Опубликована в журнале "Телекоммуникации и информатизация образования", 2002, № 5.

А.А. Шалыто,
доктор технических наук, профессор,
Н.И. Туккель,
Н.Н. Шамгунов

Реализация рекурсивных алгоритмов на основе автоматного подхода

В работе [8] предложен метод преобразования произвольных итеративных программ в автоматные программы, что позволяет реализовать произвольный итеративный алгоритм структурированной программой, содержащей один оператор do-while, телом которого является один оператор switch.

В работах [1,6] приведены примеры преобразований рекурсивных программ 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. Такие программы, как и в работе [8], будем называть автоматными. Метод иллюстрируется примерами преобразований классических рекурсивных программ, которые приведены в порядке их усложнения (факториал, числа Фибоначчи, ханойские башни, задача о ранце).

Изложение метода

Идея метода состоит в моделировании работы рекурсивной программы автоматной программой, явно (также, как и в работах [1,6]) 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 работе [8] 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.

Работа выполнена при поддержке Российского фонда фундаментальных исследований по гранту №02-07-90114 "Разработка технологии автоматного программирования".

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 [8].

Факториал

Одной из простейших задач, использующих рекурсию, является вычисление факториала. Построим автоматную программу по рекурсивной программе, в которой рекурсивный вызов выделен отдельным оператором (листинг 1).

ЛИСТИНГ 1. 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 ;
  }
}
```

```
int main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nФакториал(%d) = %d\n", input, f ) ;
  return 0 ;
}
```

Построим схему этой программы (рис. 1), а по ней – граф переходов автомата Мили (рис. 2).

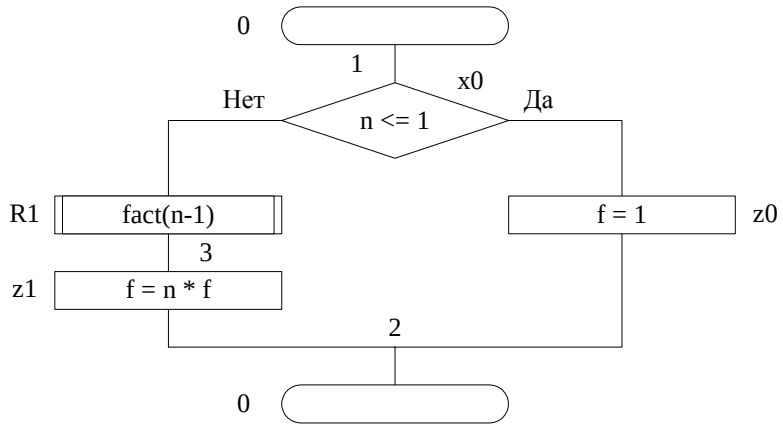

Рис. 1. Схема программы вычисления факториала

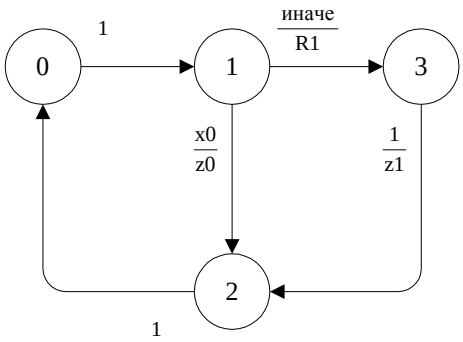

Рис. 2. Граф переходов до преобразования

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--), совершаемое над параметром рекурсивной функции при ее вызове. Условие перехода на дуге 2–0 заменяется на "стек пуст" и добавляется дуга 2–3, при переходе по которой выполняется действие pop (рис. 3).

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. 4).

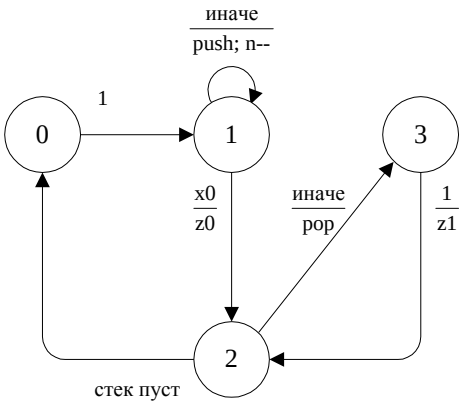

Рис. 3. Преобразованный граф переходов

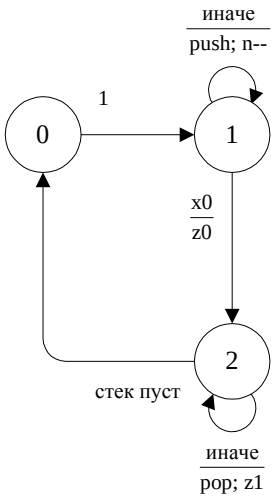

Рис. 4. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 2.

ЛИСТИНГ 2. 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 ) ;
}

int main()
{
  int input = 7 ;
  fact( input ) ;
  printf( "\nФакториал(%d) = %d\n", input, f ) ;
  return 0 ;
}
```

Протокол, отражающий работу со стеком, приведен в листинге 3.

ЛИСТИНГ 3. 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
```

Числа Фибоначчи

Более сложной задачей является вычисление чисел Фибоначчи, программа решения которой содержит две рекурсии, выделенные в виде отдельных операторов (листинг 4).

ЛИСТИНГ 4. 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 ;
  }
}
```

```
int main()
{
  int input = 30 ;
  fibo( input ) ;
  printf( "\nФибоначи(%d) = %d\n", input, res ) ;
  return 0 ;
}
```

Отметим, что приведенная программа весьма неэффективна ввиду повторного вычисления одних и тех же значений. Эта программа может быть усовершенствована с помощью динамического программирования [5]. Построим автоматную программу по приведенной выше рекурсивной программе.

Построим схему этой программы (рис. 5), а по ней – граф переходов автомата Мили (рис. 6).

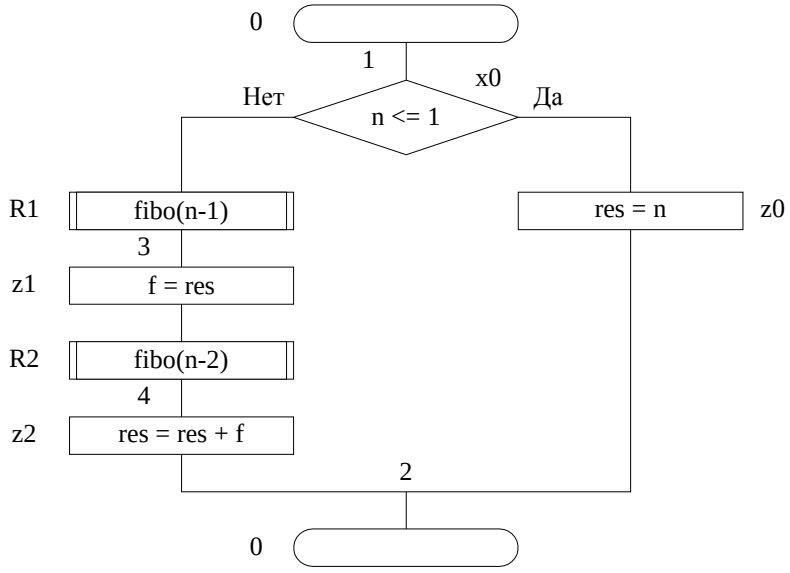

Рис. 5. Схема программы вычисления чисел Фибоначчи

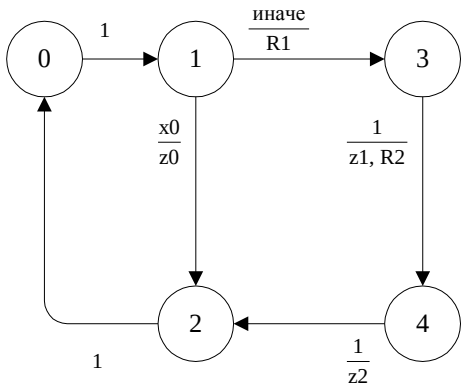

Рис. 6. Граф переходов до преобразования

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. 7).

Дуги 2–3 и 2–4 соответствуют возврату из рекурсивной функции. Так как исходная программа содержит две рекурсии, необходимо запоминать в стеке не только локальные переменные n и f, но и значение переменной состояния автомата y.

Упростим этот граф за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 8).

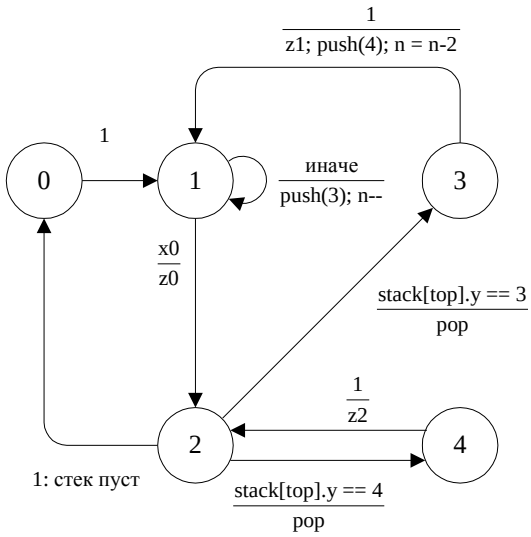

Рис. 7. Преобразованный граф переходов

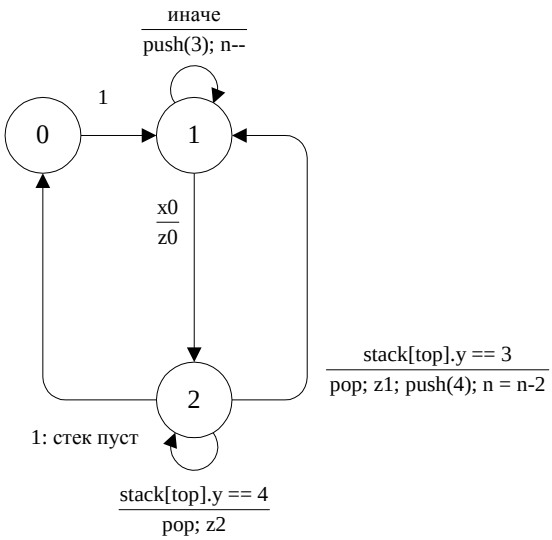

Рис. 8. Упрощенный граф переходов

Автоматная программа, построенная по упрощенному графу переходов, приведена в листинге 5.

ЛИСТИНГ 5. 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 ) ;
}

int main()
{
  int input = 4 ;
  fibo( input ) ;
  printf( "\nФибоначи(%d) = %d\n", input, res ) ;
  return 0 ;
}
```

Протокол, отражающий работу со стеком, приведен в листинге 6.

ЛИСТИНГ 6. 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

Фибоначи(4) = 3
```

Задача о ханойских башнях

Одной из наиболее известных рекурсивных задач является задача о ханойских башнях [5], 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.

Эта задача является более сложной, чем приведенные выше, так как рассматриваемый вариант ее решения (листинг 7) содержит три рекурсии. В приведенной программе содержатся вызовы функций протоколирования ее работы с отображением глубины рекурсии.

ЛИСТИНГ 7. 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 ) ;
}

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

Построим схему этой программы (рис. 9), а по ней – граф переходов автомата Мили (рис. 10).

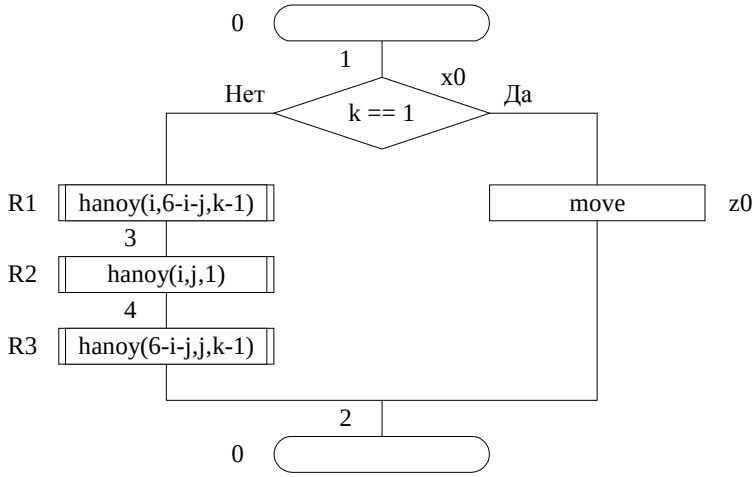

Рис. 9. Схема программы решения задачи о ханойских башнях

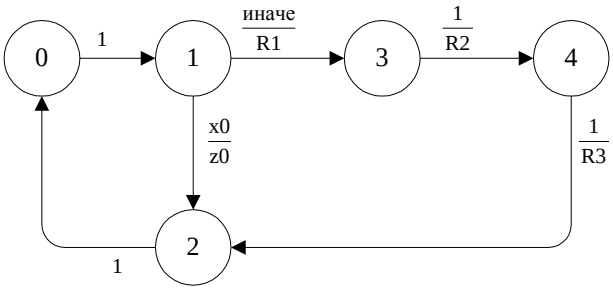

Рис. 10. Граф переходов до преобразования

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. 11 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.

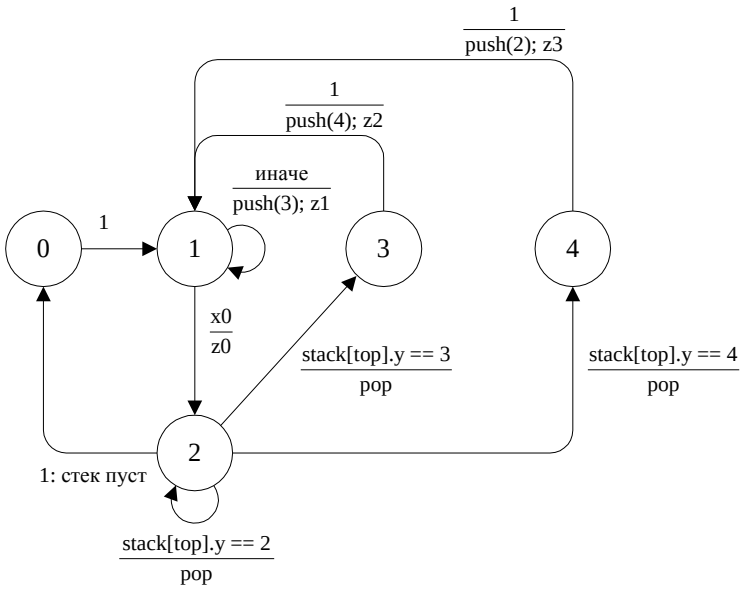

Рис. 11. Преобразованный граф переходов

Автоматная программа, построенная по графу переходов автомата Мили с пятью состояниями (рис. 11), приведена в листинге 8. В этой программе операторы, реализующие дуги, исходящие из вершины с номером 2 (за исключением дуги 2–0), 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 8. 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 ) ;
}
```

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

Упростим граф переходов, приведенный на рис. 11, за счет исключения неустойчивых вершин с номерами 3 и 4 (рис. 12).

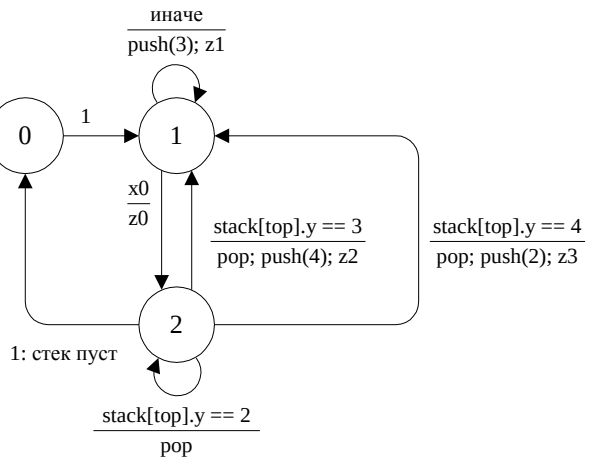

Рис. 12. Упрощенный граф переходов

При построении программы по графу переходов автомата Мили с тремя состояниями (рис. 12), функция hanoy() реализуется как показано в листинге 9. Остальная часть программы остается неизменной по сравнению с листингом 8. Отметим, что при такой реализации не удается выполнить ее дальнейшее упрощение за счет замены трех условий переходов одной функцией pop.

ЛИСТИНГ 9. 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 10.

ЛИСТИНГ 10. 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
```

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 [5]. 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 11), эта задача рассматривается в настоящей работе для иллюстрации предлагаемого метода, так как соответствующая ей программа, построенная на основе программы, предложенной в работе [5], отличается весьма сложной логикой по сравнению с приведенными выше.

ЛИСТИНГ 11. 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 } ;
int     N = sizeof(items)/sizeof(item_t) ;
enum    { knap_cap = 17 } ;
int     maxKnown[knap_cap+1] ;
item_t  itemKnown[knap_cap+1] ;

int     rec_level_prev = 0 ;
int     rec_count = 0 ;

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

  // Построение дерева рекурсивных вызовов.
  int j ;
  if( rec_level_prev >= rec_level )
  {
    printf( "\n" ) ;
    for( j = 0 ; j < rec_level ; j++ ) printf( "   " ) ;
  }
  printf( "%2d ", cap ) ;
  rec_level_prev = rec_level ;
  rec_count++ ;

  if( maxKnown[cap] != -1 ) return maxKnown[cap] ;
  for( i = 0, max = 0 ; i < N ; i++ )
  {
    space = cap - items[i].size ;
    if( space >= 0 )
    {
      t = knap( space, rec_level+1 ) + items[i].val ;
```

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

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

  return max ;
}

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

  printf( "\nВызовов функции knap: %d\n", rec_count ) ;
  printf( "Массив maxKnown:\ncap           " ) ;
  for( i = 0 ; i <= knap_cap ; i++ ) printf( "%2d ", i ) ;
  printf( "\nmaxKnown[cap] ") ;
  for( i = 0 ; i <= knap_cap ; i++ ) printf( "%2d ", maxKnown[i] ) ;
  printf( "\n\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() ;
  show_knap( knap_cap, knap(knap_cap, 0) ) ;
}
```

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. 13). 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.

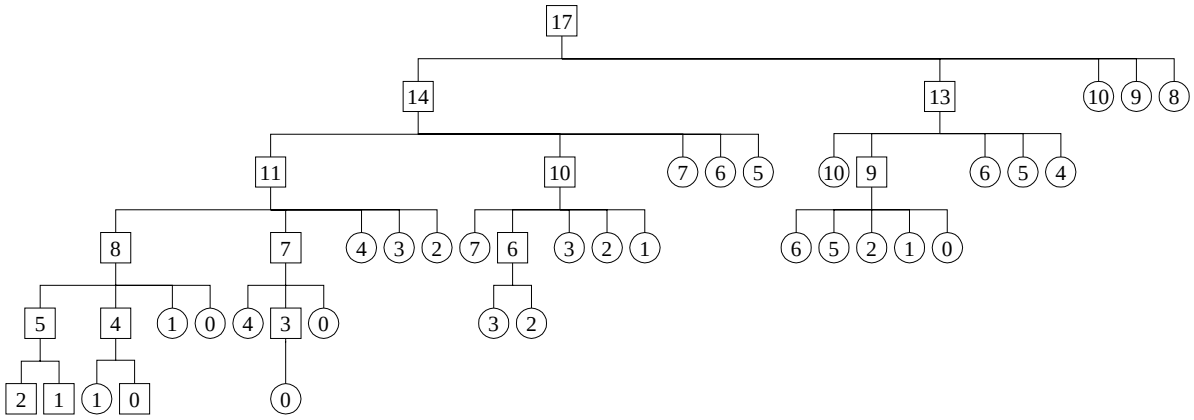

Рис. 13. Дерево рекурсивных вызовов при решении задачи о ранце

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

ТАБЛИЦА. 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. 13), 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 [5].

Построим схему рекурсивной программы (рис. 14), в которой не учитывается код, использованный для построения дерева рекурсивных вызовов.

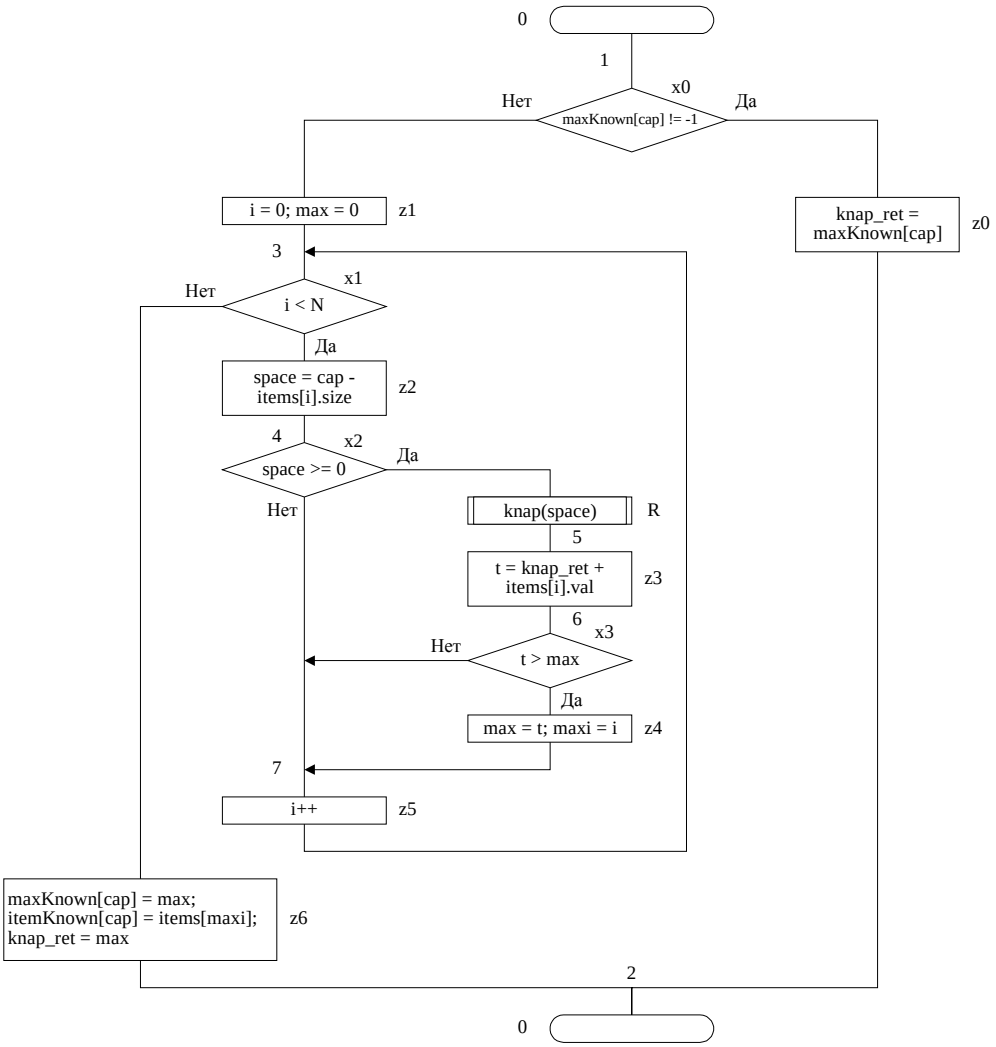

Рис. 14. Схема программы решения задачи о ранце

По схеме программы (рис. 14) построим граф переходов автомата Мили (рис. 15).

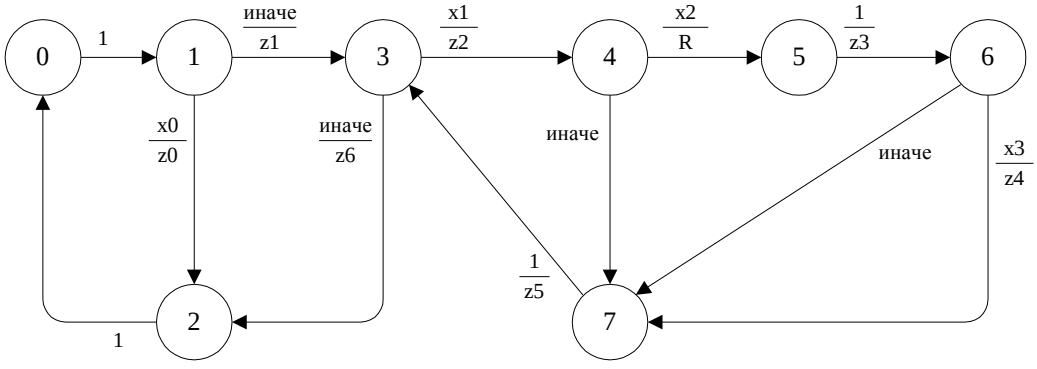

Рис. 15. Граф переходов до преобразования

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. 16).

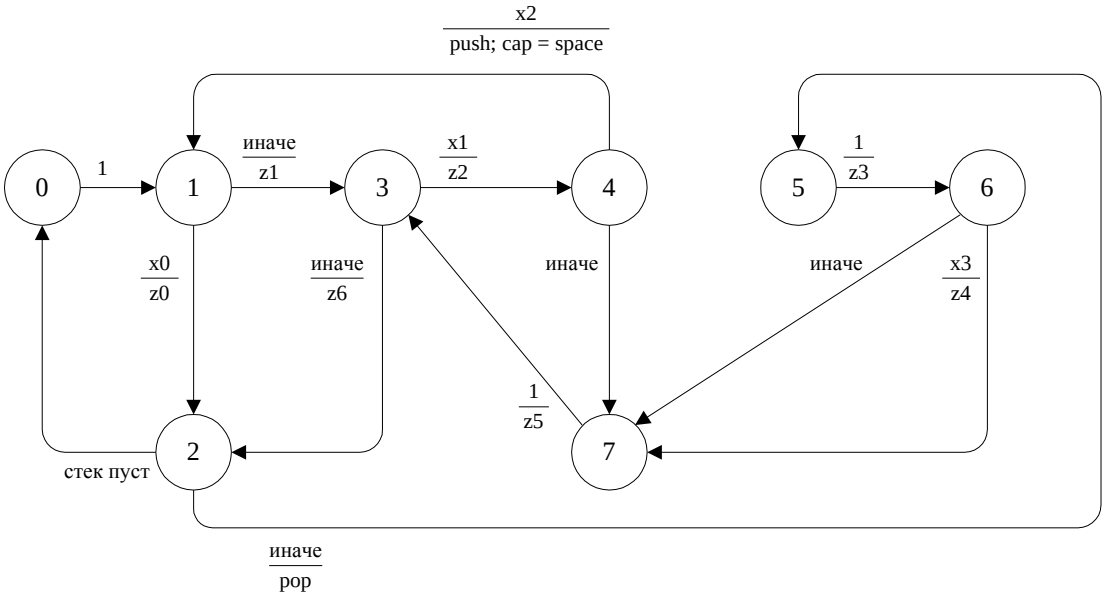

Рис. 16. Преобразованный граф переходов

Упростим этот граф за счет исключения неустойчивой вершины с номером 5 (рис. 17).

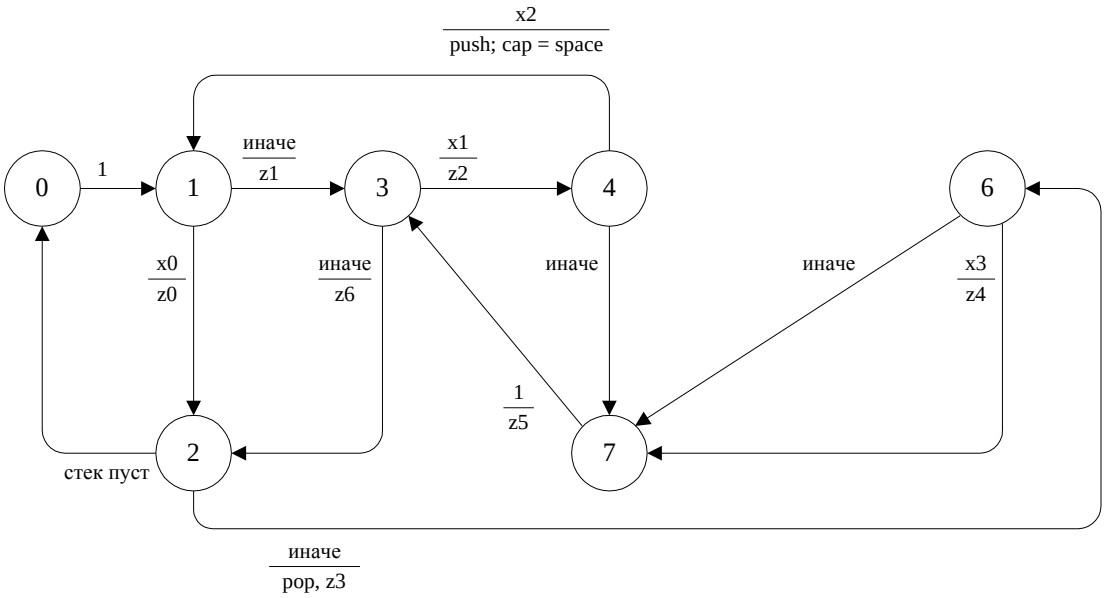

Рис. 17. Упрощенный граф переходов

Автоматная программа, построенная по графу переходов автомата Мили с семью состояниями (рис. 17) приведена в листинге 12.

ЛИСТИНГ 12. Автоматная программа 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 = 0 ;
```

```
  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 ) ;
}

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 ) ;
}
```

Результат работы этой программы, идентичный результату работы рекурсивной программы, приведен в листинге 13.

ЛИСТИНГ 13. 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
```

Заключение

В работе [2] было отмечено, что итеративные алгоритмы по вычислительной мощности эквивалентны рекурсивным. Однако, в известной авторам литературе, формальный метод преобразования рекурсивных алгоритмов в итеративные отсутствует. Как отмечалось во введении, в работах [1,6] приведены примеры такого преобразования, которые, однако, не являются формализованными. Настоящая работа устраняет указанный пробел. При этом развивается основанный на использовании конечных автоматов подход, предложенный в работе [8].

Отметим, что автоматы, которые строятся с помощью предлагаемого подхода, зависят от глубокой предыстории — в отличие от классических автоматов, переходы из одного из состояний (с номером 2) определяются ранее запомненным в стеке номером следующего состояния.

Завершая работу отметим, что предлагаемый подход позволяет по-новому взглянуть на понятие "состояние" в программировании. Обычно под состоянием программы понимается значение ячеек ее памяти [3,4]. Однако, такое определение не конструктивно, так как практически в любой программе, реализующей вычислительный алгоритм, число указанных значений чрезвычайно велико. Для решения этой проблемы обратим внимание на то, что в машине Тьюринга небольшое количество состояний в автомате может управлять произвольным количеством состояний на ленте [7]. Такая же ситуация имеет место и в рассмотренных

примерах. Например, в задаче о ханойских башнях автомат, содержащий три состояния, управляет 2^N состояниями "объекта управления", где N – число дисков.

С другими аспектами автоматного программирования можно ознакомиться на сайте **http://is.ifmo.ru**.

Список литературы

1. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. М.: Вильямс, 2000.
2. Брукшир Дж. Введение в компьютерные науки. М.: Вильямс, 2001.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, СПб.: Невский диалект, 1998.
4. Лавров С.С. Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001.
5. Седжвик Р. Фундаментальные алгоритмы на C++. Киев: ДиаСофт, 2001.
6. Стивенс Р. Delphi. Готовые алгоритмы. М.: ДМК, 2001.
7. Шалыто А.А., Туккель Н.И. От тьюрингова программирования к автоматному //Мир ПК. 2002. №2. (**http://is.ifmo.ru**, раздел "Статьи")
8. Шалыто А.А., Туккель Н.И. Преобразование итеративных алгоритмов в автоматные //Программирование. 2002. №5. (**http://is.ifmo.ru**, раздел "Статьи")

ОБ АВТОРАХ

Шалыто Анатолий Абрамович – ученый секретарь Федерального научно-производственного центра (ФНПЦ) – федерального государственного унитарного предприятия (ФГУП) "НПО "Аврора"", профессор кафедры "Компьютерные технологии" СПбГИТМО (ТУ). С ним можно связаться по адресу: Shalyto@mail.ifmo.ru.

Туккель Никита Иосифович – инженер-программист ФНПЦ – ФГУП "НПО "Аврора"". С ним можно связаться по адресу: cynical@mail.ru.

Шамгунов Никита Назимович – аспирант СПбГИТМО (ТУ).