ТОЧНОЙ МЕХАНИКИ И ОПТИКИ (ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)

УДК 681.3.06: 62—507
ВГК ОКП
№ регистрационный 01.20.00 13546
Инв. №

"УТВЕРЖДАЮ"
Ректор СПбГИТМО (ТУ)
докт. техн. наук, профессор

_______________ Васильев В.Н.

ОТЧЕТ

по научно-исследовательской работе № 10038
"Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода"

Этап 3

"Применение автоматного подхода для программной реализации вычислительных алгоритмов"

Руководитель НИР
докт. техн. наук,
профессор,
заведующий кафедры
"Информационные системы" Шалыто А.А.

Санкт-Петербург
2002

Список основных исполнителей

Руководитель НИР

Доктор технических наук, профессор, заведующий кафедры "Информационные системы"	Шалыто А.А.

Исполнители

Инженер-программист	Туккель Н.И.
Аспирант	Шамгунов Н.Н.
Аспирант	Шопырин Д.Г.
Аспирант	Резанов С.Н.
Аспирант	Гуров В.С.
Аспирант	Казаков М.А.
Студент	Макаров Н.С.
Студент	Крылов Р.А.
Студент	Григорьев А.Э.
Студент	Мазин М.А.
Студент	Корнеев Г.А.
Студент	Станкевич А.А.
Студент	Наумов Л.А.
Студент	Наумов А.С.
Студент	Штучкин А.Н.

Реферат

Отчет содержит 89 страницу, 70 рисунков, 28 источников литературы.

Система управления, вычислительный алгоритм, схема алгоритма, состояние, автомат, граф переходов, протокол, алгоритмизация, автоматное программирование, проектирование программ

Целью настоящей работы является разработка технологии автоматного программирования применительно к вычислительным алгоритмам.

В работе рассмотрен вопрос об использовании вычислительной модели, основанной на изменении машины Тьюринга за счет применения структурного автомата с произвольными множествами входных и выходных воздействий вместо абстрактного. Эта модель может быть использована при реальном программировании, что показано на примерах.

Предлагается метод преобразования итеративных программ в автоматные программы, который может быть применен также для преобразования неструктурированных схем алгоритмов в автоматные программы. Показано, что рассматриваемый метод обладает рядом достоинств по сравнению с методом Ашкрофта-Манны. Метод иллюстрируется примерами.

Предлагается метод преобразования программ с явной рекурсией в итеративные автоматные программы. Приведены примеры классических задач, решаемых с использованием предлагаемого подхода.

Содержание

Введение 5
1. Применение автоматов при реализации итеративных алгоритмов 7
1.1. От тьюрингова программирования к автоматному...................7
1.2. Классические задачи распознавания цепочек символов..........9
1.2.1. Распознавание скобок произвольной глубины...............11
1.2.2. Распознавание цепочек языка $\{1^n0^n \mid n \geq 0\}$ 17
1.2.3. Распознавание цепочек языка $\{1^n0^n1^n \mid n \geq 0\}$.............19
1.2.4. Распознавание цепочек нескольких языков.................20
2. Преобразование итеративных алгоритмов в автоматные программы .23
2.1. Изложение метода...25
2.2. Примеры преобразования итеративных программ в автоматные...28
2.3. Преобразование неструктурированных алгоритмов в автоматные программы 35
3. Преобразование программ с явной рекурсией в автоматные 42
3.1. Изложение метода...43
3.2. Вычисление факториала..45
3.3. Вычисление чисел Фибоначчи....................................49
3.4. Задача о ранце..53
3.5. Задача о ханойских башнях......................................61
3.5.1. Классическое рекурсивное решение задачи.................62
3.5.2. Моделирование рекурсии автоматной программой............65
3.5.3. Реализация рекурсивной программы автоматно-рекурсивной программой 71
3.5.4. Реализация автоматной программы с использованием классов 72
3.5.5. Обход дерева действий.....................................75
3.5.6. Решение задачи в терминах "объекта управления"..........80
4. Заключение 84
5. Публикации по результатам этапа 86
Список литературы 88

Введение

В отчете излагаются основные положения технологии автоматного программирования применительно к вычислительным алгоритмам.

Отчет создан в ходе выполнения работ по теме "Разработка технологии создания программного обеспечения систем управления на основе автоматного подхода" (регистрационный номер темы во Всероссийском научно-техническом центре (ВНТИЦ) - 01.20.00 13546), третий этап которой, выполненный в 2002 г., имеет наименование "Применение автоматного подхода для программной реализации вычислительных алгоритмов".

Работа является продолжением исследований, проведенных в 2000 и 2001 годах по первому и второму этапам темы. Первый этап был посвящен разработке технологии автоматного программирования для систем логического управления. На втором этапе был создан вариант указанной технологии для событийных систем.

Третий этап посвящен еще одному классу задач — вычислительным алгоритмам. Перечислим характерные особенности задач этого класса:

– входные воздействия являются двоичными переменными;

– входные и выходные воздействия реализуются в виде функций (подпрограмм);

– автоматы запускаются в конечном цикле;

– время в явном виде отсутствует.

В первой части отчета рассматривается преобразование классической модели машины Тьюринга в другую модель, которая была бы более эффективна в практическом использовании. В предложенной модели вместо абстрактного автомата применяется структурный автомат с произвольным

количеством входов и выходов, реализуемых произвольными подпрограммами. Эффективность предложенной модели демонстрируется на примере классических для теории трансляторов задач распознавания цепочек символов. Показано, что каждая из рассмотренных задач может быть решена в рамках предложенной модели на основе конечных автоматов, в то время как в теории трансляторов первые две задачи решаются с использованием автоматов с магазинной памятью, а третья задача требует использования линейно-ограниченного автомата.

Во второй части отчета предлагается метод преобразования итеративных программ и схем алгоритмов в автоматные программы. Предложенный метод применим также и для преобразования неструктурированных программ и схем алгоритмов в автоматные программы. Рассмотрены примеры и показано, что предложенный метод позволяет получать схемы алгоритмов с меньшим, чем при использовании метода Ашкрофта-Манны, количеством вершин.

В третьей части отчета предлагается метод преобразования программ с явной рекурсией в итеративные автоматные программы. Этот метод напоминает подходы по снятию концевых рекурсий, рассмотренные в работах [4,19], однако, в отличие от последних, является формальным и одинаково применим к различным программам с явными рекурсивными вызовами. Предложенный метод иллюстрируется на примерах классических задач, имеющих рекурсивное решение: вычисление факториала и чисел Фибоначчи, задачи о ранце и о ханойских башнях. При этом задача о ханойских башнях рассмотрена более подробно, чем требуется для иллюстрации метода — на примере этой задачи дополнительно предлагается два эвристических подхода, позволяющих раскрывать рекурсию и строить автоматные программы.

1. Применение автоматов при реализации итеративных алгоритмов

1.1. От тьюрингова программирования к автоматному

> В понимании теории программ большую помощь могут оказать один-два оператора, описывающих состояние машины...
>
> А.Тьюринг [1]

Известна роль машин Тьюринга как универсального средства реализации алгоритмов [2,3]. В работе [3] программирование этих машин названо "тьюрингово программирование". Однако применение такой парадигмы программирования на практике невозможно из-за чрезвычайной простоты тьюринговых операций и крайне ограниченного доступа к внешней памяти машины, что приводит к большой трудоемкости написания программ этого типа и значительному числу шагов при их выполнении. Так, например, умножение на машине Тьюринга двух на три требует (при задании этих чисел штрихами на ленте) 188 шагов [2].

Рассмотрим преобразование классической модели машины Тьюринга в другую модель, обеспечивающую переход от тьюрингова программирования к автоматному. Последнее весьма эффективно для систем со сложным поведением, характерным для задач управления или сводимым к ним, как показано ниже.

Классическая схема машины Тьюринга (рис. 1), состоящая из абстрактного конечного автомата и бесконечной ленты, может быть изображена по-другому (рис. 2). Такое изображение характерно для теории управления и содержит контур обратной связи. Отметим, что на этих рисунках не изображен выход автомата, обеспечивающий управление передвижением ленты.

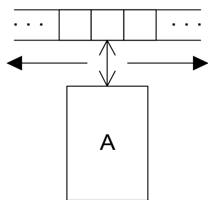


Рис. 1. Машина Тьюринга

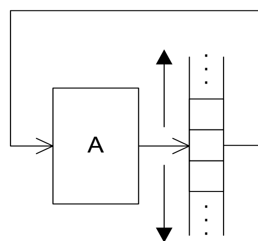


Рис. 2. Другое изображение машины Тьюринга

Абстрактный автомат, применяемый в машине Тьюринга, сам по себе не является универсальным и не удобен для практического применения. Второй из указанных недостатков снимается при переходе от абстрактного автомата к структурному (рис. 3), главное отличие которого заключается в параллелизме по входам и выходам.

Рассмотрим предлагаемую модель (рис. 4), базирующуюся на структурном автомате, в которую для обеспечения универсальности и практического применения введены формирователи входных воздействий 1, а самое главное — объекты управления 2.

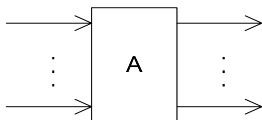


Рис. 3. Структурный автомат

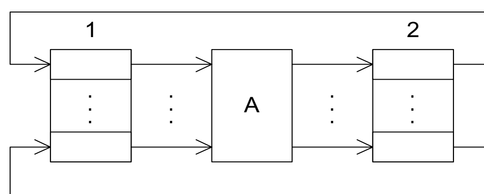


Рис. 4. Схема связей автомата

Эту модель будем называть "схема связей автомата". В ней формирователь 1 преобразует поступающую от источников информацию во входные воздействия, например, путем сравнения значения счетчика с нулем. В этой схеме объектами управления могут быть устройства, в которых по командам автомата возможно выполнение сколь угодно сложных операций (в том числе и над памятью).

В предложенной модели выходные воздействия формируются в определенных "точках" пространства

состояний, задаваемого графом переходов автомата. Поэтому процесс управления объектами полностью задается этим графом в наглядной и понятной форме.

1.2. Классические задачи распознавания цепочек символов

Эффективность применения предлагаемой модели покажем на примере классических задач распознавания цепочек символов.

Известно [4,5], что теория построения трансляторов базируется на иерархии моделей автоматов. При этом для задач распознавания цепочек символов с усложнением языка усложняется также модель применяемого автомата. Покажем, что при использовании предлагаемого подхода перечисленные ниже задачи могут быть решены в рамках одной модели (рис. 4).

Рассмотрим три классические [5] задачи распознавания цепочек для:

- скобок произвольной глубины;
- языка $\{1^n0^n \mid n \geq 0\}$;
- языка $\{1^n0^n1^n \mid n \geq 0\}$.

Несмотря на то, что первая задача кажется сложнее второй, это не так. В первой из них проверяется только парность скобок, а во второй аналогичная задача решается после определения правильности порядка символов.

Применение распознавателей, являющихся конечными автоматами без выхода, позволяет, в частности, решить задачу распознавания четности числа символов в цепочке. Однако даже возможность формирования выходных цепочек в классической модели конечного автомата не позволяет на ее основе решить перечисленные выше задачи. При этом первые два языка, задаваемые контекстно-свободными грамматиками, требуют использования модели автомата с магазинной

памятью, а для третьего языка, определяемого контекстно-зависимой грамматикой, необходим переход к линейно-ограниченным автоматам [5].

В первых двух рассматриваемых задачах в качестве объектов управления выступают счетчик, заменяющий магазин, и два индикатора "Допустить" и "Отвергнуть". Счетчик может быть обнулен, а его значение — увеличено или уменьшено на единицу. С выхода счетчика на вход автомата поступает информация о том, равно ли значение счетчика нулю.

В третьей задаче в схему связей введен дополнительный объект управления — переменная, запоминающая значение счетчика. На вход автомата дополнительно поступает информация о том, что значение счетчика равно n.

При реализации (в рамках предлагаемой технологии) преобразование графа переходов в текст программы выполняется формально и изоморфно. При этом граф переходов автомата Мили реализуется одним оператором switch. Если на всех входящих дугах некоторой вершины этого графа присутствуют одинаковые действия, то они могут быть перенесены в нее. Таким образом, автомат Мили преобразуется в более компактный смешанный автомат, который, как и автомат Мура, реализуется по шаблону, содержащему два оператора switch [6].

Хотелось бы верить, что упомянутые в эпиграфе один или два оператора относятся к одному или двум операторам switch, используемым в предлагаемом подходе.

В классической теории трансляторов в описание автомата включены действия, обеспечивающие чтение с ленты. В предлагаемом подходе такая работа с лентой может быть заменена работой с массивом, обрабатываемым в цикле,

тело которого содержит вызов функции, реализующей автомат.

1.2.1. Распознавание скобок произвольной глубины

Первая задача была решена двумя способами. Первый из них базируется на применении автомата с одним состоянием, для которого может быть построен граф переходов с одной вершиной и четырьмя петлями (рис. 5). Это решение совпадает с приведенным в работе [5] с точностью до работы с лентой и замены магазина счетчиком. По мнению авторов, более естественным является второй способ, при использовании которого распознавание выполняется не на петлях, а в соответствующих состояниях смешанного автомата: "Начальное", "Анализ", "Допустить", "Отвергнуть". В этом графе переходов переменные x2, x3 и x4 ортогональны.

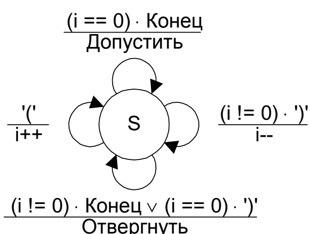


Рис. 5. Распознавание скобок произвольной глубины. Традиционный подход

В соответствии с предложенной моделью для второго способа построена схема связей (рис. 6) и граф переходов автомата Мили (рис. 7).

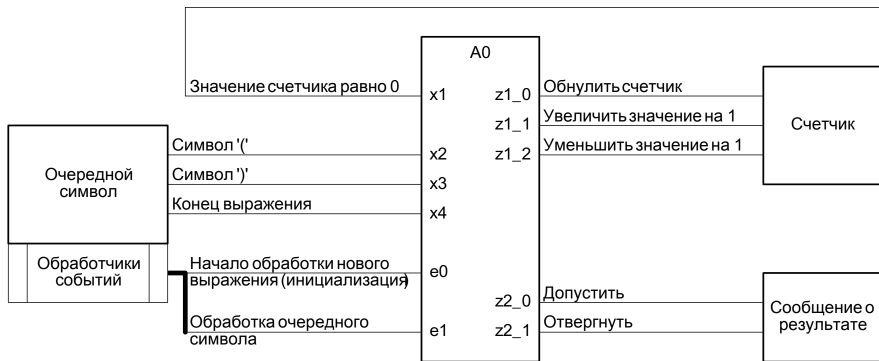


Рис. 6. Распознавание скобок произвольной глубины. Схема связей автомата

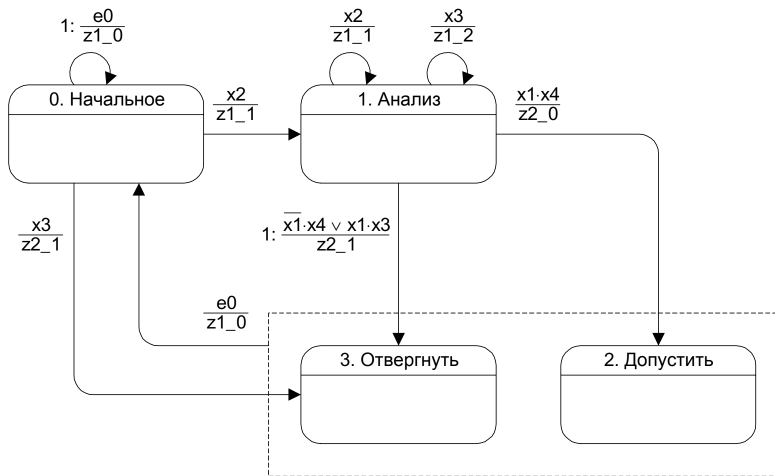


Рис. 7. Распознавание скобок произвольной глубины. Граф переходов автомата Мили

На рис. 8 этот граф упрощен за счет перехода к смешанному автомату, в котором действия с дуг перенесены в вершины, что позволило использовать пометку z2_1 только один раз. При этом отметим, что пометки действий с петель в вершины не могут быть перенесены, так как на петле состояние автомата не изменяется, а в предложенной в [6] стандартной реализации действия в вершинах выполняются только при изменении состояний.

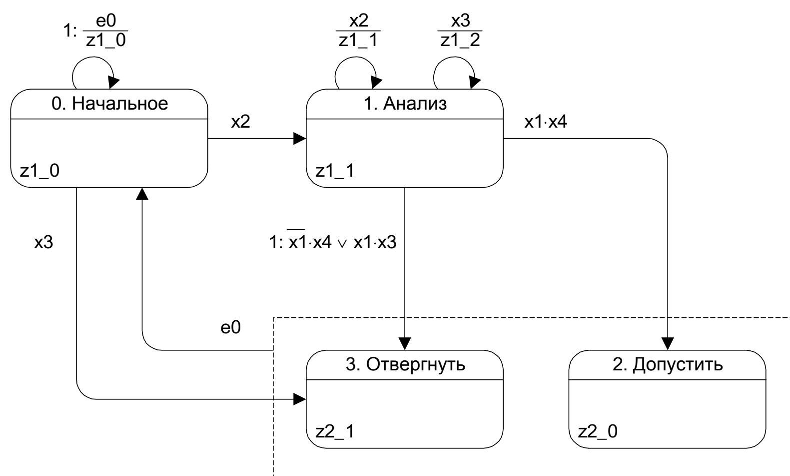


Рис. 8. Распознавание скобок произвольной глубины. Граф переходов смешанного автомата

Для упрощения чтения графа переходов в нем все абстрактные обозначения (кроме e0) могут быть заменены смысловыми (рис. 9).

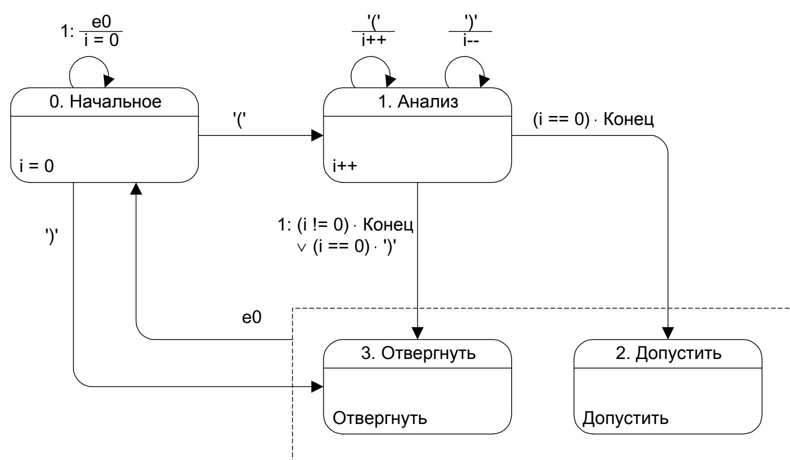


Рис. 9. Распознавание скобок произвольной глубины. Граф переходов смешанного автомата со смысловыми обозначениями

В соответствии с предложенной в [6] программной реализацией автоматов, номер события передается функции, реализующей автомат, в качестве первого параметра. Вторым параметром передается код очередного символа.

Событие e0 предназначено для инициализации автомата перед началом обработки нового выражения, и поэтому петли в графах переходов, помеченные этим событием, имеют первый приоритет. Событие e1, указанное на рис. 6, не используется в графах переходов и предназначено для ввода

очередного символа. Оно применяется в программе при вызове автомата в цикле.

В листингах 1–3 приведены программы для решения первой задачи вторым способом. Каждая из этих программ содержит цикл, моделирующий последовательный ввод символов цепочки, и программную реализацию соответствующего автомата. При этом функция автомата вызывается из тела цикла.

Листинг 1 содержит программу, в которой автомат Мили (рис. 7) формально и изоморфно [6] реализован одним оператором switch.

ЛИСТИНГ 1. Автоматная программа распознавания последовательностей скобок произвольной глубины, реализующая автомат Мили

```
#include <stdio.h>
#include <string.h>

int     y0 = 0 ;    // Переменная состояния автомата
int     i = 0 ;     // Счетчик

void main()
{
  char  str[100] = "" ;
  int   j ;
  for(;;)
  {
    printf( "\nВведите строку: " ) ;
    scanf( "%s", str ) ;
    printf( "Введена строка: %s", str ) ;
    A0( 0, 0 ) ;
    for( j = 0 ; j <= strlen(str) ; j++ )
      A0( 1, str[j] ) ;
  }
}

void A0( int e, char c )
{
  switch( y0 )
  {
    case 0:
      if( e == 0 )              { z1_0() ; }
      else
      if( x2(c) )               { z1_1() ;        y0 = 1 ; }
      else
      if( x3(c) )               { z2_1() ;        y0 = 3 ; }
    break ;
    case 1:
      if( !x1() && x4(c) || x1() && x3(c) )
                                { z2_1() ;        y0 = 3 ; }
      else
      if( x1() && x4(c) )       { z2_0() ;        y0 = 2 ; }
      else
      if( x2(c) )               { z1_1() ; }
      else
      if( x3(c) )               { z1_2() ; }
    break ;
```

```
    case 2:
      if( e == 0 )              { z1_0() ;          y0 = 0 ; }
    break ;
    case 3:
      if( e == 0 )              { z1_0() ;          y0 = 0 ; }
    break ;
  }
}

int x1()          { return i == 0 ; }
int x2( char c )  { return c == '(' ; }
int x3( char c )  { return c == ')' ; }
int x4( char c )  { return c == 0 ; }
void z1_0()       { i = 0 ; }
void z1_1()       { i++ ; }
void z1_2()       { i-- ; }
void z2_0()       { printf( "\nДопустить.\n" ) ; }
void z2_1()       { printf( "\nОтвергнуть.\n" ) ; }
```

Листинги 2 и 3 содержат программы, в которых смешанные автоматы, представленные на рис. 8 и 9 соответственно, реализованы двумя операторами switch. Эти реализации также построены формально и изоморфно.

ЛИСТИНГ 2. Автоматная программа распознавания последовательностей скобок произвольной глубины, реализующая смешанный автомат

```
#include <stdio.h>
#include <string.h>

int     y0 = 0 ;    // Переменная состояния автомата
int     i = 0 ;     // Счетчик

void main()
{
  char  str[100] = "" ;
  int   j ;
  for(;;)
  {
    printf( "\nВведите строку: " ) ;
    scanf( "%s", str ) ;
    printf( "Введена строка: %s", str ) ;
    A0( 0, 0 ) ;
    for( j = 0 ; j <= strlen(str) ; j++ )
      A0( 1, str[j] ) ;
  }
}

void A0( int e, char c )
{
  int y_old = y0 ;

  switch( y0 )
  {
    case 0:
      if( e == 0 )              { z1_0() ; }
      else
      if( x2(c) )               y0 = 1 ;
      else
      if( x3(c) )               y0 = 3 ;
    break ;
    case 1:
      if( !x1() && x4(c) || x1() && x3(c) )
                                y0 = 3 ;
```

```
      else
      if( x1() && x4(c) )       y0 = 2 ;
      else
      if( x2(c) )               { z1_1() ; }
      else
      if( x3(c) )               { z1_2() ; }
    break ;
    case 2:
      if( e == 0 )              y0 = 0 ;
    break ;
    case 3:
      if( e == 0 )              y0 = 0 ;
    break ;
  }

  if( y_old == y0 ) return ;

  switch( y0 )
  {
    case 0:
      z1_0() ;
    break ;
    case 1:
      z1_1() ;
    break ;
    case 2:
      z2_0() ;
    break ;
    case 3:
      z2_1() ;
    break ;
  }
}

int x1()          { return i == 0 ; }
int x2( char c ) { return c == '(' ; }
int x3( char c ) { return c == ')' ; }
int x4( char c ) { return c == 0 ; }
void z1_0()       { i = 0 ; }
void z1_1()       { i++ ; }
void z1_2()       { i-- ; }
void z2_0()       { printf( "\nДопустить.\n" ) ; }
void z2_1()       { printf( "\nОтвергнуть.\n" ) ; }
```

ЛИСТИНГ 3. Автоматная программа распознавания последовательностей скобок произвольной глубины, реализующая смешанный автомат с реализацией входных и выходных воздействий внутри функции автомата

```
#include <stdio.h>
#include <string.h>

int     y0 = 0 ;    // Переменная состояния автомата
int     i = 0 ;     // Счетчик

void main()
{
  char  str[100] = "" ;
  int   j ;
  for(;;)
  {
    printf( "\nВведите строку: " ) ;
    scanf( "%s", str ) ;
    printf( "Введена строка: %s", str ) ;
    A0( 0, 0 ) ;
    for( j = 0 ; j <= strlen(str) ; j++ )
      A0( 1, str[j] ) ;
```

```
  }
}

void A0( int e, char c )
{
  int y_old = y0 ;

  switch( y0 )
  {
    case 0:
      if( e == 0 )         { i = 0 ; }
      else
      if( c == '(' )                                   y0 = 1 ;
      else
      if( c == ')' )                                   y0 = 3 ;
    break ;
    case 1:
      if( (i != 0) && (c == 0) || (i == 0) && (c == ')') )   y0 = 3 ;
      else
      if( (i == 0) && (c == 0) )                       y0 = 2 ;
      else
      if( c == '(' )       { i++ ; }
      else
      if( c == ')' )       { i-- ; }
    break ;
    case 2:
      if( e == 0 )                                     y0 = 0 ;
    break ;
    case 3:
      if( e == 0 )                                     y0 = 0 ;
    break ;
  }

  if( y_old == y0 ) return ;

  switch( y0 )
  {
    case 0:
      i = 0 ;
    break ;
    case 1:
      i++ ;
    break ;
    case 2:
      printf( "\nДопустить.\n" ) ;
    break ;
    case 3:
      printf( "\nОтвергнуть.\n" ) ;
    break ;
  }
}
```

Интересно отметить, что упрощение графа переходов (переход от автомата Мили к смешанному автомату) приводит к усложнению программной реализации (два оператора switch вместо одного).

1.2.2. Распознавание цепочек языка $\{1^n0^n \mid n \geq 0\}$

Вторая задача также была решена двумя способами. Первый из них базируется на применении автомата Мили с

двумя состояниями, в котором отсутствует достижимость начального состояния (рис. 10).

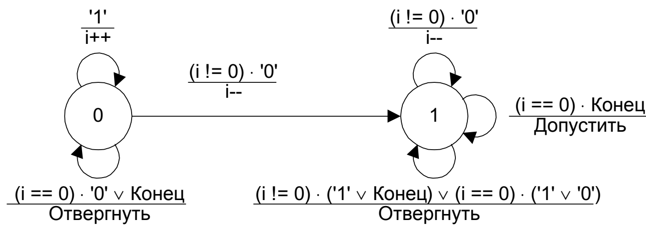


Рис. 10. Распознавание цепочек $\{1^n0^n \mid n \geq 0\}$. Традиционный подход

Это решение также совпадает с приведенным в [5] с точностью до работы с лентой и замены магазина счетчиком. Второе решение основывается на применении смешанного автомата с пятью состояниями: "Начальное", "Считать \`1\`", "Считать \`0\`", "Допустить", "Отвергнуть". Схема связей этого автомата приведена на рис. 11, а его граф переходов — на рис. 12. В этом графе переходов переменные x2, x3 и x4 ортогональны.

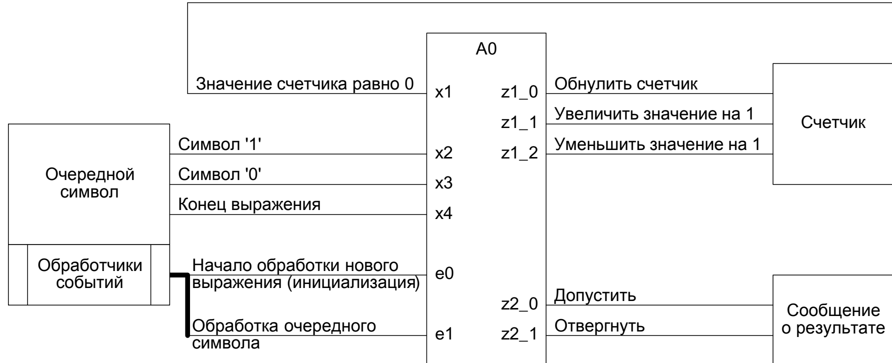


Рис. 11. Распознавание цепочек $\{1^n0^n \mid n \geq 0\}$. Схема связей автомата

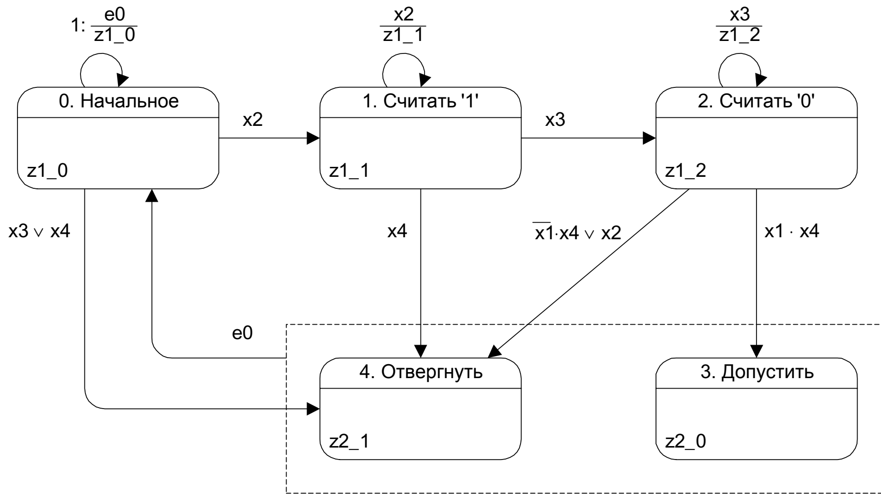


Рис. 12. Распознавание цепочек $\{1^n0^n \mid n \geq 0\}$. Граф переходов автомата

1.2.3. Распознавание цепочек языка $\{1^n0^n1^n \mid n \geq 0\}$

Третья задача, которая не может быть решена магазинным автоматом, реализуется смешанным автоматом с шестью состояниями: "Начальное", "Считать `1`", "Считать `0`", "Повторно считать `1`", "Допустить", "Отвергнуть". Схема связей этого автомата приведена на рис. 13, а его граф переходов — на рис. 14. В этом графе переходов переменные x3, x4 и x5 ортогональны.

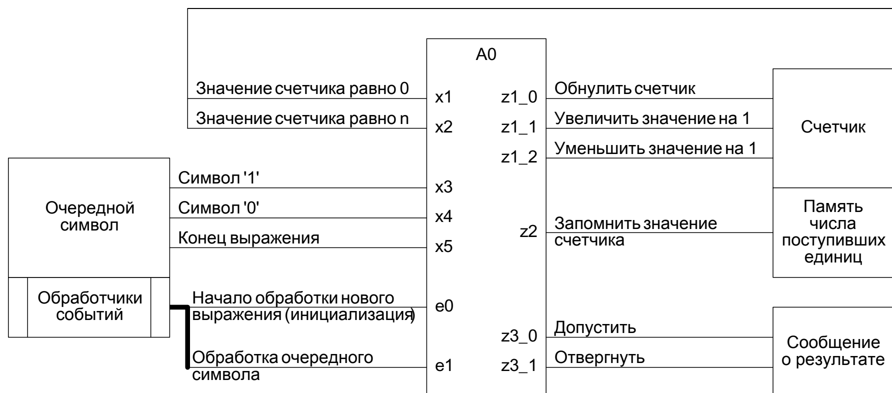


Рис. 13. Распознавание цепочек $\{1^n0^n1^n \mid n \geq 0\}$. Схема связей автомата

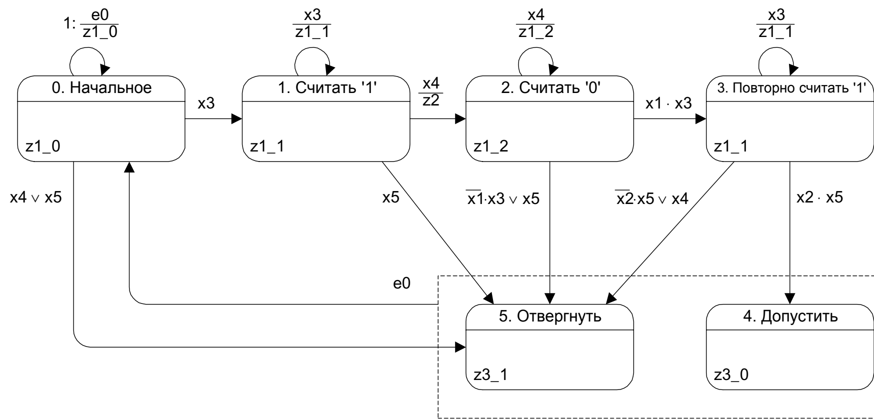


Рис. 14. Распознавание цепочек $\{1^n0^n1^n \mid n \geq 0\}$. Граф переходов автомата

Из приведенных примеров следует, что при использовании предлагаемого подхода, изменение типа грамматики не приводит к необходимости использования модели другого типа, а вызывает лишь незначительное усложнение графа переходов.

1.2.4. Распознавание цепочек нескольких языков

Кроме рассмотренных задач, на основе излагаемого подхода может быть решена задача распознавания цепочек символов языков: $\{0^n1^n, \ldots, 0^n1^n \mid n \geq 0\}$, $\{0^n1^n, \ldots, 0^n1^n, 0^n \mid n \geq 0\}$, $\{1^n0^n, \ldots, 1^n0^n \mid n \geq 0\}$ и $\{1^n0^n, \ldots, 1^n0^n, 1^n \mid n \geq 0\}$. Усложнение этой задачи по сравнению с предыдущей приводит к дальнейшему увеличению числа объектов управления в схеме связей. Новыми объектами в этой схеме являются переменные a и b, определяющие порядок следования единиц и нулей, одна из которых, в принципе, может быть исключена. Для первых двух языков переменные a и b принимают значения '0' и '1', соответственно, а для оставшихся — '1' и '0'. Эта задача, как и предыдущая, реализуется смешанным автоматом с шестью состояниями: "Начальное", "Вычисление n", "Считать a", "Считать b", "Допустить", "Отвергнуть". Схема связей этого автомата приведена на рис. 15, а его

граф переходов — на рис. 16. В этом графе переходов переменные x5, x6 и x7 ортогональны.

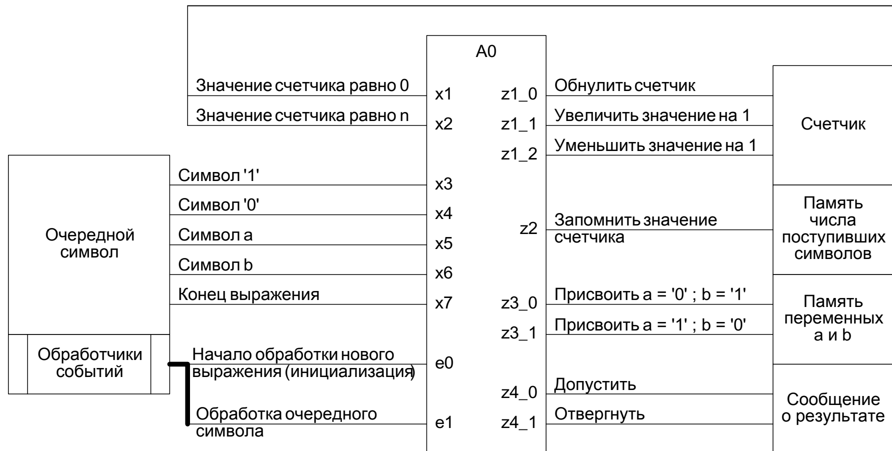


Рис. 15. Распознавание цепочек четырех языков. Схема связей автомата

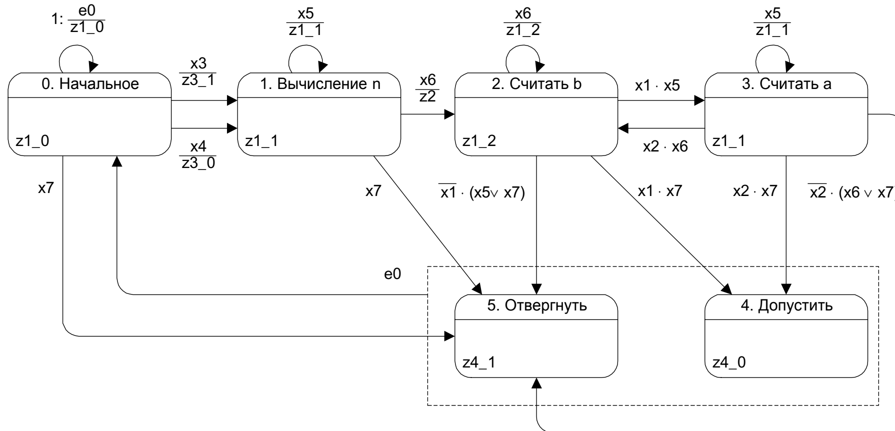


Рис. 16. Распознавание цепочек четырех языков. Граф переходов автомата

В рассматриваемой модели вместо одиночного автомата может использоваться система взаимосвязанных автоматов. Эти автоматы могут быть вложенными. Кроме этого способа взаимодействия они могут также вызывать друг друга из выходных воздействий и обмениваться номерами состояний, что позволяет весьма эффективно реализовывать параллельные процессы.

Графы переходов, используемые в предлагаемом подходе, являются не "картинками", а математическими моделями, что подтверждается автоматическим протоколированием, выполняемым в терминах автоматов [6]. Это чрезвычайно

важно, так как "протоколы (история вычислений) являются конструкциями, вскрывающими механизм работы программы и, поэтому, постепенно среди теоретиков программирования сложилось представление, что множество протоколов лучше характеризует программу, нежели сам исходный программный текст" [7]. Изложенное связано с тем, что "идея доказательства правильности программ в значительной мере исчерпала себя, так как выяснилось, что в общем случае невозможно установить свойства результата работы программы или процедуры, не исполнив ее до конца... В теории вычислений доказывается, что в общем случае распознать определенные свойства в программе можно только динамически, прослеживая весь процесс вычисления при соответствующих входных воздействиях" [8].

В заключение отметим, что в отличие от применения автоматов в теории трансляторов, предлагаемый подход может использоваться при программировании различных задач со сложным поведением. Автоматное программирование позволяет в наглядной форме (в терминах состояний и переходов) описывать процессы управления объектами, обеспечивая с целью уменьшения размерности решаемых задач сокрытие деталей, связанных с вычислениями.

Отметим также, что если Дж. фон Нейман изменил машину Тьюринга для эффективной аппаратной реализации алгоритмов (например, введением арифметическо-логического устройства) [9], то в настоящей работе делается аналогичная попытка преобразования этой машины с целью повышения эффективности программирования. При этом, если в машине Тьюринга автомат решает две задачи (управление и вычисление, не свойственное для него), то в предлагаемом подходе он решает только задачу управления, а вычисления осуществляются в объектах управления, предназначенных для этой цели. Например, автоматы (рис. 7, 8) вызывают

функцию z1_1, увеличивающую значение переменной на единицу, что отражено в текстах реализующих их программ (листинг 1 и 2). При использовании в автомате смысловых обозначений (рис. 9), его реализация, приведенная в листинге 3, по сравнению с предыдущим листингом упрощается.

Автоматный подход в программировании весьма конструктивен, так как позволяет, в отличие от работы [8], под состоянием программы понимать не значения **всех** ячеек памяти, а разделять состояния на две группы: автоматные и остальные. При этом с помощью конечного (обычно небольшого и независящего от размерности задачи) числа состояний автомата удается управлять сколь угодно большим числом состояний объекта управления, которое может увеличиваться в зависимости от размерности решаемой задачи. Так, в рассмотренных примерах число состояний в автоматах не зависит от параметра n.

2. Преобразование итеративных алгоритмов в автоматные программы

Для систем логического управления и событийных систем, рассмотренных в предыдущих этапах работы, выделение состояний является естественным, так как они в значительной мере определяются физическими состояниями объектов управления.

Для вычислительных алгоритмов выделение состояний не столь естественно, однако и для этого класса задач автоматный подход может быть применен [10].

Такой подход позволяет унифицировать процесс построения структурированных программ с визуализацией их состояний, что имеет важное значение особенно для целей обучения [11]. Это не удается обеспечить при традиционном

процедурном подходе, так как в нем в качестве базовых используются понятия "условие" и "действие", а не "состояние".

Программы, построенные на основе явного выделения состояний, назовем *автоматными*.

Психологические трудности [10], возникающие при непосредственном построении алгоритмов для вычислительных задач с применением автоматов, связаны, в частности, с непривычной реализацией циклов. Эти трудности привели к тому, что в работе [12] автоматный подход был использован только для построения одного из алгоритмов поиска подстрок, в то время как для десятков других алгоритмов, описанных в этой книге, автоматы не применялись ни при алгоритмизации, ни при реализации.

Разделим рассматриваемый класс алгоритмов на два подкласса: без использования событий (вычислительные алгоритмы) и с их использованием (событийно-вычислительные алгоритмы). Настоящая работа посвящена первому из указанных подклассов. В этом подклассе будем рассматривать схемы итеративных алгоритмов с одним входом и одним выходом, в которых через каждую вершину проходит путь от входа схемы к ее выходу (отсутствуют недостижимые вершины и бесконечные циклы).

Покажем, что схему произвольного итеративного алгоритма (программы), принадлежащую этому подклассу, можно реализовать одним оператором do-while, телом которого является один оператор switch. Последний изоморфно реализует граф переходов конечного автомата, формально построенный по исходной схеме. Отметим, что в отличие от событийных систем, в рассматриваемом подклассе алгоритмов при реализации каждого автомата используется один оператор switch.

Эта реализация проще получаемой при применении метода Ашкрофта-Манны [13,14], который требует введения состояния для каждой вершины неструктурированной схемы алгоритма и приводит к построению его структурированной схемы с числом вершин, определяемым соотношением:

$$B = 4У + 3O + 2,$$

где *У* и *О* — количество условных и операторных вершин в неструктурированной схеме алгоритма, соответственно.

Подход к структурированию программ с использованием булевых признаков [14] в настоящей работе не рассматривается, так как он является эвристическим.

2.1. Изложение метода

Ниже излагается метод построения по схеме итеративного алгоритма (программы) графа переходов, по которому, в свою очередь, строится структурированная программа, являющаяся автоматной. Если задан текст программы, то для применения предлагаемого метода необходимо предварительно построить ее схему с "раскрытыми" циклами, которую назовем традиционной. Она может быть как структурированной, так и неструктурированной.

Метод состоит из этапов, перечисленных ниже.

1. Используя подход, предложенный в работе [15] для аппаратных реализаций схем алгоритмов, в заданную схему в зависимости от выбранного для ее замены типа автомата (Мура или Мили) вводятся пометки, соответствующие номерам его состояний. При этом для автоматов обоих типов начальной и конечной вершинам схемы присваивается номер 0, что обеспечивает построение "замкнутого" графа переходов. При построении автомата Мура k-й группе соединенных последовательно операторных вершин (она может

состоять также и из одной вершины) присваивается номер k. При построении автомата Мили соответствующий номер присваивается точке, следующей за последней из последовательно соединенных операторных вершин группы, причем указанные точки для различных групп могут совпадать. Это приводит к тому, что автомат Мили имеет число состояний, не превышающее их число в эквивалентном автомате Мура. Номера, применяемые для построения автомата Мили будем указывать на схеме алгоритма в скобках, а номера для построения автомата Мура — без скобок. Используемая нумерация начальной и конечной вершин отличается от принятой в работах [13,14], где этим вершинам присваиваются различные номера, а получающийся при этом граф переходов разомкнут.

2. Обеспечение "замкнутости" графа переходов и выбранная стандартная программная реализация могут привести к необходимости введения дополнительного состояния (как для автоматов Мили, так и для автоматов Мура), если схема алгоритма содержит контур без операторных вершин и пометок других состояний.

3. В случае построения автомата Мили в схеме алгоритма за счет дублирования некоторых операторных вершин уменьшается число точек, следующих за ними, что обеспечивает сокращение числа состояний автомата этого типа.

4. В схеме программы выделяются пути между смежными точками (пометками), включая пути начинающиеся и оканчивающиеся в одной и той же точке. Для каждого пути выписываются условия перехода и выполняемые при этом действия. На основе выделенных состояний и переходов (включая петли) строится граф переходов автомата выбранного типа.

5. Построенный граф переходов изоморфно реализуется с помощью оператора switch языка Си или его аналога из других языков программирования.

6. Полученный фрагмент программы используется в качестве тела оператора do-while, условием выхода из которой является нахождение автомата в состоянии 0.

При необходимости после получения графа переходов по нему может быть построена структурированная схема программы, соответствующая оператору do-while, тело которого изоморфно графу переходов и содержит дешифратор состояний. Схемы с такой структурой назовем *автоматными*.

В отдельных случаях в построенной схеме может быть уменьшено число вершин за счет потери указанного изоморфизма (но не структурированности), по аналогии с тем, как это выполняется в [16] для схем, построенных методом Ашкрофта-Манны, с помощью перехода к рекурсивным структурированным схемам.

Завершив изложение метода, отметим, что подход, описанный в [15], состоит только из первого и четвертого этапов.

С помощью предлагаемого метода в настоящей работе решаются две задачи: преобразование традиционных программ (обычно структурированных) в автоматные (примеры 1–5) и преобразование неструктурированных схем алгоритмов в автоматные (примеры 6–12).

Перейдем к рассмотрению этих задач.

2.2. Примеры преобразования итеративных программ в автоматные

***Пример* 1.** Задана программа поиска максимума в одномерном массиве (листинг 4), по которой требуется построить программу, соответствующую автомату Мура.

ЛИСТИНГ 4. Программа поиска максимума в одномерном массиве

```
void main ()
{
  enum        { n = 8 } ;
  int         a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int         max = a[0] ;
  int         i ;

  for( i = 1 ; i < n ; i++ )
    if( a[i] > max )
      max = a[i] ;
}
```

Схема этой программы, в которой без скобок указаны пометки, соответствующие состояниям автомата Мура, приведена на рис. 17.

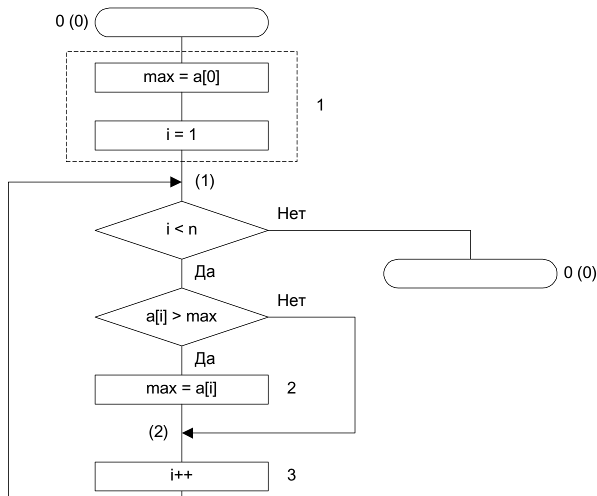


Рис. 17. Программа поиска максимума. Схема программы

Граф переходов автомата Мура с четырьмя вершинами, построенный по этой схеме, приведен на рис. 18. На этом графе пометки некоторых дуг упрощены (вычеркнуты) за счет введения приоритетов и пометки "иначе". Пометка петли исключена, так как ей при программной реализации соответствует оператор break.

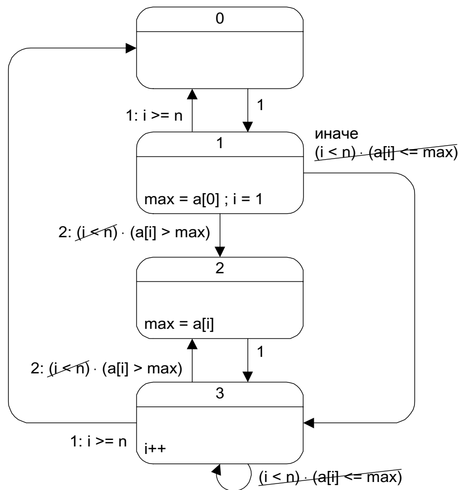


Рис. 18. Программа поиска максимума. Граф переходов автомата Мура

Используя этот граф переходов, построим автоматную программу требуемого типа (листинг 5).

ЛИСТИНГ 5. Программа поиска максимума в одномерном массиве, реализующая автомат Мура

```
void main()
{
  int       y = 0 ;
  enum      { n = 8 } ;
  int       a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int       max ;
  int       i ;

  do
    switch( y )
    {
      case 0:
                                  y = 1 ;
      break ;
      case 1:
        max = a[0] ; i = 1 ;
        if( i >= n )              y = 0 ;
        else
        if( a[i] > max )          y = 2 ;
        else                      y = 3 ;
      break ;
      case 2:
        max = a[i] ;              y = 3 ;
      break ;
      case 3:
        i++ ;
        if( i >= n )              y = 0 ;
        else
        if( a[i] > max )          y = 2 ;
      break ;
    }
  while( y != 0 ) ;
}
```

***Пример* 2.** По программе, приведенной в предыдущем примере, построить программу, соответствующую автомату Мили.

Используя пометки, указанные на рис. 17 в скобках, построим граф переходов автомата Мили с тремя состояниями (рис. 19).

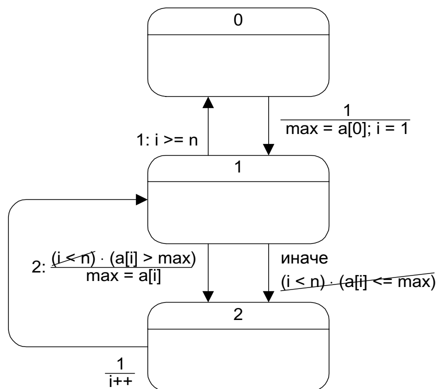


Рис. 19. Программа поиска максимума. Граф переходов автомата Мили

Используя этот граф переходов, построим автоматную программу требуемого типа (листинг 6).

ЛИСТИНГ 6. Программа поиска максимума в одномерном массиве, реализующая автомат Мили

```
void main()
{
  int       y = 0 ;
  enum      { n = 8 } ;
  int       a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int       max ;
  int       i ;

  do
    switch( y )
    {
      case 0:
        max = a[0] ; i = 1 ;    y = 1 ;
      break ;
      case 1:
        if( i >= n )            y = 0 ;
        else
        if( a[i] > max )
        { max = a[i] ;          y = 2 ; }
        else                    y = 2 ;
      break ;
      case 2:
        i++ ;                   y = 1 ;
      break ;
    }
  while( y != 0 ) ;
}
```

***Пример* 3.** Преобразуем схему программы на рис. 17 с целью получения автомата Мили с двумя состояниями и построим по этой схеме автоматную программу. Для этого продублируем операторную вершину с пометкой "i++" (рис. 20).

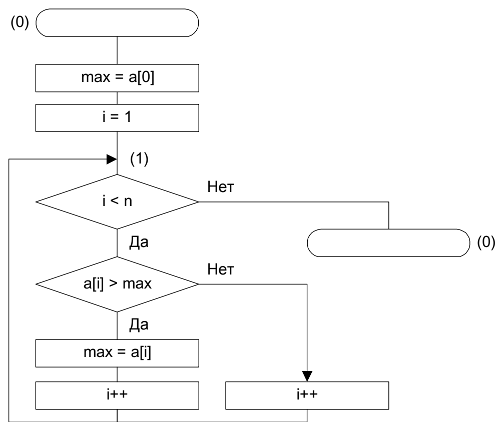


Рис. 20. Программа поиска максимума. Модифицированная схема программы

Схеме программы на рис. соответствует граф переходов автомата Мили с двумя состояниями (рис. 21).

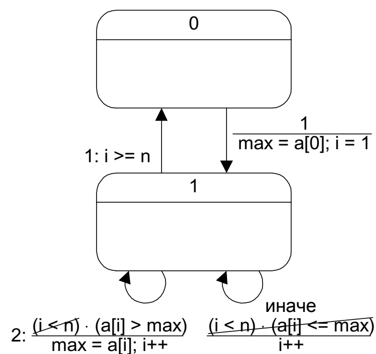


Рис. 21. Программа поиска максимума. Граф переходов автомата Мили

Приведем текст автоматной программы, реализующей этот граф переходов (листинг 7).

ЛИСТИНГ 7. Программа поиска максимума в одномерном массиве, реализующая автомат Мили

```
void main()
{
  int       y = 0 ;
  enum      { n = 8 } ;
  int       a[n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int       max ;
  int       i ;

  do
    switch( y )
    {
      case 0:
        max = a[0] ; i = 1 ;    y = 1 ;
      break ;
      case 1:
        if( i >= n )            y = 0 ;
        else
        if( a[i] > max )
        { max = a[i] ; i++ ; }
        else
          i++ ;
```

```
      break ;
    }
  while( y != 0 ) ;
}
```

Пример **4.** Задана программа поиска максимума в двумерном массиве (листинг 8), по которой надо построить программу, соответствующую автомату Мили.

ЛИСТИНГ 8. Программа поиска максимума в двумерном массиве

```
void main()
{
  enum      { m = 2, n = 4 } ;
  int       a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int       max = a[0][0] ;
  int       i, j ;

  for( i = 0 ; i < m ; i++ )
    for( j = 0 ; j < n ; j++ )
      if( a[i][j] > max )
        max = a[i][j] ;
}
```

Если по схеме этой программы построить граф переходов автомата Мили, то он будет содержать четыре вершины. Дублирование операторной вершины с пометкой "j++" приводит к построению схемы программы, приведенной на рис. 22.

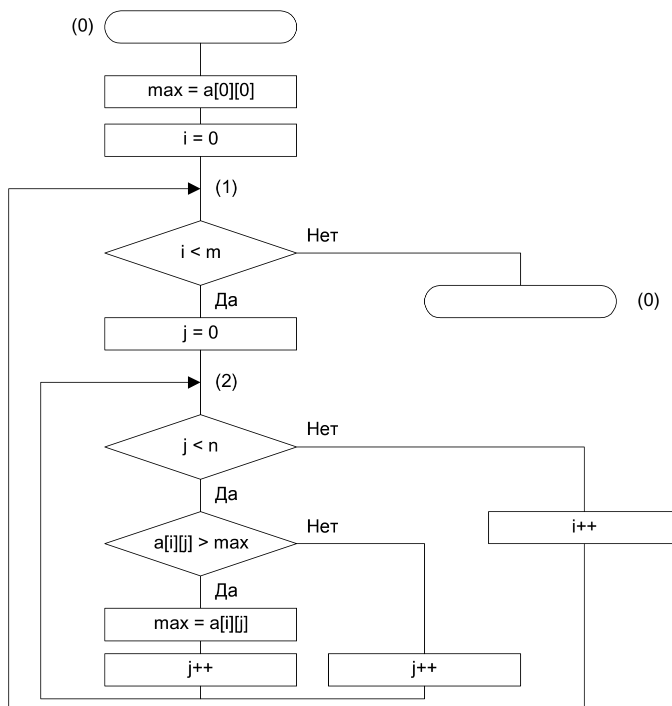


Рис. 22. Программа поиска максимума. Схема программы

Эта схема реализуется графом переходов автомата Мили с тремя состояниями (рис. 23).

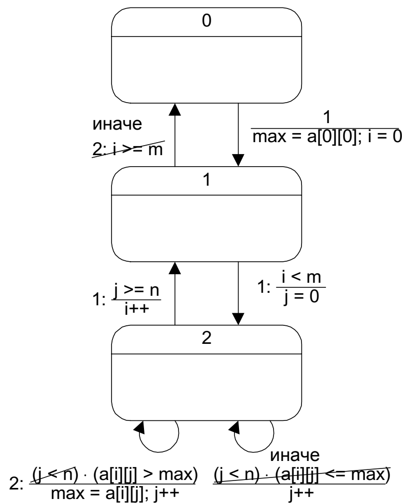


Рис. 23. Программа поиска максимума. Граф переходов автомата Мили

Приведем текст автоматной программы, построенной на основе этого графа переходов (листинг 9).

ЛИСТИНГ 9. Программа поиска максимума в двумерном массиве

```
void main()
{
  int       y = 0 ;
  enum      { m = 2, n = 4 } ;
  int       a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
  int       max = a[0][0] ;
  int       i, j ;

  do
    switch( y )
    {
      case 0:
        i = 0 ;                y = 1 ;
      break ;
      case 1:
        if( i < m )  { j = 0 ;  y = 2 ; }
        else                   y = 0 ;
      break ;
      case 2:
        if( j >= n ) { i++ ;    y = 1 ; }
        else
        if( a[i][j] > max )
        { max = a[i][j] ; j++ ; }
        else
          j++ ;
      break ;
    }
  while( y != 0 ) ;
}
```

***Пример* 5.** Автомат Мили, граф переходов которого приведен на рис. 23, можно декомпозировать на два автомата Мили, которые обозначим A0 и A1. Каждый из автоматов содержит по два состояния. При этом автомат A1 вложен в автомат A0 (рис. 24).

Рис. 24. Программа поиска максимума. Граф переходов автомата Мили

Эта система взаимосвязанных автоматов может быть реализована вложенными операторами switch. Однако "более изоморфной" является программа, в которой каждый автомат реализуется отдельной функцией (листинг 10).

ЛИСТИНГ 10. Программа поиска максимума в двумерном массиве

```
int         y0 = 0 ;
int         y1 = 0 ;
enum        { m = 2, n = 4 } ;
int         a[m][n] = { 44, 55, 12, 42, 94, 18, 6, 67 } ;
int         max ;
int         i, j ;

void main()
{
  max = a[0][0] ;
  do
    A0() ;
  while( y0 != 0 ) ;
}

void A0()
{
  switch( y0 )
  {
    case 0:
      i = 0 ;                 y0 = 1 ;
    break ;
    case 1:
      A1() ;
      if( y1 == 0 )  i++
      else
      if( i >= m )            y0 = 0 ;
    break ;
  }
}

void A1()
{
  switch( y1 )
  {
    case 0:
      j = 0 ;                 y1 = 1 ;
    break ;
    case 1:
      if( j >= n )            y1 = 0 ;
      else
      if( a[i][j] > max )
      { max = a[i][j] ; j++ ; }
```

```
      else
        j++ ;
    break ;
  }
}
```

Как отметил студент СПбГИТМО (ТУ) Макаров Н.С., алгоритм поиска максимума в массиве произвольной размерности может быть реализован автоматом Мили с двумя состояниями, если рассматривать массив любой размерности как одномерный.

Применяя предлагаемый подход для реализации алгоритма Дейкстры, предназначенного для поиска кратчайших расстояний в положительном взвешенном графе от вершины с номером ноль до всех остальных вершин, удается построенную традиционно структурированную программу, содержащую цикл for, в который вложены два таких же цикла, преобразовать в автоматную программу, состоящую из одной конструкции do-while, телом которой является конструкция switch, реализующая автомат Мили с четырьмя состояниями [17].

В заключение этого раздела работы отметим, что изложенный подход напоминает известный метод Ашкрофта-Манны [13], предназначенный для структурирования неструктурированных программ, который по этой причине (в отличие от предлагаемого) к структурированным программам не применяется.

2.3. Преобразование неструктурированных алгоритмов в автоматные программы

Предлагаемый подход при структурировании неструктурированных схем алгоритмов является более эффективным по сравнению с методом Ашкрофта-Манны ввиду явного использования автоматов разных типов и меньшего числа состояний в них. Продемонстрируем это на примерах.

Пример **6.** Построить по неструктурированной схеме алгоритма (рис. 25), приведенной в [16], автоматную схему.

Используя изложенный метод, построим граф переходов автомата Мили (рис. 26).

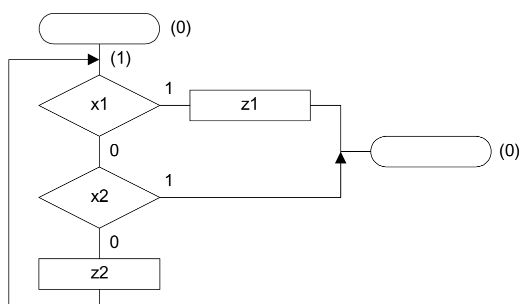


Рис. 25. Неструктурированная схема алгоритма

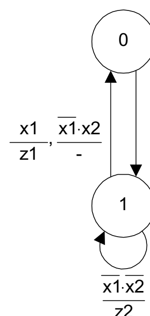


Рис. 26. Граф переходов автомата Мили

Автоматная схема алгоритма, являющаяся структурированной, которая изоморфно построена по этому графу, приведена на рис. 27.

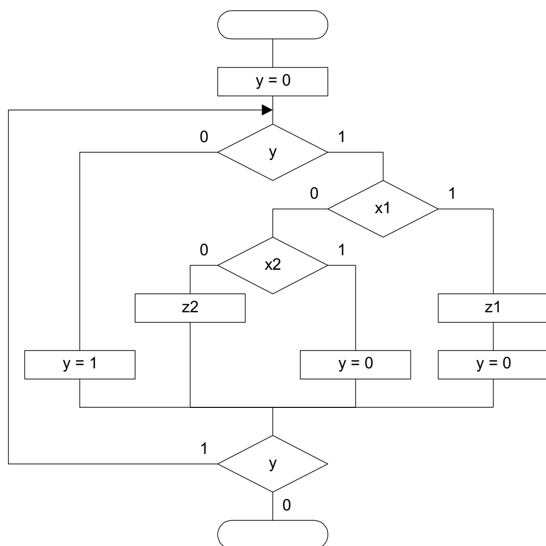


Рис. 27. Автоматная схема алгоритма

Эта схема содержит 10 условных и операторных вершин, в то время, как аналогичная схема, построенная методом Ашкрофта-Манны в [16], содержит 16 таких вершин (*У* = *О* = 2). Последняя схема может быть упрощена с

помощью перехода к рекурсивной структурированной схеме. При этом она будет содержать 8 вершин [16].

Эта схема, не являющаяся автоматной, может быть также получена и в результате упрощения схемы, представленной на рис. 27.

***Пример* 7.** Задана неструктурированная схема алгоритма (рис. 28). Построим графы переходов автоматов Мура (рис. 29) и Мили (рис. 30), которые изоморфно преобразуются в автоматные программы.

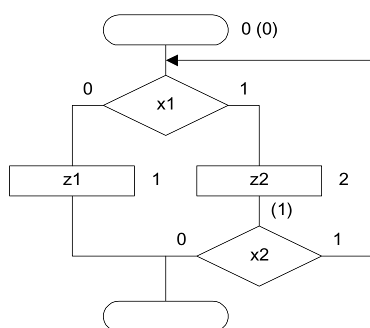


Рис. 28. Неструктурированная схема алгоритма

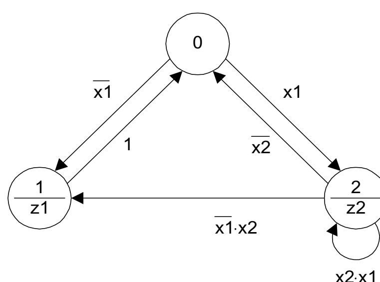


Рис. 29. Автомат Мура

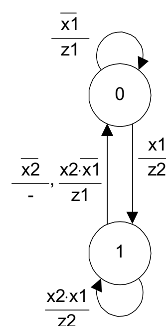


Рис. 30. Автомат Мили

***Пример* 8.** Задана неструктурированная схема алгоритма (рис. 31), в которой при x1 = x2 = 1 осуществляется задержка. Выполним на этом примере обоснование введения второго этапа в предлагаемый метод.

Применяя подход, изложенный в [9], построим графы переходов автоматов Мура (рис. 32) и Мили (рис. 33). Эти графы не могут быть использованы для построения автоматных программ. Так, например, автомат Мили имеет три петли, однократное исполнение двух из которых должно приводить к завершению алгоритма, а третья петля (с пометкой "x1x2") должна исполняться многократно. Однако, при принятой в настоящей работе реализации третья петля вместо многократного исполнения, будет также выполнена однократно с последующим завершением работы программы.

Полученный граф переходов автомата Мили обладает интересным свойством: он содержит только одну вершину, что характерно для автоматов без памяти. Однако, за счет умолчания выходных воздействий (прочерка) на одной из петель, этот граф описывает поведение автомата с памятью. Кроме того отметим, что и на других петлях, указаны не все выходные переменные.

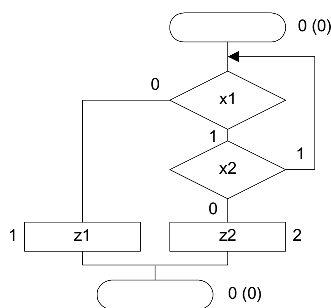


Рис. 31. Неструктурированная схема алгоритма

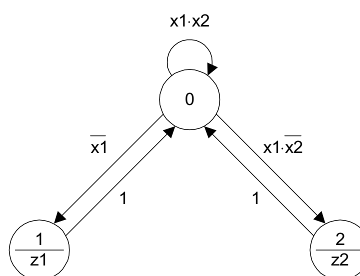


Рис. 32. Автомат Мура

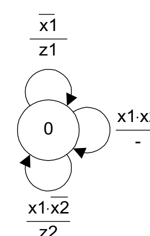


Рис. 33. Автомат Мили

Пример **9.** Задана неструктурированная схема алгоритма (рис. 31). Так как она содержит контур без операторных вершин, замыкающийся на дуге, исходящей из начальной вершины, то в соответствии со вторым этапом предлагаемого метода введем пометку для дополнительного состояния автоматов Мура и Мили сразу после начальной вершины схемы (рис. 34). Отметим, что эта пометка достаточно естественна для построения автомата Мили, но нетрадиционна для автомата Мура.

Построенные графы переходов автоматов Мура и Мили приведены на рис. 35 и 36, соответственно. По этим графам, как изложено выше, могут быть построены автоматные схемы алгоритмов и автоматные программы.

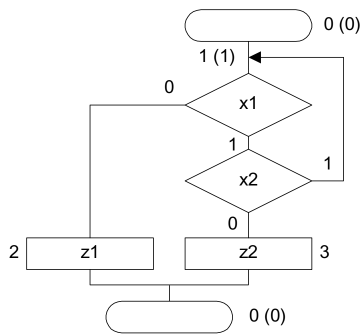


Рис. 34. Неструктурированная схема алгоритма

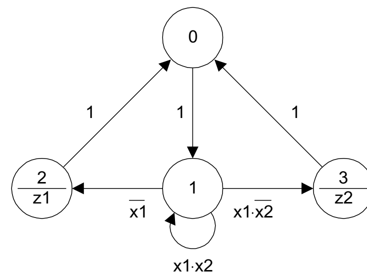


Рис. 35. Автомат Мура

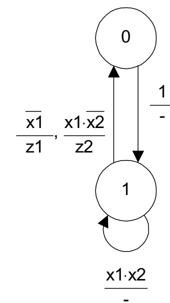


Рис. 36. Автомат Мили

***Пример* 10.** Задана неструктурированная схема алгоритма (рис. 37), в которой при x2 = x3 = 1 запоминаются значения выходных воздействий, вычисленные в первом блоке. В отличие от предыдущего примера, наличие "пустого" контура в схеме не требует введения дополнительного состояния для автомата Мили, так как этот контур замыкается не после начальной вершины. Граф переходов автомата Мили, построенный для этой схемы, приведен на рис. 38. По этому графу, как изложено выше, могут быть построены автоматная схема алгоритма и автоматная программа.

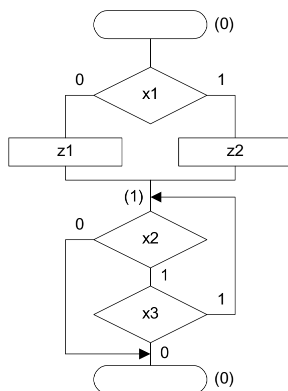


Рис. 37. Неструктурированная схема алгоритма

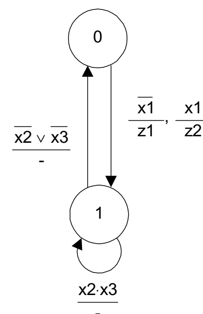


Рис. 38. Граф переходов автомата Мили

***Пример* 11.** Задана неструктурированная и непланарная схема алгоритма (рис. 39). Граф переходов автомата Мили, построенный по этой схеме и содержащий всего лишь два состояния, приведен на рис. 40. По этому графу, как

изложено выше, могут быть построены автоматная схема алгоритма и автоматная программа.

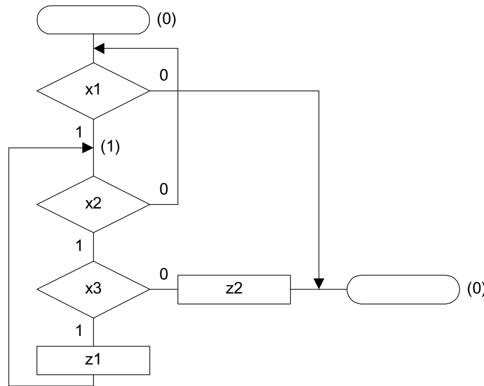


Рис. 39. Неструктурированная схема алгоритма

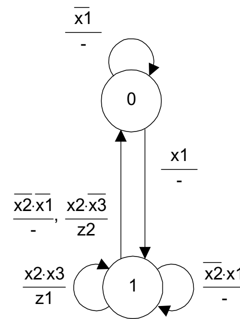
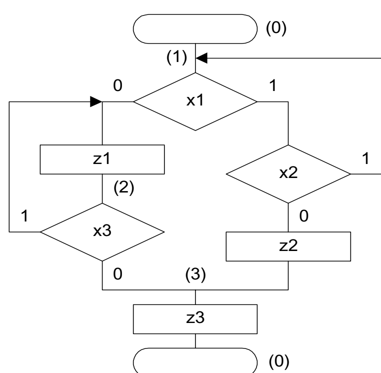
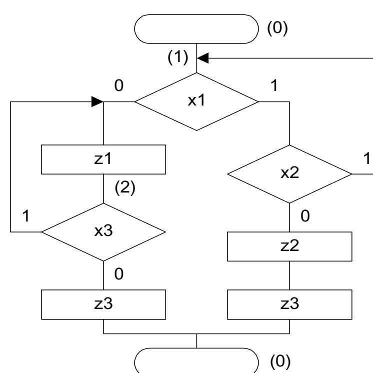


Рис. 40. Граф переходов автомата Мили

Рассмотренная схема содержит контур, замыкающийся после начальной вершины и не содержащий операторных вершин. Так как этот контур содержит пометку "(1)", соответствующую состоянию автомата Мили, то вводить дополнительное состояние не требуется.

***Пример* 12.** Задана неструктурированная схема алгоритма, приведенная в [14] и представленная на рис. 41. Введя пометку для дополнительного состояния в соответствии со вторым этапом предлагаемого метода, можно построить граф переходов автомата Мили с четырьмя состояниями. Продублировав операторную вершину с пометкой "z3" в соответствии с третьим этапом метода (рис. 42), построим граф переходов автомата Мили с тремя состояниями (рис. 43).





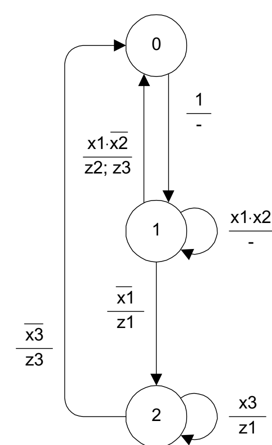


Рис. 41. Неструктурированная схема алгоритма

Рис. 42. Автомат Мура

Рис. 43. Автомат Мили

Автоматная схема алгоритма, изоморфная этому графу, приведена на рис. 44.

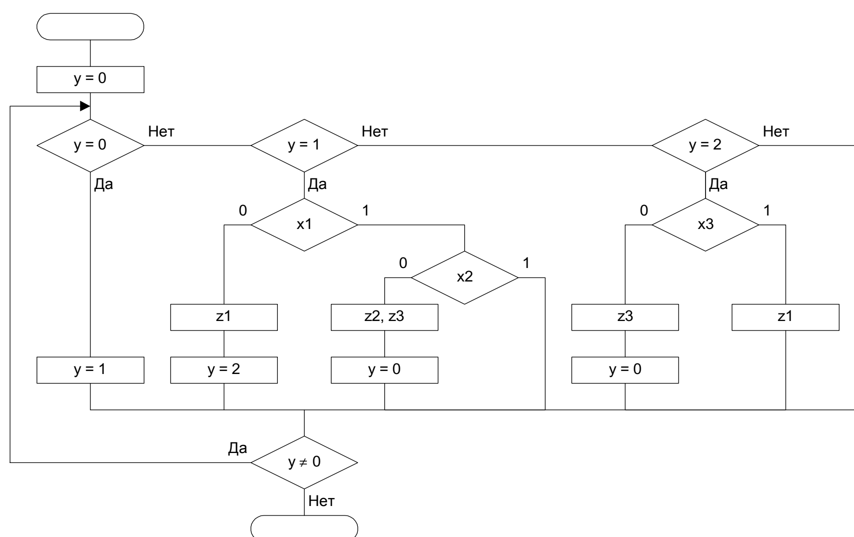


Рис. 44. Автоматная схема алгоритма

Построенная структурированная схема алгоритма содержит 16 условных и операторных вершин, в то время, как аналогичная схема, построенная методом Ашкрофта-Манны в [14] (с учетом исправления допущенной там ошибки) содержит 23 таких вершины (*У* = *О* = 3).

Приведенная на рис. 44 схема соответствует предлагаемой программной реализации с использованием операторов do-while и switch.

Все рассмотренные в примерах схемы алгоритмов и программ имели начальную и конечную вершины, что

характерно для вычислительных алгоритмов. Предложенный метод может быть использован также и для преобразования алгоритмов, содержащих бесконечный внешний цикл. При этом граф переходов, построенный по такой схеме, будет обладать свойством недостижимости начальной вершины.

3. Преобразование программ с явной рекурсией в автоматные

В предыдущем разделе был предложен метод преобразования произвольных итеративных программ в автоматные программы [6]. Этот метод позволяет реализовать произвольный итеративный алгоритм структурированной программой, содержащей один оператор do-while, телом которого является один оператор switch.

В работах [4,19] приведены примеры преобразований программ с концевой рекурсией в итеративные, однако эти преобразования выполнялись неформально в связи с отсутствием соответствующего метода.

В настоящей работе такой метод предлагается. Он состоит в преобразовании заданной программы с явной рекурсией в итеративную программу, построенную с использованием автомата Мили. Как и ранее, будем называть программы построенные таким образом автоматными. Метод иллюстрируется примерами преобразований классических рекурсивных программ, которые приведены в следующих разделах в порядке их усложнения (факториал, числа Фибоначчи, задача о ранце, ханойские башни). При этом для задачи о ханойских башнях кроме предлагаемого метода преобразования рассматриваются и другие подходы к преобразованию рекурсивных алгоритмов в автоматные.

3.1. Изложение метода

Идея метода состоит в моделировании работы рекурсивной программы автоматной программой, явно (также, как и в работах [4,19]) использующей стек. Отметим, что явное выделение стека по сравнению со "скрытым" его применением в рекурсии, позволяет программно задавать его размер, следить за его содержимым и добавлять отладочный код в функции, реализующие операции над стеком.

Перейдем к изложению метода.

1. Каждый рекурсивный вызов в программе выделяется как отдельный оператор. По преобразованной рекурсивной программе строится ее схема, в которой применяются символьные обозначения условий переходов (x), действий (z) и рекурсивных вызовов (R). Схема строится таким образом, чтобы каждому рекурсивному вызову соответствовала отдельная операторная вершина. Отметим, что здесь и далее под термином "схема программы" понимается схема ее рекурсивной функции.

2. Для определения состояний эквивалентного построенной схеме автомата Мили в нее вводятся пометки, по аналогии с тем, как это выполнялось в предыдущем разделе при преобразовании итеративных программ в автоматные. При этом начальная и конечная вершины помечаются номером 0. Точка, следующая за начальной вершиной, помечается номером 1, а точка, предшествующая конечной вершине — номером 2. Остальным состояниям автомата соответствуют точки, следующие за операторными вершинами.

3. Используя пометки в качестве состояний автомата Мили, строится его граф переходов.

4. Выделяются действия, совершаемые над параметрами рекурсивной функции при ее вызове.

5. Составляется перечень параметров и других локальных переменных рекурсивной функции, определяющий структуру ячейки стека. Если рекурсивная функция содержит более одного оператора рекурсивного вызова, то в стеке также запоминается значение переменной состояния автомата.

6. Выполняется преобразование графа переходов для моделирования работы рекурсивной функции, состоящее из трех этапов.

6.1. Дуги, содержащие рекурсивные вызовы (R), направляются в вершину с номером 1. В качестве действий на этих дугах указываются: операция запоминания значений локальных переменных push(s), где s — номер вершины графа переходов, в которую была направлена рассматриваемая дуга до преобразования; соответствующее действие, выполняемое над параметрами рекурсивной функции.

6.2. Безусловный переход на дуге, направленной из вершины с номером 2 в вершину с номером 0, заменяется условием "стек пуст".

6.3. К вершине с номером 2 добавляются дуги, направленные в вершины, в которые до преобразания графа входили дуги с рекурсией. На каждой из введеных дуг выполняется операция pop, извлекающая из стека верхний элемент. Условия переходов на этих дугах имеют вид stack[top].y == S, где stack[top] — верхний элемент стека, y — значение переменной состояния автомата, запомненное в верхнем элементе стека, а S — номер вершины, в которую направлена рассматриваемая дуга. Таким образом, в рассматриваемом автомате имеет место **зависимость от глубокой предыстории** — в отличие от классических автоматов, переходы из состояния с номером 2

зависят также и от ранее запомненного в стеке номера следующего состояния.

7. Граф переходов может быть упрощен за счет исключения неустойчивых вершин (кроме вершины с номером 0).

8. Строится автоматная программа, содержащая функции для работы со стеком и цикл do-while, телом которого является оператор switch, формально и изоморфно построенный по графу переходов.

В заключение изложения метода отметим, что для преобразования рекурсивных программ в автоматные может быть предложена аналогичная методика, базирующаяся на применении автоматов Мура вместо автоматов Мили. Однако эта методика и получаемые в результате графы переходов являются более громоздкими.

3.2. Вычисление факториала

Одной из простейших задач, использующих рекурсию, является вычисление факториала. Построим автоматную программу по рекурсивной программе, в которой рекурсивный вызов выделен отдельным оператором (листинг 11).

ЛИСТИНГ 11. Рекурсивная программа вычисления факториала

```
#include <stdio.h>

int     f = 0 ;

void fact( int n )
{
  if( n <= 1 )
    f = 1 ;
  else
  {
    fact( n-1 ) ;
    f = n * f ;
  }
}

void main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nÔàêòîðèàë(%d) = %d\n", input, f ) ;
```

}

Построим схему этой программы (рис. 45), а по ней — граф переходов автомата Мили (рис. 46).

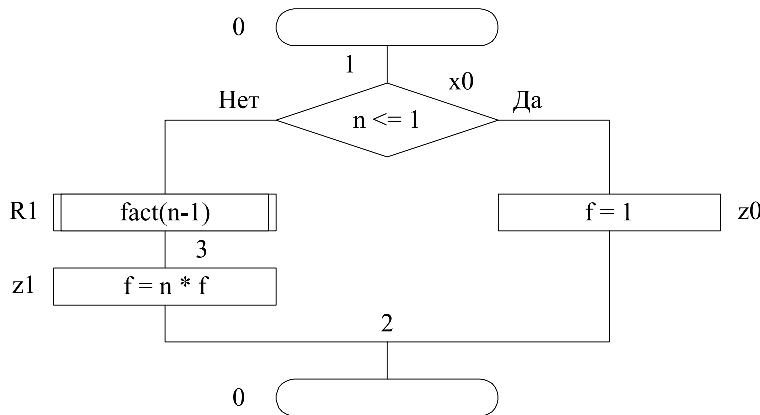


Рис. 45. Схема программы вычисления факториала

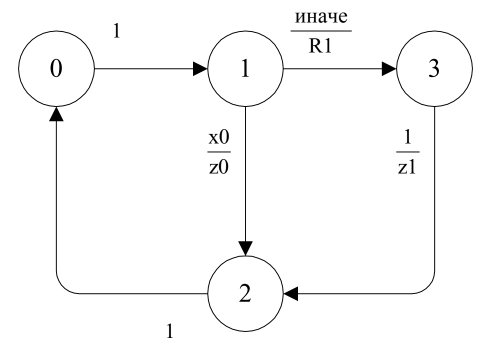


Рис. 46. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга 1—3 направляется в вершину с номером 1 и превращается в петлю. Условие перехода на этой петле остается неизменным, а рекурсивный вызов заменяется на операцию push и действие (n--), выполняемое над параметром рекурсивной функции при ее вызове. Это моделирует работу рекурсивной программы: при рекурсивном вызове значения локальных переменных запоминаются, а выполнение рекурсивной функции начинается сначала — с точки, соответствующей вершине с номером 1 графа переходов. Безусловный переход на дуге 2—0 заменяется на условие "стек пуст". Добавляется дуга 2—3, при переходе по которой выполняется действие pop. Эта дуга соответствует возврату из рекурсивной функции, при котором восстанавливаются значения локальных переменных и выполнение функции продолжается из точки, следующей за

оператором рекурсивного вызова. Эта точка соответствует вершине с номером 3 графа переходов (рис. 47).

Так как исходная программа содержит только одну рекурсию, запоминать значение переменной состояния автомата в данном случае нет необходимости. Поэтому операции push и pop будут сохранять и восстанавливать только значение локальной переменной n.

Упростим этот граф за счет исключения неустойчивой вершины с номером 3 (рис. 48).

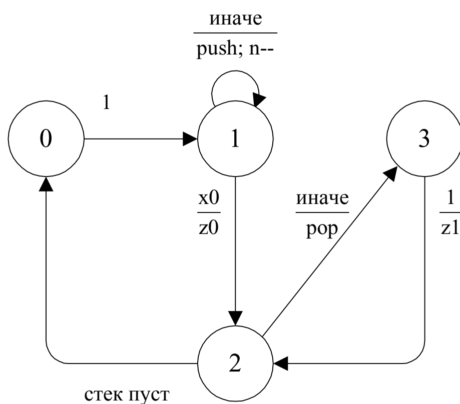


Рис. 47. Преобразованный граф переходов

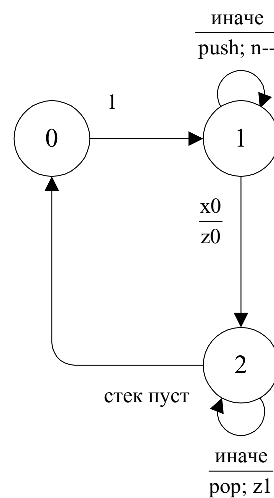


Рис. 48. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 12.

ЛИСТИНГ 12. Автоматная программа вычисления факториала

```
#include <stdio.h>

typedef struct
{
  int n ;
} stack_t ;

stack_t  stack[200] ;
int      top = -1 ;

push( int n )
{
  top++ ;
  stack[top].n = n ;
```

```
  printf( "push {%d}: ", n ) ; show_stack() ;
}

pop( int *n )
{
  printf( "pop  {%d}: ", stack[top].n ) ;
  *n = stack[top].n ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;
  for( i = top ; i >= 0 ; i-- )
    printf( "{%d} ", stack[i].n ) ;
  printf( "\n" ) ;
}

int     f = 0 ;

void fact( int n )
{
  int y = 0, y_old ;

  do
  {
    y_old = y ;

    switch( y )
    {
      case 0:
                                   y = 1 ;
      break ;

      case 1:
        if( n <= 1 ) { f = 1 ;   y = 2 ; }
        else
        { push( n ) ; n-- ; }
      break ;

      case 2:
        if( stack_empty() )      y = 0 ;
        else
        { pop( &n ) ; f = n * f ; }
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n", y ) ;
  } while( y != 0 ) ;
}

void main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nФакториал(%d) = %d\n", input, f ) ;
}
```

Протокол, отражающий работу со стеком, приведен в листинге 13.

ЛИСТИНГ 13. Протокол, отражающий работу со стеком при вычислении факториала

```
Переход в состояние 1
```

```
push {7}: {7}
push {6}: {6} {7}
push {5}: {5} {6} {7}
push {4}: {4} {5} {6} {7}
push {3}: {3} {4} {5} {6} {7}
push {2}: {2} {3} {4} {5} {6} {7}
Переход в состояние 2
pop  {2}: {3} {4} {5} {6} {7}
pop  {3}: {4} {5} {6} {7}
pop  {4}: {5} {6} {7}
pop  {5}: {6} {7}
pop  {6}: {7}
pop  {7}:
Переход в состояние 0

Факториал(7) = 5040
```

3.3. Вычисление чисел Фибоначчи

Более сложной задачей является вычисление чисел Фибоначчи, программа решения которой содержит две рекурсии, выделенные в виде отдельных операторов (листинг 14).

ЛИСТИНГ 14. Рекурсивная программа вычисления чисел Фибоначчи

```
#include <stdio.h>

int res ;

void fibo( int n )
{
  int f ;

  if( n <= 1 )
    res = n ;
  else
  {
    fibo( n-1 ) ;
    f = res ;
    fibo( n-2 ) ;
    res += f ;
  }
}

void main()
{
  int input = 30 ;
  fibo( input ) ;
  printf( "\nФибоначи(%d) = %d\n", input, res ) ;
}
```

Отметим, что приведенная программа весьма неэффективна ввиду повторного вычисления одних и тех же значений. Эта программа может быть усовершенствована с помощью динамического программирования [20]. Построим

автоматную программу по приведенной выше рекурсивной программе.

Построим схему этой программы (рис. 49), а по ней — граф переходов автомата Мили (рис. 50).

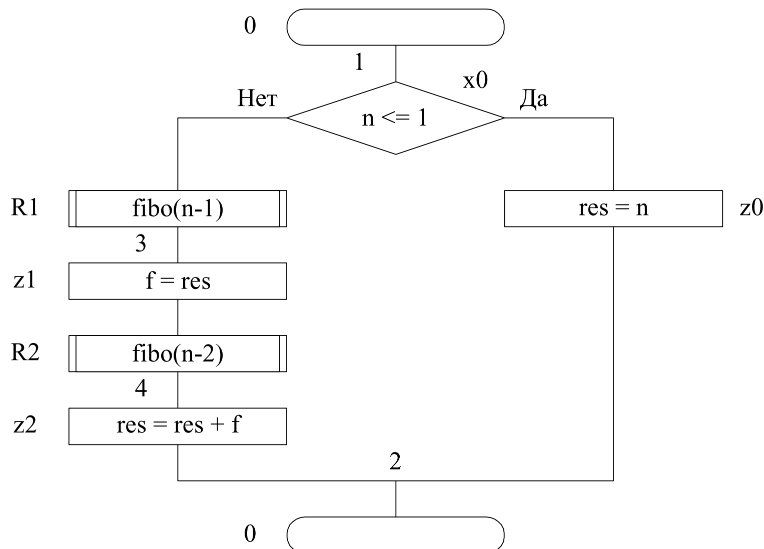


Рис. 49. Схема программы вычисления чисел Фибоначчи

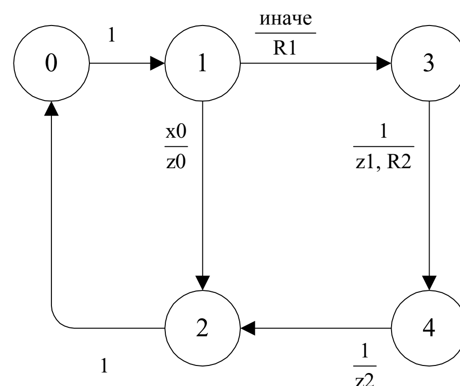


Рис. 50. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга 1—3 направляется в вершину с номером 1, а рекурсивный вызов R1 заменяется на операцию push(3) и действие (n--), выполняемое над параметром рекурсивной функции при ее вызове. Аналогичным образом корректируется дуга 3—4, содержащая рекурсивный вызов R2. Эта дуга направляется в вершину с номером 1, а рекурсивный вызов заменяется на операцию push(4) и действие (n = n-2), выполняемое над параметром рекурсивной функции при ее вызове. Безусловный переход на дуге 2—0 заменяется на условие "стек пуст", и ему присваивается первый приоритет. Добавляются дуги 2—3 и 2—4, переход по которым зависит от значения переменной состояния автомата, запомненного в верхнем элементе стека. Если это значение равно "3", осуществляется

переход по дуге 2—3, а если это значение равно "4" — то по дуге 2—4. При переходе по этим дугам выполняется действие pop (рис. 51).

Дуги 2—3 и 2—4 соответствуют возврату из рекурсивной функции. Так как исходная программа содержит две рекурсии (две точки возврата из рекурсивных вызовов), необходимо запоминать в стеке не только локальные переменные n и f, но и значение переменной состояния автомата y, соответствующее точке, в которую необходимо осуществить возврат из рекурсивного вызова.

Упростим этот граф за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 52).

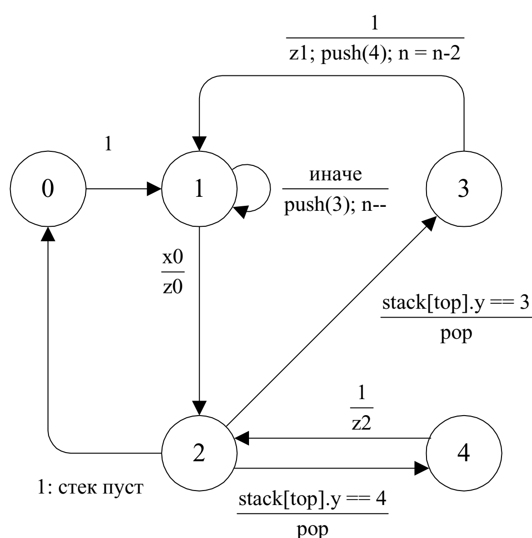


Рис. 51. Преобразованный граф переходов

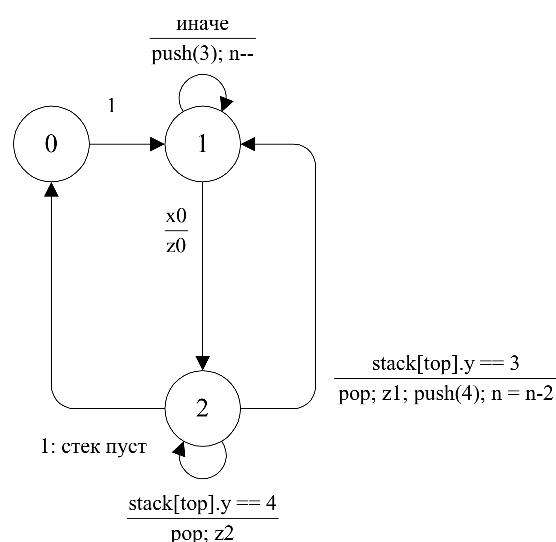


Рис. 52. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 15.

ЛИСТИНГ 15. Автоматная программа вычисления чисел Фибоначчи

```
#include <stdio.h>

typedef struct
{
  int y, n, f ;
} stack_t ;

stack_t  stack[200] ;
int      top = -1 ;
```

```
push( int y, int n, int f )
{
  top++ ;
  stack[top].y = y ;
  stack[top].n = n ;
  stack[top].f = f ;
  printf( "push {%d,%d,%d}: ", y, n, f ) ; show_stack() ;
}

pop( int *y, int *n, int *f )
{
  printf( "pop  {%d,%d,%d}: ", stack[top].y, stack[top].n, stack[top].f ) ;
  if( y ) *y = stack[top].y ;
  *n = stack[top].n ;
  *f = stack[top].f ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;

  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d} ", stack[i].y, stack[i].n, stack[i].f ) ;
  printf( "\n" ) ;
}

int     res = 0 ;

void fibo( int n )
{
  int f ;
  int y = 0, y_old ;

  do
  {
    y_old = y ;

    switch( y )
    {
      case 0:
                                         y = 1 ;
      break ;

      case 1:
        if( n <= 1 ) { res = n ;         y = 2 ; }
        else
        { push( 3, n, f ) ; n-- ; }
      break ;

      case 2:
        if( stack_empty() )              y = 0 ;
        else
        if( stack[top].y == 3 )
        {
          pop( NULL, &n, &f ) ;
          f = res ;
          push( 4, n, f ) ; n = n - 2 ;
                                         y = 1 ;
        }
        else
        if( stack[top].y == 4 )
        {
          pop( NULL, &n, &f ) ;
          res = res + f ;
        }
```

```
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n", y ) ;
  } while( y != 0 ) ;
}

void main()
{
  int input = 4 ;
  fibo( input ) ;
  printf( "\nФибоначчи(%d) = %d\n", input, res ) ;
}
```

Протокол, отражающий работу со стеком, приведен в листинге 16.

ЛИСТИНГ 16. Протокол, отражающий работу со стеком при вычислении чисел Фибоначчи

```
Переход в состояние 1
push {3,4,0}: {3,4,0}
push {3,3,0}: {3,3,0} {3,4,0}
push {3,2,0}: {3,2,0} {3,3,0} {3,4,0}
Переход в состояние 2
pop  {3,2,0}: {3,3,0} {3,4,0}
push {4,2,1}: {4,2,1} {3,3,0} {3,4,0}
Переход в состояние 1
Переход в состояние 2
pop  {4,2,1}: {3,3,0} {3,4,0}
pop  {3,3,0}: {3,4,0}
push {4,3,1}: {4,3,1} {3,4,0}
Переход в состояние 1
Переход в состояние 2
pop  {4,3,1}: {3,4,0}
pop  {3,4,0}:
push {4,4,2}: {4,4,2}
Переход в состояние 1
push {3,2,2}: {3,2,2} {4,4,2}
Переход в состояние 2
pop  {3,2,2}: {4,4,2}
push {4,2,1}: {4,2,1} {4,4,2}
Переход в состояние 1
Переход в состояние 2
pop  {4,2,1}: {4,4,2}
pop  {4,4,2}:
Переход в состояние 0

Фибоначчи(4) = 3
```

3.4. Задача о ранце

Задача о ранце может быть сформулирована следующим образом. Имеется N типов предметов, для каждого из которых известны его объем и цена, причем количество предметов каждого типа неограничено. Необходимо уложить предметы в ранец объема M таким образом, чтобы стоимость его содержимого была максимальна.

Как и в случае задачи о вычислении чисел Фибоначчи, непосредственное рекурсивное решение этой задачи является крайне неэффективным из-за повторного решения одних и тех же подзадач [20]. Этот недостаток может быть устранен путем применения динамического программирования. При этом промежуточные результаты решения задачи запоминаются и используются в дальнейшем.

Несмотря на то, что решение задачи содержит только одну рекурсию (листинг 17), эта задача рассматривается в настоящей работе для иллюстрации предлагаемого метода, так как соответствующая ей программа, построенная на основе программы, предложенной в работе [20], отличается весьма сложной логикой по сравнению с программами, приведенными выше. Также, как и в приведенных выше примерах, в рассматриваемой программе рекурсивный вызов выделен отдельным оператором.

ЛИСТИНГ 17. Рекурсивная программа решения задачи о ранце с использованием динамического программирования

```
#include <stdio.h>

typedef struct
{
  int size, val ;
} item_t ;

item_t  items[] = { 3,4, 4,5, 7,10, 8,11, 9,13 } ;  // Типы предметов.
int     N = sizeof(items)/sizeof(item_t) ;          // Количество типов предметов.
enum    { knap_cap = 17 } ;                         // Объем ранца.
int     maxKnown[knap_cap+1] ;
item_t  itemKnown[knap_cap+1] ;
int     knap_ret = 0 ;

void knap( int cap )
{
  int i, space, max, maxi = 0, t ;

  if( maxKnown[cap] != -1 )
  {
    knap_ret = maxKnown[cap] ;
    return ;
  }

  for( i = 0, max = 0 ; i < N ; i++ )
  {
    space = cap - items[i].size ;
    if( space >= 0 )
    {
      knap( space ) ;
      t = knap_ret + items[i].val ;
```

```
      if( t > max )
      {
        max = t ;
        maxi = i ;
      }
    }
  }

  maxKnown[cap] = max ; itemKnown[cap] = items[maxi] ;
  knap_ret = max ;
}

void show_knap( int cap, int value )
{
  int i ;
  int total_size = 0 ;

  printf( "\nВиды предметов {объем, цена} (%d):\n", N ) ;
  for ( i = 0 ; i < N ; i++ )
    printf( "{%d,%d} ", items[i].size, items[i].val ) ;

  printf( "\n\nОбъем ранца: %d.\n", cap ) ;
  printf( "Содержимое ранца:\n" ) ;

  i = cap ;
  while( i > 0 && i - itemKnown[i].size >= 0 )
  {
    printf( "{%d,%d} ", itemKnown[i].size, itemKnown[i].val ) ;
    total_size += itemKnown[i].size ;
    i -= itemKnown[i].size ;
  }
  printf( "\nЗанятый объем: %d. Ценность: %d\n", total_size, value ) ;
}

void init()
{
  int i ;

  for ( i = 0 ; i < knap_cap+1 ; i++ )
    maxKnown[i] = -1 ;
}

void main()
{
  init() ;
  knap( knap_cap ) ;
  show_knap( knap_cap, knap_ret ) ;
}
```

В приведенной программе массив itemKnown применяется для определения содержимого ранца, полученного в результате решения задачи.

С целью обеспечения сокращения перебора на основе метода динамического программирования массив maxKnown используется для запоминания результатов решения подзадач. Для иллюстрации процесса перебора, функция knap() дополнительно содержит код, позволяющий построить дерево рекурсивных вызовов (рис. 53). В этом дереве прямоугольниками помечены вершины, соответствующие

подзадачам, требующим решения, а кружками — подзадачи, результат решения которых уже известен и содержится в массиве maxKnown. Числа в вершинах означают объем ранца для соответствующей подзадачи.

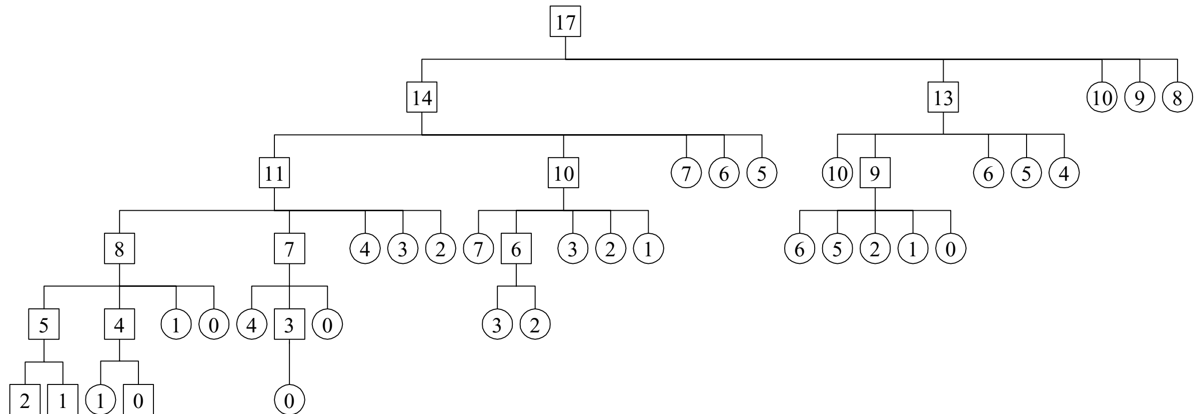


Рис. 53. Дерево рекурсивных вызовов при решении задачи о ранце

Содержимое массива maxKnown, хранящего результаты решения подзадач для ранцев разного объема, приведено в табл. 1. Значение "-1" означает, что решение подзадачи для указанного объема ранца не требовалось.

ТАБЛИЦА 1. *Содержимое массива* maxKnown

cap	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
maxKnown[cap]	0	0	0	4	5	5	8	10	11	13	14	15	-1	18	20	-1	-1	24

Проиллюстрируем особенности применения динамического программирования, частично рассмотрев процесс перебора. Перебор начинается с корня дерева, которое строится сверху вниз и слева направо. Количество подзадач, порождаемых на каждом уровне дерева (рис. 53), равно числу типов предметов, помещающихся в ранец. Например, второй уровень дерева получается в результате размещения в ранце предметов каждого типа по очереди. При размещении в ранце предметов объемом 3 и 4 выполняется дальнейшее решение подзадач для ранцев объемом 14 и 13, а для предметов объемом 7, 8 и 9 получаемые подзадачи для ранцев объемом 10, 9 и 8 оказываются уже решенными.

Отметим, что если динамическое программирование не использовать, дерево рекурсивных вызовов значительно увеличивается, так как все вершины, обозначенные кружками, заменяются поддеревьями, содержащими полный процесс перебора, необходимый для решения соответствующих подзадач [20].

Построим схему рекурсивной программы (рис. 54), в которой не учитывается код, использованный для построения дерева рекурсивных вызовов.

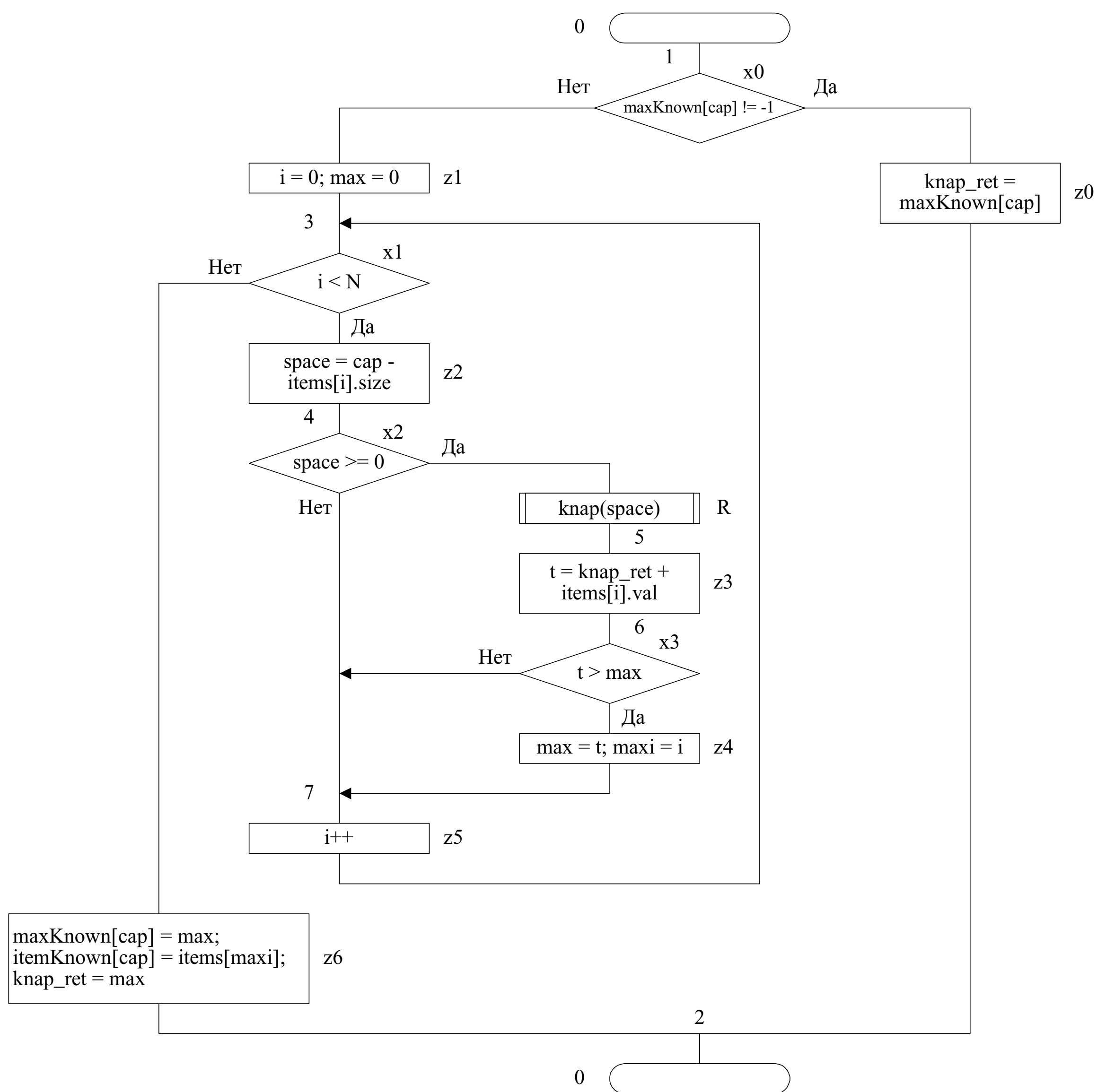


Рис. 54. Схема программы решения задачи о ранце

По схеме программы (рис. 54) построим граф переходов автомата Мили (рис. 55).

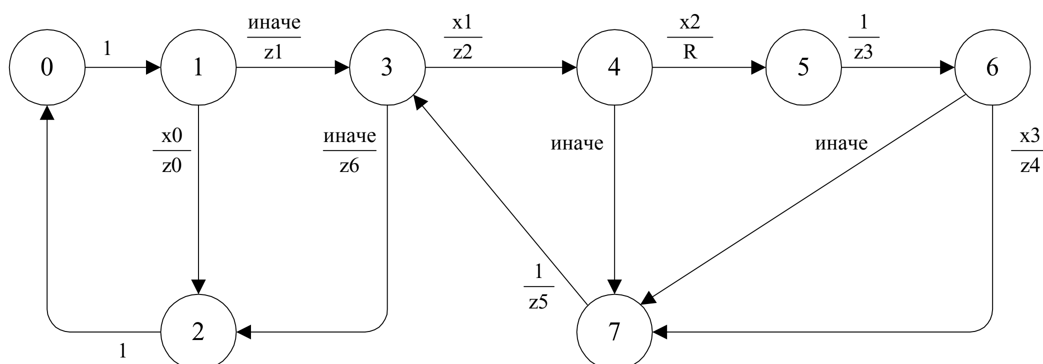


Рис. 55. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. При этом дуга 4—5 направляется в вершину с номером 1, а выполняемый при переходе по этой дуге рекурсивный вызов заменяется на операцию **push** и действие (**cap** = **space**), выполняемое над параметром рекурсивной функции при ее вызове. Безусловный переход на дуге 2—0 заменяется на условие "стек пуст". Добавляется дуга 2—5, при переходе по которой выполняется действие **pop** (рис. 56).

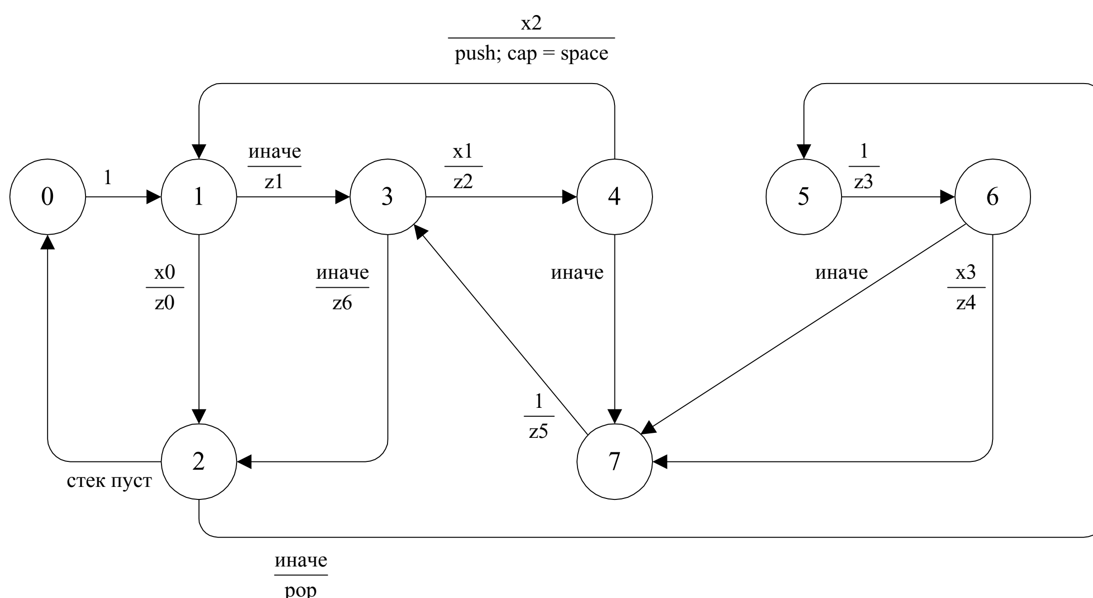


Рис. 56. Преобразованный граф переходов

Упростим этот граф за счет исключения неустойчивой вершины с номером 5 (рис. 57).

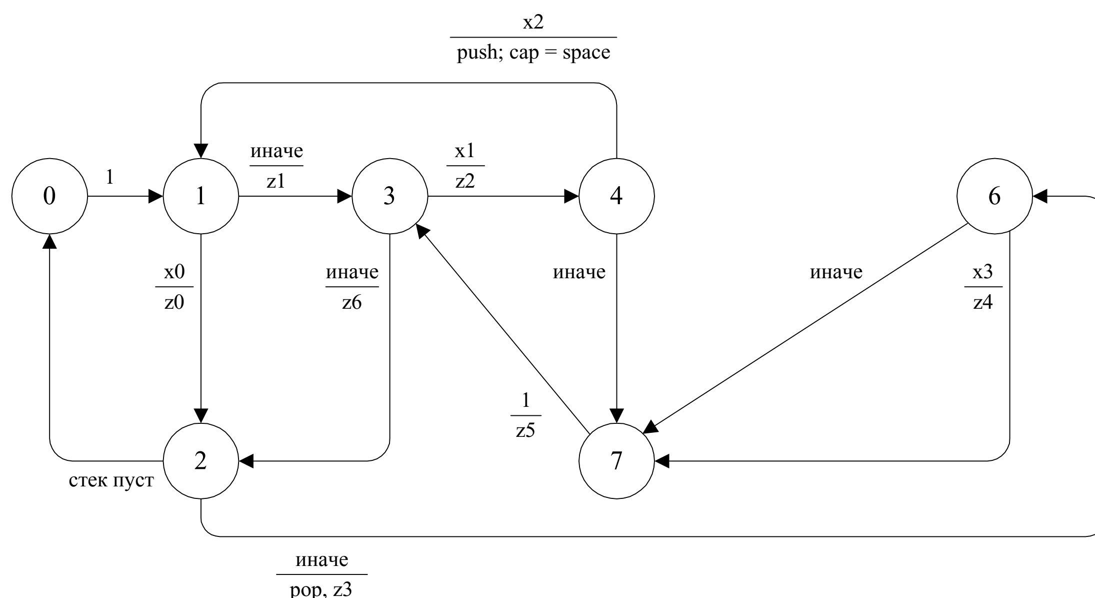


Рис. 57. Упрощенный граф переходов

Фрагмент автоматной программы, построенной по графу переходов автомата Мили (рис. 57), приведен в листинге 18. В этом фрагменте отсутствуют функции `show_knap()`, `init()` и `main()`, так как они не изменяются по сравнению с предыдущим листингом.

ЛИСТИНГ 18. Фрагмент автоматной программы решения задачи о ранце

```
#include <stdio.h>

typedef struct
{
  int cap, i, space, max, maxi, t ;
} stack_t ;

stack_t  stack[200] ;
int      top = -1 ;

push( int cap, int i, int space, int max, int maxi, int t )
{
  top++ ;
  stack[top].cap = cap ;
  stack[top].i = i ;
  stack[top].space = space ;
  stack[top].max = max ;
  stack[top].maxi = maxi ;
  stack[top].t = t ;
  printf( "push {%d,%d,%d,%d,%d,%d}: ", cap, i,
          space, max, maxi, t ) ;
  show_stack() ;
}

pop( int *cap, int *i, int *space, int *max, int *maxi, int *t )
{
  printf( "pop  {%d,%d,%d,%d,%d,%d}: ", stack[top].cap, stack[top].i,
          stack[top].space, stack[top].max, stack[top].maxi, stack[top].t ) ;
  *cap = stack[top].cap ;
  *i = stack[top].i ;
  *space = stack[top].space ;
```

```
  *max = stack[top].max ;
  *maxi = stack[top].maxi ;
  *t = stack[top].t ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;

  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d,%d,%d,%d} ", stack[i].cap, stack[i].i,
            stack[i].space, stack[i].max, stack[i].maxi, stack[i].t ) ;
  printf( "\n" ) ;
}

typedef struct
{
  int size, val ;
} item_t ;

item_t  items[] = { 3,4, 4,5, 7,10, 8,11, 9,13 } ;
int     N = sizeof(items)/sizeof(item_t) ;
enum    { knap_cap = 17 } ;
int     maxKnown[knap_cap+1] ;
item_t  itemKnown[knap_cap+1] ;
int     knap_ret = 0 ;

void knap( int cap )
{
  int y = 0, y_old ;
  int i, space, max, maxi = 0, t ;

  do
  {
    y_old = y ;

    switch( y )
    {
      case 0:
                                          y = 1 ;
      break ;

      case 1:
        if( maxKnown[cap] != -1 )
        { knap_ret = maxKnown[cap] ;       y = 2 ; }
        else
        { i = 0 ; max = 0 ;                y = 3 ; }
      break ;

      case 2:
        if( stack_empty() )                y = 0 ;
        else
        {
          pop( &cap, &i, &space, &max, &maxi, &t ) ;
          t = knap_ret + items[i].val ;    y = 6 ;
        }
      break ;

      case 3:
        if( i < N )
        { space = cap - items[i].size ;    y = 4 ; }
        else
        {
          maxKnown[cap] = max ;
          itemKnown[cap] = items[maxi] ;
          knap_ret = max ;                 y = 2 ;
```

```
      }
    break ;

    case 4:
      if( space >= 0 )
      {
        push( cap, i, space, max, maxi, t ) ;
        cap = space ;
                                  y = 1 ;
      }
      else
                                  y = 7 ;
    break ;

    case 6:
      //if( t >= max ) // Фиксируется последнее из оптимальных решений.
      if( t > max ) // Фиксируется первое из оптимальных решений.
      { max = t ; maxi = i ;      y = 7 ; }
      else
                                  y = 7 ;
    break ;

    case 7:
      i++ ;                       y = 3 ;
    break ;
  } ;

  if( y_old != y ) printf( "Переход в состояние %d\n", y ) ;
 } while( y != 0 ) ;
}
```

Результат работы этой программы, идентичный результату работы рекурсивной программы, приведен в листинге 19.

ЛИСТИНГ 19. Результат работы автоматной программы решения задачи о ранце

```
Виды предметов {объем, цена} (5):
{3,4} {4,5} {7,10} {8,11} {9,13}

Объем ранца: 17.
Содержимое ранца:
{3,4} {7,10} {7,10}
Занятый объем: 17. Ценность: 24
```

3.5. Задача о ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [20], которая формулируется следующим образом. Имеются три стержня, на первом из которых размещено N дисков. Диск наименьшего диаметра находится сверху, а ниже — диски последовательно увеличивающегося диаметра. Задача состоит в перекладывании по одному диску со стержня на стержень

так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы, в конце концов, все диски оказались на другом стержне.

Рассмотрим три подхода, применение автоматов в которых позволяет либо формально переходить к итеративным алгоритмам решения этой задачи, либо строить такие алгоритмы непосредственно.

Первый из них заключается в использовании рассмотренного выше метода и обеспечивает построение автоматной программы на основе раскрытия рекурсии с применением стека. В дополнение к первому методу рассматривается подход, в котором при реализации автоматной программы используются классы. После этого изложены особенности применения к рекурсивным программам подхода, используемого в предыдущем разделе для преобразования итеративных программ в автоматные. Этот подход позволяет строить автоматно-рекурсивные программы.

Второй метод также обеспечивает раскрытие рекурсии и состоит в построении автомата, осуществляющего обход дерева действий, выполняемых рекурсивной программой.

Третий метод обеспечивает решение задачи непосредственно в терминах дисков и стержней и не использует такие абстракции как деревья и стеки.

Отметим, что применение каждого из предлагаемых методов для рассматриваемой задачи порождает автоматы с двумя или тремя состояниями, управляющие "объектом управления" с 2^N состояниями, которые возникают в процессе перекладывания дисков.

3.5.1. Классическое рекурсивное решение задачи

Известны рекурсивные алгоритмы для решения рассматриваемой задачи [20,21]. Приведем один из таких

алгоритмов, построенный по методу "разделяй и властвуй". Задача о перекладывании N дисков с i-ого на j-тый стержень может быть декомпозирована следующим образом.

1. Переложить N-1 диск со стержня с номером i на стержень с номером 6-i-j.

2. Переложить диск со стержня с номером i на стержень с номером j.

3. Переложить N-1 диск со стержня с номером 6-i-j на стержень с номером j.

Таким образом, задача сводится к решению двух аналогичных задач размерности N-1 и одной задачи размерности один. При этом, если для решения одной задачи размерности N-1 требуется F_{N-1} перекладываний, то их общее число определяется соотношением:

$$F_N = 2F_{N-1} + 1.$$

Из этого соотношения следует [22], что

$$F_N = 2^N - 1.$$

Указанный процесс декомпозиции можно изобразить с помощью дерева декомпозиции (рис. 58).

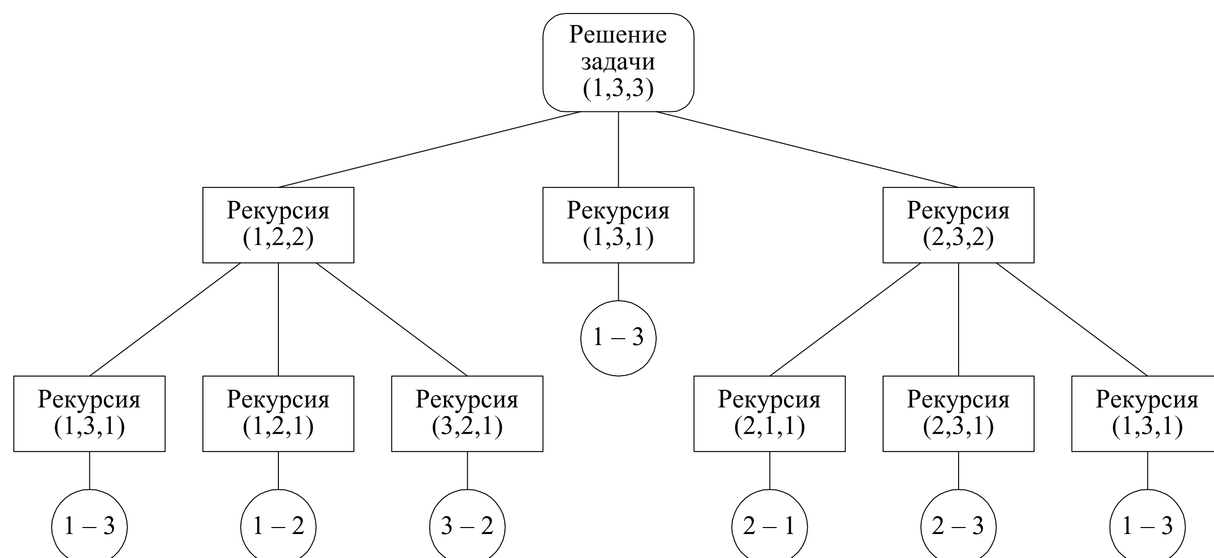


Рис. 58. Дерево декомпозиции для задачи о ханойских башнях при N = 3

В этом дереве вершины, обозначенные прямоугольниками, соответствуют подзадачам, решаемым при каждом вызове рекурсивной функции. Эти вершины помечаются номером стержня, с которого осуществляется перекладывание, номером стержня, на который осуществляется перекладывание, и числом перекладываемых дисков. При этом для вершины с пометкой (i,j,k) левая вершина поддерева будет помечена (i,6-i-j,k-1), средняя — (i,j,1), а правая — (6-i-j,j,k-1).

Перекладывание дисков выполняются в вершинах, обозначенных кружками. Для каждой из этих вершин сохраняются первые две позиции пометки соответствующего прямоугольника — (i,j). Количество таких вершин равно F_N.

Приведем программу (листинг 20), реализующую рассматриваемый рекурсивный алгоритм. Ее поведение эквивалентно обходу дерева декомпозиции слева направо, причем в вершинах обозначенных кружками производятся перекладывания.

ЛИСТИНГ 20. Рекурсивная программа решения задачи о ханойских башнях

```
#include <stdio.h>

void hanoy( int i, int j, int k, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "hanoy(%d,%d,%d)\n", i, j, k ) ;

  if( k == 1 )
    move( i, j, d ) ;
  else
  {
    hanoy( i, 6-i-j, k-1, d+1 ) ;
    hanoy( i, j, 1, d+1 ) ;
    hanoy( 6-i-j, j, k-1, d+1 ) ;
  }
}

void move( int i, int j, int d )
{
  int m ;
  for( m = 0 ; m < d ; m++ ) printf( "   " ) ;
  printf( "move(%d,%d)\n", i, j ) ;
}

void main()
{
```

```
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input, 0 ) ;
}
```

В эту программу введено протоколирование рекурсивных вызовов и четвертый параметр рекурсивной функции, используемый для определения глубины рекурсии. Ниже приводится протокол работы программы, эквивалентный дереву декомпозиции (рис. 58).

ЛИСТИНГ 21. Протокол работы рекурсивной программы

```
Ханой с 3 дисками:
hanoy(1,3,3)
   hanoy(1,2,2)
      hanoy(1,3,1)
      move(1,3)
      hanoy(1,2,1)
      move(1,2)
      hanoy(3,2,1)
      move(3,2)
   hanoy(1,3,1)
   move(1,3)
   hanoy(2,3,2)
      hanoy(2,1,1)
      move(2,1)
      hanoy(2,3,1)
      move(2,3)
      hanoy(1,3,1)
      move(1,3)
```

3.5.2. Моделирование рекурсии автоматной программой

Построим схему программы, приведенной в листинге 20 (рис. 59). Расставим пометки, соответствующие состояниям автомата Мили. При этом точка с номером 2 будет совпадать с точками, следующими за операторными вершинами R3 и z0.

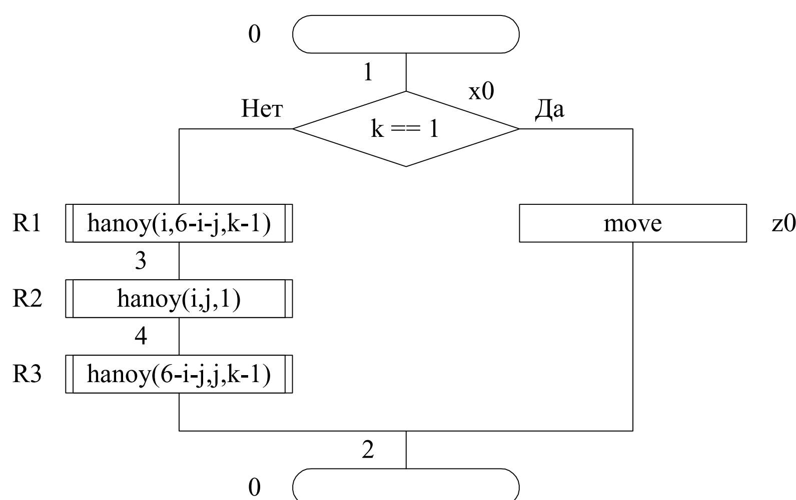


Рис. 59. Схема программы решения задачи о ханойских башнях

По этой схеме построим граф переходов автомата Мили (рис. 60).

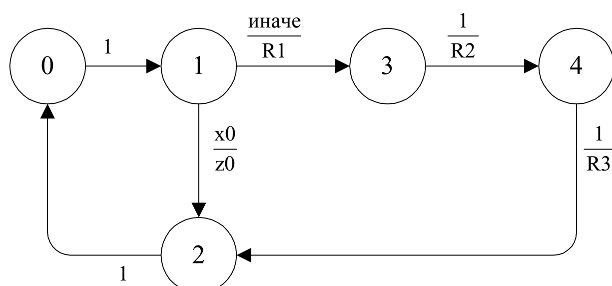


Рис. 60. Граф переходов до преобразования

Преобразуем этот граф переходов для моделирования работы рекурсивной программы. Дуги, содержащие рекурсивные вызовы направляются в вершину с номером 1. При этом сами рекурсивные вызовы заменяются операцией push и действием, выполняемым над параметрами функции. На рис. 61 такие действия обозначены символом z с номером, соответствующим номеру рекурсивного вызова: z1 (j=6-i-j ; k--) для R1, z2 (k=1) для R2 и z3 (i=6-i-j ; k--) для R3. Остальные преобразования выполняются по аналогии с предыдущими примерами.

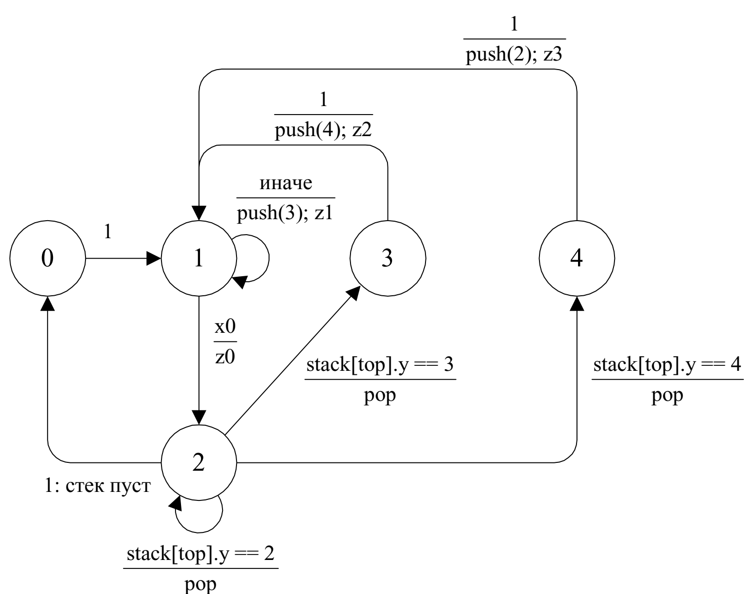


Рис. 61. Преобразованный граф переходов

Автоматная программа, построенная по графу переходов автомата Мили с пятью состояниями (рис. 61), приведена в листинге 22. В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2—0) могут быть закомментированы и заменены одним оператором pop(&y, &i, &j, &k). Такое упрощение программы, однако, приводит к тому, что по указанному оператору невозможно определить, в какое состояние будет выполнен переход.

ЛИСТИНГ 22. Автоматная программа решения задачи о ханойских башнях

```
#include <stdio.h>

typedef struct
{
  int y, i, j, k ;
} stack_t ;

stack_t  stack[200] ;
int      top = -1 ;

push( int y, int i, int j, int k )
{
  top++ ;
  stack[top].y = y ;
  stack[top].i = i ;
  stack[top].j = j ;
  stack[top].k = k ;
  printf( "push {%d,%d,%d,%d}: ", y, i, j, k ) ; show_stack() ;
}

pop( int *y, int *i, int *j, int *k )
{
  printf( "pop  {%d,%d,%d,%d}: ", stack[top].y, stack[top].i,
                                  stack[top].j, stack[top].k ) ;
  if( y ) *y = stack[top].y ;
  *i = stack[top].i ;
  *j = stack[top].j ;
  *k = stack[top].k ;
  top-- ;
  show_stack() ;
}

int stack_empty() { return top < 0 ; }

void show_stack()
{
  int i ;
  for( i = top ; i >= 0 ; i-- )
    printf( "{%d,%d,%d,%d} ", stack[i].y, stack[i].i, stack[i].j, stack[i].k ) ;
  printf( "\n" ) ;
}

void hanoy( int i, int j, int k )
{
  int y = 0, y_old ;

  do
```

```
  {
    y_old = y ;

    switch( y )
    {
      case 0:
                                        y = 1 ;
      break ;

      case 1:
        if( k == 1 ) { move( i, j ) ;  y = 2 ; }
        else
        {
          push( 3, i, j, k ) ;
          j = 6 - i - j ; k-- ;
        }
      break ;

      case 2:
        if( stack_empty() )             y = 0 ;
        else
          //pop( &y, &i, &j, &k ) ;

        if( stack[top].y == 4 )
        {
          pop( NULL, &i, &j, &k ) ;    y = 4 ;
        }
        else
        if( stack[top].y == 3 )
        {
          pop( NULL, &i, &j, &k ) ;    y = 3 ;
        }
        else
        if( stack[top].y == 2 )
        { pop( NULL, &i, &j, &k ) ; }
      break ;

      case 3:
        push( 4, i, j, k ) ;
        k = 1 ;                         y = 1 ;
      break ;

      case 4:
        push( 2, i, j, k ) ;
        i = 6 - i - j ; k-- ;           y = 1 ;
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n", y ) ;
  }
  while( y != 0 ) ;
}

void move( i, j )
{
  printf( "%d -> %d\n", i, j ) ;
}

void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

Упростим граф переходов, приведенный на рис. 61, за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 62).

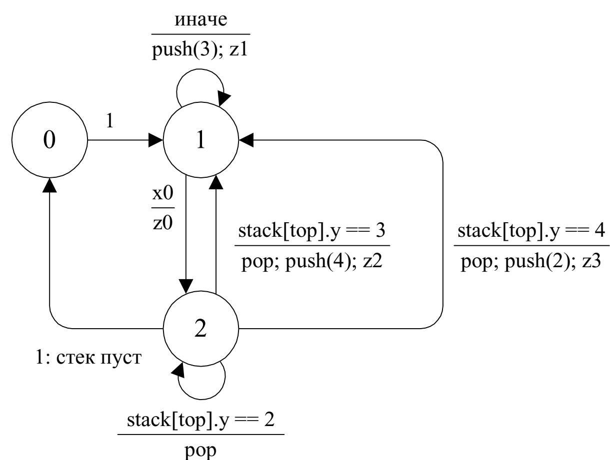


Рис. 62. Упрощенный граф переходов

При построении программы по графу переходов автомата Мили с тремя состояниями (рис. 62), функция hanoy() реализуется как показано в листинге 23. Остальная часть программы остается неизменной по сравнению с листингом 22. Отметим, что при такой реализации не удается выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией pop.

ЛИСТИНГ 23. Автоматная функция решения задачи о ханойских башнях

```
void hanoy( int i, int j, int k )
{
  int y = 0, y_old ;

  do
  {
    y_old = y ;

    switch( y )
    {

      case 0:
                                         y = 1 ;
      break ;

      case 1:
        if( k == 1 ) { move( i, j ) ;  y = 2 ; }
        else
        {
          push( 3, i, j, k ) ;
          j = 6 - i - j ; k-- ;
        }
      break ;

      case 2:
        if( stack_empty() )              y = 0 ;
        else
        if( stack[top].y == 4 )
        {
          pop( NULL, &i, &j, &k ) ;
          push( 2, i, j, k ) ;
```

```
          i = 6 - i - j ; k-- ;        y = 1 ;
        }
        else
        if( stack[top].y == 3 )
        {
          pop( NULL, &i, &j, &k ) ;
          push( 4, i, j, k ) ;
          k = 1 ;                      y = 1 ;
        }
        else
        if( stack[top].y == 2 )
        { pop( NULL, &i, &j, &k ) ; }
      break ;
    }

    if( y_old != y ) printf( "Переход в состояние %d\n", y ) ;
  }
  while( y != 0 ) ;
}
```

Протокол, отражающий работу со стеком для программы, построенной по упрощенному графу переходов, приведен в листинге 24.

ЛИСТИНГ 24. Протокол, отражающий работу со стеком при решении задачи о ханойских башнях

```
Ханой с 3 дисками:
Переход в состояние 1
push {3,1,3,3}: {3,1,3,3}
push {3,1,2,2}: {3,1,2,2} {3,1,3,3}
1 -> 3
Переход в состояние 2
pop  {3,1,2,2}: {3,1,3,3}
push {4,1,2,2}: {4,1,2,2} {3,1,3,3}
Переход в состояние 1
1 -> 2
Переход в состояние 2
pop  {4,1,2,2}: {3,1,3,3}
push {2,1,2,2}: {2,1,2,2} {3,1,3,3}
Переход в состояние 1
3 -> 2
Переход в состояние 2
pop  {2,1,2,2}: {3,1,3,3}
pop  {3,1,3,3}:
push {4,1,3,3}: {4,1,3,3}
Переход в состояние 1
1 -> 3
Переход в состояние 2
pop  {4,1,3,3}:
push {2,1,3,3}: {2,1,3,3}
Переход в состояние 1
push {3,2,3,2}: {3,2,3,2} {2,1,3,3}
2 -> 1
Переход в состояние 2
pop  {3,2,3,2}: {2,1,3,3}
push {4,2,3,2}: {4,2,3,2} {2,1,3,3}
Переход в состояние 1
2 -> 3
Переход в состояние 2
pop  {4,2,3,2}: {2,1,3,3}
push {2,2,3,2}: {2,2,3,2} {2,1,3,3}
Переход в состояние 1
1 -> 3
Переход в состояние 2
pop  {2,2,3,2}: {2,1,3,3}
```

```
pop {2,1,3,3}:
Переход в состояние 0
```

3.5.3. Реализация рекурсивной программы автоматно-рекурсивной программой

Изложенный в предыдущем разделе метод построения автоматной программы по итеративной может быть применен и к рекурсивным программам. При этом в функциях, реализующих получаемые графы переходов, сохраняются рекурсивные вызовы. Получаемые в результате программы относятся к классу автоматно-рекурсивных.

Особенность применения указанного подхода для рекурсивных программ состоит в том, что для корректной работы автоматно-рекурсивных программ переменные состояния автоматов должны быть объявлены как локальные внутри соответствующих рекурсивных функций, в то время как для итеративных программ они могли быть и глобальными.

Автоматно-рекурсивная программа может быть построена непосредственно по графу переходов на рис. 60. Функция, реализующая этот граф переходов, приведена в листинге 25.

ЛИСТИНГ 25. Функция hanoy() для автоматно-рекурсивной программы

```
void hanoy( int i, int j, int k )
{
  int y = 0 ;

  do
    switch( y )
    {
      case 0:
                                        y = 1 ;
      break ;

      case 1:
        if( k == 1 ) { move( i, j ) ;  y = 2 ; }
        else
        { hanoy( i, 6-i-j, k-1 ) ;     y = 3 ; }
      break ;

      case 2:
                                        y = 0 ;
      break ;

      case 3:
        hanoy( i, j, 1 ) ;             y = 4 ;
```

```
      break ;

      case 4:
        hanoy( 6-i-j, j, k-1 ) ;       y = 2 ;
      break ;
    }
  while( y != 0 ) ;
}
```

Отметим, что в общем случае для применения этого подхода нет необходимости в выделении рекурсивных вызовов в отдельные операторы и введении дополнительных вершин в граф переходов, как это было сделано на рис. 59, 60.

3.5.4. Реализация автоматной программы с использованием классов

Рассмотренные в предыдущих примерах программы могут быть реализованы с применением классов. При этом область видимости используемых переменных ограничивается рамками класса, что позволяет эффективно реализовать входные и выходные воздействия в виде методов класса.

Покажем это на примере автомата (рис. 61), построенного в соответствии с рассмотренным подходом к раскрытию рекурсии. Схема связей для этого автомата приведена на рис. 63.

Рис. 63. Схема связей автомата

Граф переходов этого автомата приведен на рис. 64. Он отличается от графа, приведенного на рис. 61, использованием символа x1 на дуге 2—0, а также тем, что остальные дуги, исходящие из вершины с номером 2, изображены пунктиром, так как они будут реализованы одной операцией pop.

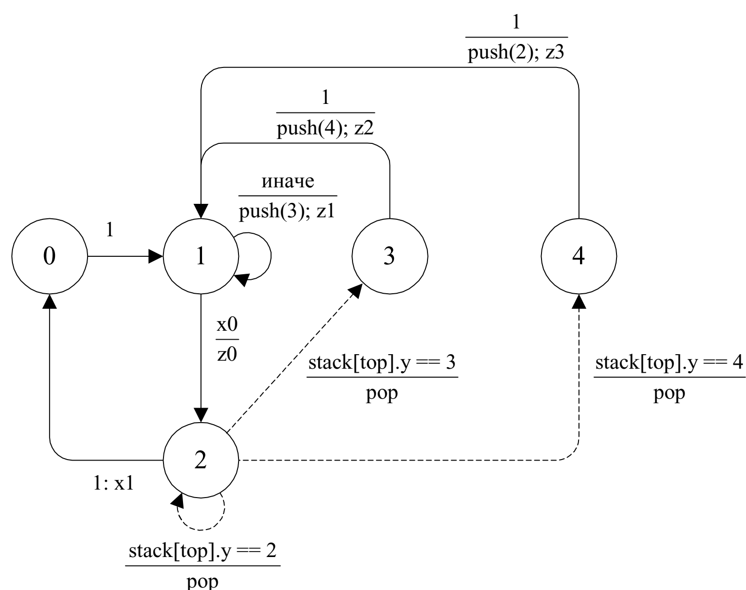


Рис. 64. Граф переходов автомата

Структурная схема класса, в котором автомат реализуется в методе hanoy(), приведена на рис. 65. К методам этого класса относятся также методы, реализующие входные переменные, выходные воздействия и операции со стеком.

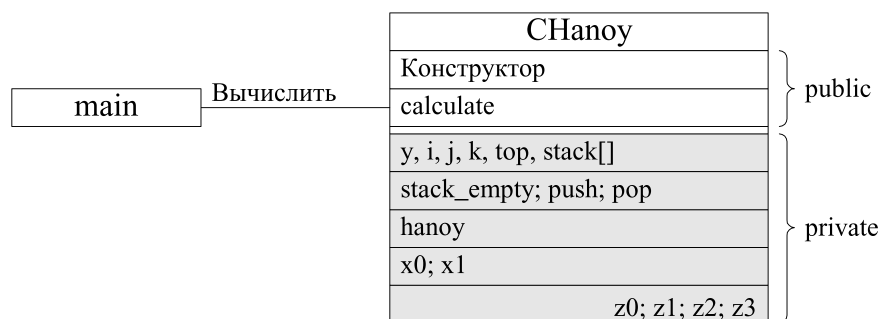


Рис. 65. Схема класса

Текст программы, в которой функция, реализующая рассмотренный граф переходов, является методом класса, приведен в листинге 26.

ЛИСТИНГ 26. Программа решения задачи о ханойских башнях с применением классов

```
#include <stdio.h>

class CHanoy
{
public:
  CHanoy() : y(0), top(-1) {} ;  // Конструктор.

  void calculate( int from, int to, int N ) ;
```

```
private:
  int y ;
  int i, j, k ;

  // Элемент стека.
  struct stackEntry
  {
    int y, i, j, k ;
    stackEntry() {} ;
    stackEntry( int y, int i, int j, int k )
      : y(y), i(i), j(j), k(k) {} ;
  } ;

  // Атрибуты и методы для работы со стеком.
  int         top ;
  stackEntry  stack[100] ;
  int         stack_empty() { return top < 0 ; }
  void        push( int y ) { stack[++top] = stackEntry ( y, i, j, k ) ; }
  void        pop() ;

  void hanoy() ;

  int  x0() { return k == 1 ; }
  int  x1() { return stack_empty() ; }

  void z0() { printf( "%d -> %d\n", i, j ) ; }
  void z1() { j = 6-i-j ; k-- ; }
  void z2() { k = 1 ; }
  void z3() { i = 6-i-j ; k-- ; }
};

void CHanoy::calculate( int from, int to, int N )
{
  i = from ; j = to ; k = N ;
  printf( "\nХаной с %d дисками:\n", k ) ;
  hanoy() ;
}

void CHanoy::hanoy()
{
  do
    switch ( y )
    {
      case 0:
                               y = 1 ;
      break ;

      case 1:
        if( x0() ) { z0() ;  y = 2 ; }
        else
        { push(3) ; z1() ; }
      break;

      case 2:
        if( x1() )             y = 0 ;
        else
        { pop() ; }
      break;

      case 3:
        push(4) ; z2() ;       y = 1 ;
      break;

      case 4:
        push(2) ; z3() ;       y = 1 ;
      break;
    }
  while( y != 0 ) ;
}
```

```
void CHanoy::pop()
{
  y = stack[top].y ;
  i = stack[top].i ;
  j = stack[top].j ;
  k = stack[top].k ;
  top-- ;
}

void main()
{
  CHanoy hanoy ;
  hanoy.calculate( 1, 3, 3 ) ;
}
```

Из рассмотрения приведенной программы следует, что "оборачивание" автомата классом обеспечивает сокрытие переменных внутри этого класса. Область видимости этих переменных ограничена рамками класса, и поэтому не требуется передавать их значения в виде параметров функций, реализующих входные и выходные воздействия. В качестве дополнительного достоинства приведенной программы следует отметить компактность реализации операции push по сравнению с другими приведенными выше программами.

3.5.5. Обход дерева действий

Дерево декомпозиции (рис. 58), вершинами которого являются рекурсивные вызовы, может быть преобразовано в дерево действий, выполняемых рассмотренной рекурсивной программой, путем исключения всех вершин кроме тех, в которых выполняется перекладывание (рис. 66).

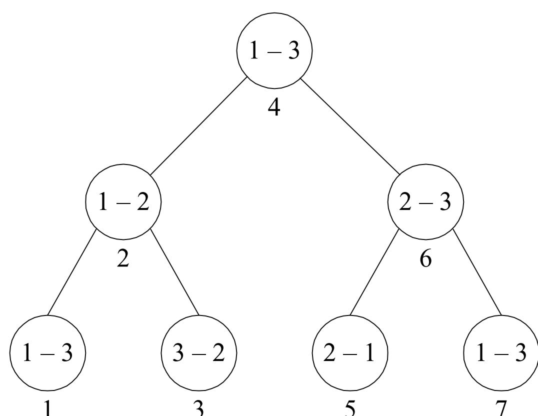


Рис. 66. Дерево действий при N = 3

Это дерево обладает следующим свойством: для вершины с пометкой (i,j) вершина левого поддерева имеет пометку (i,6-i-j), а вершина правого поддерева — (6-i-j,j).

Обход полученного дерева может быть выполнен как рекурсивно, так и итеративно. Построение итеративного алгоритма обхода может рассматриваться как способ раскрытия рекурсии. При этом необходимая последовательность перекладываний получается в результате обхода этого дерева слева направо. Такому обходу соответствуют числа, указанные рядом с каждой из вершин.

Построим итеративный алгоритм обхода этого дерева. Не храня дерево в памяти, будем осуществлять двоичный поиск каждой вершины, начиная с корневой, так как для нее известны номера стержней и способ определения их номеров для корневых вершин левого и правого поддеревьев. Например, для вершины с номером 5, на первом шаге алгоритма осуществляется переход от вершины 4 к ее правому поддереву — вершине 6, так как пять больше четырех. На втором шаге осуществляется переход от вершины 6 к ее левому поддереву — искомой вершине 5, так как пять меньше шести. Таким образом, несмотря на то, что обход дерева осуществляется слева направо, алгоритм поиска вершин работает сверху вниз.

Программа, реализующая этот алгоритм, приведена в листинге 27.

ЛИСТИНГ 27. Итеративная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k )
{
  int max_nodes = (1 << k) - 1 ; // Всего вершин.
  int root = 1 << (k-1) ;        // Номер корневой вершины.
  int node ;                     // Номер искомой вершины в дереве.

  // Определить номера стержней для каждой вершины в дереве.
  for( node = 1 ; node <= max_nodes ; node++ )
  {
    int a = i, b = j ;
    // Начальная позиция поиска соответствует корневой вершине.
```

```
    int current = root ;
    // Изменение номера вершины при переходе к следующему поддереву.
    int ind = root / 2 ;

    // Двоичный поиск нужной вершины.
    while( node != current )
    {
      if( node < current )
      {
        // Искомая вершина в левом поддереве.
        b = 6-a-b ;
        current -= ind ; // Переход к левому поддереву.
      }
      else
      {
        // Искомая вершина в правом поддереве.
        a = 6-a-b ;
        current += ind ; // Переход к правому поддереву.
      }
      // Разница в номерах вершин при переходе к
      // следующему поддереву уменьшается в два раза.
      ind /= 2 ;
    }

    // Номера стержней для рассматриваемой вершины определены.
    printf( "Вершина %d. %d -> %d\n", node, a, b ) ;
  }
}

void main()
{
  int input = 3 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

Построим схему этой программы (рис. 67).

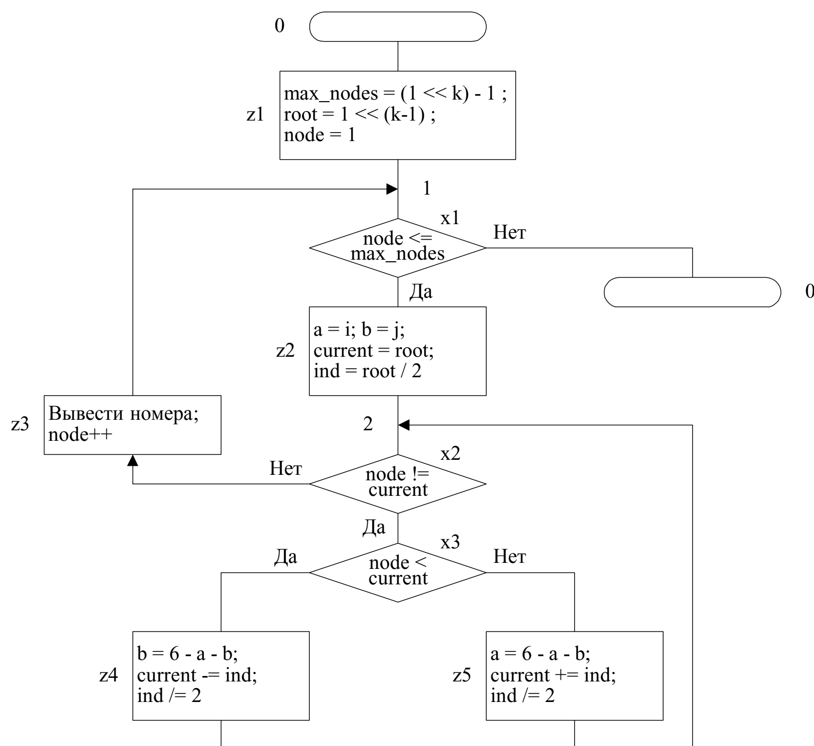


Рис. 67. Схема программы обхода дерева действий

В соответствии с методикой, изложенной в предыдущем разделе, построим по этой схеме граф переходов автомата Мили. При этом состояниям автомата Мили на схеме программы соответствуют точки, следующие за операторными вершинами. Нулевому состоянию соответствуют вершины схемы, обозначающие начало и конец программы. Построенный граф переходов приведен на рис. 68.

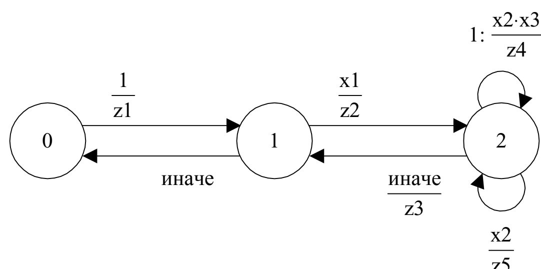


Рис. 68. Граф переходов автомата, реализующего обход дерева действий

Программа, реализующая этот автомат, приведена ниже.

ЛИСТИНГ 28. Автоматная программа обхода дерева действий

```
#include <stdio.h>

void hanoy( int i, int j, int k )
{
  int y = 0 ;
  int max_nodes, root, node, a, b, current, ind ;

  do
    switch( y )
    {
      case 0:
        max_nodes = (1 << k) - 1 ;
        root = 1 << (k-1) ;
        node = 1 ;                  y = 1 ;
      break ;

      case 1:
        if( node <= max_nodes )
        {
          a = i ; b = j ;
          current = root ;
          ind = root / 2 ;          y = 2 ;
        }
        else                        y = 0 ;
      break ;

      case 2:
        if( node != current && node < current )
        {
          b = 6-a-b ;
          current -= ind ;
          ind /= 2 ;
        }
        else
```

```
        if( node != current )
        {
          a = 6-a-b ;
          current += ind ;
          ind /= 2 ;
        }
        else
        {
          printf( "Вершина %2d. %d -> %d\n", node, a, b ) ;
          node++ ;                    y = 1 ;
        }
      break ;
    }
  while( y != 0 ) ;
}

void main()
{
  int input = 4 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}
```

Авторы благодарны студенту СПбГИТМО (ТУ) Крылову Р.А., который в своей курсовой работе начал рассмотрение метода, излагаемого в настоящем разделе.

Отметим, что при использовании этого метода номер перекладываемого диска явно не указывается. Номера дисков могут быть определены с помощью подхода, изложенного в работе [23], который состоит в следующем: строится таблица, строки которой содержат двоичное представление номера шага (табл. 2). При этом номер разряда двоичного числа, в котором размещена "младшая единица" (при условии, что счет разрядов начинается с единицы), является номером перекладываемого на данном шаге диска. Как отмечено в работе [24], последовательность номеров перекладываемых дисков является палиндромом.

ТАБЛИЦА 2. *Определение номера перекладываемого диска* (*для* N = 3)

Номер шага	Номер разряда			Номер диска
	3	2	1	
1	0	0	1	1
2	0	1	0	2
3	0	1	1	1
4	1	0	0	3
5	1	0	1	1

6	1 1 0	2
7	1 1 1	1

Отметим, что аналитически номер перекладываемого диска может быть определен как единица плюс количество делений номера шага на два без остатка. При этом для нечетных номеров шагов количество делений на два без остатка равно нулю, и поэтому номер диска равен единице. Таким образом, на каждом нечетном шаге всегда перекладывается диск наименьшего диаметра.

Обобщая изложенное выше, построим дерево решения задачи (рис. 69), которое содержит исчерпывающую информацию для перекладывания дисков. В этом дереве для каждой вершины снизу указан номер шага, а внутри — номера диска и стержней, участвующих в перекладывании.

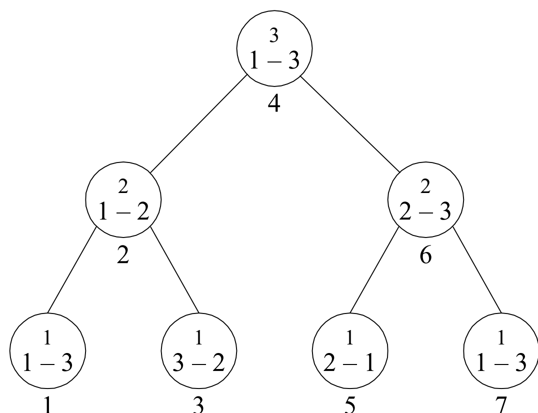


Рис. 69. Дерево решения задачи при N = 3

Из рассмотрения рис. 69 следует также, что номера перекладываемых дисков во всех вершинах одного уровня равны и совпадают с номером этого уровня.

3.5.6. Решение задачи в терминах "объекта управления"

Если алгоритм решения рассматриваемой задачи задавать непосредственно в терминах объекта управления (дисков и стержней), как это предлагается П. Бьюнеманом и Л. Леви

[25], то его удается описать в виде графа переходов автомата с двумя состояниями (рис. 70).

Этот автомат по очереди выполняет всего два действия: перекладывает наименьший диск слева направо циклически (z1) и перекладывает единственно возможный диск, кроме наименьшего (z2). После выполнения действия z1 автомат проверяет условие завершения алгоритма (x1).

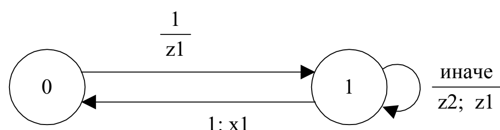


Рис. 70. Граф переходов при непосредственном перекладывании дисков

Текст программы, реализующей описываемый автомат, приведен в листинге 29. В этой программе функции z1, z2 и x1 могут быть реализованы по-разному. Например, определение номеров перекладываемого диска и участвующих в перекладывании стержней может выполняться либо с помощью перебора (как это имеет место ниже), либо аналитически, например, как это предложено в работе [26].

Особенность рассматриваемой программы состоит в том, что несмотря на простой управляющий автомат, она содержит достаточно много строк, так как выполняемые в ней действия z1 и z2 являются сложными. При этом программа непосредственно управляет дисками, и поэтому необходимо запоминать их расположение и реализовать соответствующие вспомогательные функции.

Кроме того, в используемом алгоритме направление перекладывания первого диска (функция z1) зависит от четности числа дисков и номера стержня, на который их требуется переложить. При перекладывании дисков с первого стержня на третий, первый диск следует перекладывать в порядке номеров стержней 1—2—3 при четном количестве дисков, и в порядке 1—3—2 при их нечетном количестве.

ЛИСТИНГ 29. Автоматная программа, реализующая непосредственное перекладывание дисков

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int   y = 0 ;
int   N ;             // Количество дисков.
int   dest ;          // На какой стержень перекладываем.
int   step = 0 ;      // Номер текущего шага.
int   max_steps = 0 ; // Необходимое количество шагов.
int   first_on = 1 ;  // На каком стержне находится первый диск.

// Состояние объекта управления - "содержимое" стержней.
char  s[4][100] = { "", "1", "", "" } ;

void main()
{
  int input = 14 ;
  printf( "\nХаной с %d дисками:\n", input ) ;
  hanoy( 1, 3, input ) ;
}

void hanoy( int from, int to, int disk_num )
{
  int i ;

  N = disk_num ;
  max_steps = (1 << N) - 1 ;
  dest = to ;

  // Заполнить первый стержень дисками.
  for( i = 2 ; i <= N ; i++ )
  {
    sprintf( s[0], "-%d", i ) ;
    strcat( s[1], s[0] ) ;
  }

  do
    switch( y )
    {
      case 0:
        z1() ;              y = 1 ;
      break ;

      case 1:
        if( x1() )          y = 0 ;
        else
        { z2() ; z1() ; }
      break ;
    }
  while( y != 0 ) ;

  // Вывести результат.
  printf( "\nРезультат:\n" ) ;
  for( i = 1 ; i <= 3 ; i++ )
    printf( "%d: %s\n", i, s[i] ) ;
}

// Проверить, завершено ли перекладывание.
int x1() { return step >= max_steps ; }

// Переложить первый диск по часовой стрелке.
void z1()
{
  int from = first_on ;
  int i ;

  // Определить номер следующего стержня.
```

```
  if(    (dest == 2 && N%2 != 0)
      || (dest == 3 && N%2 == 0))
    first_on = (from + 1)%3 ; // Порядок стержней 1-2-3.
  else
    first_on = (from + 2)%3 ; // Порядок стержней 1-3-2.
  if( first_on == 0 ) first_on = 3 ;
  move( 1, from, first_on ) ;
}

// Переложить единственный возможный диск, кроме наименьшего.
void z2()
{
  int  i, j ;
  int  disk_from, disk_to ;

  // Определить перекладываемый диск.
  for( i = 1 ; i <= 3 ; i++ )
  {
    disk_from = disk_on(i) ;
    if( disk_from > 1 )
      // Определить на какой стержень перекладывать.
      for( j = 1 ; j <= 3 ; j++ )
      {
        disk_to = disk_on(j) ;
        if( disk_to == 0 || disk_from < disk_to )
        {
          move( disk_from, i, j ) ;
          return ;
        }
      }
  }
}

// Вернуть номер диска на указанном стержне.
int disk_on( int s_num ) { return atoi( s[s_num] ) ; }

// Переложить заданный диск.
int move( int disk, int from, int to )
{
  char  *str_pos = strchr( s[from], '-' ) ;

  if( str_pos == NULL )
    s[from][0] = 0 ;
  else
    strcpy( s[from], str_pos+1 ) ;

  if( s[to][0] == 0 )
    sprintf( s[to], "%d", disk ) ;
  else
  {
    strcpy( s[0], s[to] ) ;
    sprintf( s[to], "%d-%s", disk, s[0] ) ;
  }

  step++ ;
  printf( "Шаг %d. Диск %d: %d -> %d\n", step, disk, from, to ) ;

  return 0 ;
}
```

4. Заключение

Предложенная на предыдущих этапах НИР технология автоматного программирования расширена применительно еще к одному классу задач — вычислительным алгоритмам.

Предложена вычислительная модель, которая является более удобной в практическом использовании по сравнению с машиной Тьюринга.

Сопоставление рассмотренных в настоящей работе традиционных схем алгоритмов и графов переходов показывает, что последние более компактны и понятней специфицируют алгоритм. Это связано с тем, что построение графа переходов сводится к исследованию всех путей между соседними пометками в традиционной схеме алгоритма и компактному их представлению в виде помеченных дуг и петель графа.

Преимущество автоматного подхода по сравнению с традиционным достигается за счет добавления к понятиям "входное и выходное воздействия" понятия "**состояние**", а по сравнению с методом Ашкрофта-Манны — за счет сокращения числа состояний.

Предложенное для вычислительных алгоритмов явное выделение состояний позволяет рассматривать такие алгоритмы не в терминах условных переходов (и, соответственно, флагов), а в терминах состояний, что более наглядно отражает их суть. Кроме того, задание вычислительных алгоритмов в виде графов переходов позволяет формализовать задачу их визуализации, что существенно упрощает их отладку и особенно важно с точки зрения обучения [17].

В третьей части работы предложен метод преобразования программ с явной рекурсией в автоматные программы.

1. Программа с явной рекурсией преобразуется в итеративную программу, построенную с применением автомата Мили. При этом работа рекурсивной программы моделируется итеративной автоматной программой, в которой явно используется стек.

2. Явное выделение стека по сравнению со "скрытым" его применением в рекурсии, позволяет программно задавать его размер, следить за его содержимым и добавлять отладочный код в функции, реализующие операции над стеком.

3. Предлагаемый метод иллюстрируется примерами преобразований классических рекурсивных программ, которые приведены в порядке их усложнения (факториал, числа Фибоначчи, задача о ранце, ханойские башни).

4. Для задачи о ханойских башнях кроме предлагаемого метода преобразования рассматриваются и другие подходы к преобразованию рекурсивных алгоритмов в автоматные: обход дерева действий, выполняемых рекурсивной программой, и решение задачи в терминах "объекта управления".

5. На примере задачи о ханойских башнях рассматривается подход, в котором при реализации автоматной программы применяются классы. Это позволяет ограничить область видимости используемых переменных пределами класса и эффективно реализовать сам автомат и его входные и выходные воздействия в виде методов.

6. Показано, что метод, предложенный для построения автоматных программ по итеративным, применим также и к рекурсивным программам.

7. В работе [27] отмечено, что итеративные алгоритмы по вычислительной мощности эквивалентны рекурсивным. Однако, в известной авторам литературе, формальный метод преобразования рекурсивных алгоритмов в итеративные отсутствует. Как отмечалось выше, в работах [4,19]

приведены примеры такого преобразования (для концевых рекурсий), выполненного эвристически. Материалы, изложенные в настоящей работе, устраняют указанный пробел.

Можно считать, что предложенные в работе методы преобразования итеративных и рекурсивных программ в автоматные обеспечивают преобразование процедурных знаний в декларативные [18].

5. Публикации по результатам этапа

По результатам выполненных в ходе этапа работ опубликованы следующие статьи:

1 Шалыто А.А., Туккель Н.И. Реализация вычислительных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. 2001. №6. С.35—53.

2 Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному // Мир ПК. 2002. №2. С.144—149.

3 Шалыто А.А., Туккель Н.И., Шамгунов Н.Н. Ханойские башни и автоматы // Программист. — 2002. — № 8. — С. 82—90.

4 Шалыто А.А., Туккель Н.И. Преобразование итеративных алгоритмов в автоматные // Программирование. — 2002. — №5. — С. 12—26.

5 Шалыто А.А., Туккель Н.И., Шамгунов Н.Н. Реализация рекурсивных алгоритмов на основе автоматного подхода // Телекоммуникации и информатизация образования. — 2002. — №5. - С. 72-99.

6 Шалыто А.А., Туккель Н.И., Шамгунов Н.Н. Задача о ходе коня // Мир ПК. — 2003. — №1. С. 152-155.

По результатам выполненных в ходе этапа работ в материалах конференций опубликованы:

1 Казаков М.А., Шалыто А.А., Туккель Н.И. Использование автоматного подхода для реализации вычислительных алгоритмов /Труды международной научно-методической конференции "Телематика'2001". СПб.: СПбГИТМО (ТУ), 2001. С.174—176.
2 Шалыто А.А., Туккель Н.И. Применение SWITCH-технологии для решения классических задач распознавания цепочек символов /Труды международной научно-методической конференции "Телематика'2001". СПб.: СПбГИТМО (ТУ), 2001. С.177—179.
3 Шалыто А.А., Туккель Н.И. Автоматное программирование как практическое развитие тьюрингова программирования /Тезисы докладов международной научной конференции "Интеллектуальные и многопроцессорные системы - 2001". Таганрог: ТРТУ, 2001. С.123—126.
4 Шалыто А.А., Туккель Н.И. Реализация рекурсивных алгоритмов автоматными программами /Труды международной научно-методической конференции "Телематика'2002". СПб.: СПбГИТМО (ТУ), 2002. С.181—182.

Список литературы

1. *Кнут Д.* Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2000.

2. *Хопкрофт Д.* Машины Тьюринга //В мире науки. 1984. ¹ 7.

3. *Трахтенброт В.А.* Алгоритмы и вычислительные автоматы. М.: Советское радио. 1974.

4. *Ахо А.*, *Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. М.: Мир, 1978.

5. *Льюис Ф.*, *Розенкранц Д.*, *Стирнз Р.* Теоретические основы проектирования компиляторов. М.: Мир, 1979.

6. *Шалыто А.А.*, *Туккель Н.И.* Программирование с явным выделением состояний // Мир ПК. — 2001. — №8, 9.

7. *Ершов А.П.* Смешанные вычисления //В мире науки. 1984. ¹ 6.

8. *Лавров С.С.* Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001.

9. *Фон Нейман Дж.* Общая и логическая теория автоматов /В кн. Тьюринг А. Может ли машина мыслить? Саратов: Колледж, 1999.

10. *Кузнецов Б.П.* Психология автоматного программирования //BYTE/Россия. 2000. №11.

11. *Казаков М.А.*, *Столяр С.Е.* Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования //Телематика 2000. Тез. докл. Международной научно-метод. конф. СПб.: СПбГИТМО (ТУ), 2000.

12. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.

13. Aschcroft E., Manna Z. The translation of "goto" programm into "while" programm //Proceeding of 1971 IFIP Congress.

14. *Йодан Э.* Структурное проектирование и конструирование программ. М.: Мир, 1979.

15. *Баранов С.И.* Синтез микропрограммных автоматов (граф-схемы и автоматы). Л.: Энергия, 1979.

16. *Лингер Р.*, *Миллс Х.*, *Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982.

17. *Казаков М.А.*, *Шалыто А.А.*, *Туккель Н.И.* Использование автоматного подхода для реализации вычислительных алгоритмов //Труды международной научно-методической конференции "Телематика'2001". СПб.: СПбГИТМО, 2001.

18. *Станкевич Л.А.* Интеллектуальные технологии представления знаний. Интеллектуальные системы. СПб.: СПбГТУ, 2000.

19. *Стивенс Р.* Delphi. Готовые алгоритмы. М.: ДМК, 2001.

20. *Седжвик Р.* Фундаментальные алгоритмы на С++. Киев: ДиаСофт, 2001.

21. *Бобак И.* Алгоритмы: "возврат назад" и "разделяй и властвуй" //Программист. 2002. №3.

22. *Грэхем Р.*, *Кнут Д.*, *Поташник О.* Конкретная математика. М.: Мир, 1998.

23. *Гарднер М.* Математические головоломки и развлечения. М.: Мир, 1971.

24. *Романовский И.В.*, *Столяр С.Е.* Стек и его использование. http://ips.ifmo.ru.

25. *Анисимов А.В.* Информатика. Творчество. Рекурсия. Киев: Наукова думка, 1988.

26. Áûñòðèöêèé Â.Ä. Õàíîéñêèå áàøíè. http://alglib.chat.ru/paper/hanoy.html

27. *Брукшир Дж*. Введение в компьютерные науки. М.: Вильямс, 2001.

28. *Шалыто А.А.*, *Туккель Н.И.*, *Шамгунов Н.Н*. Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2002. №6.