Опубликовано в журнале “Телекоммуникации и информатизация образования”, 2003, № 6, стр. 27–58

Разработка логики визуализаторов алгоритмов на основе конечных автоматов

М.А. Казаков, Г.А. Корнеев, А.А. Шалыто
Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
E-mail: kgeorgiy@rain.ifmo.ru

1. Введение

При изучении алгоритмов обработки информации, представляемой различными структурами данных, [1, 2] важную роль играют визуализаторы алгоритмов, позволяющие в наглядной форме динамически отображать детали их работы.

Это открывает возможность использования новой технологии изучения дискретной математики и программирования [3, 4].

Визуализатор — это программа, в процессе работы которой на экране компьютера динамически демонстрируется применение алгоритма к выбранному набору данных. Визуализаторы позволяют изучать работу алгоритмов в пошаговом режиме, аналогичном режиму трассировки программ. Они при необходимости допускают трассировку укрупненными шагами, игнорируя рутинную часть вычислительного процесса, что существенно, например, для переборных алгоритмов.

Для некоторых алгоритмов динамический вариант демонстрации его работы является более естественным, чем набор статических иллюстраций. Для родственных алгоритмов (например, алгоритмов сортировки) визуализация позволяет наглядно продемонстрировать как общий подход, так и различие в механизмах их действия.

При изучении большинства алгоритмов наряду с режимом "шаг вперед" весьма полезен также и режим "шаг назад" [3], позволяющий более быстро и полно понять алгоритм. Например, в алгоритмах поиска с возвратом бывает необходимо сделать несколько шагов назад, для того чтобы понять, почему та или иная ветвь поиска отброшена.

Многолетний опыт построения и применения визуализаторов на кафедре “Компьютерные технологии” СПбГУ ИТМО показал, что они могут быть использованы как

основной инструмент преподавания указанных выше курсов [4], в частности, при дистанционном обучении (http://ips.ifmo.ru) [5].

Можно утверждать, что к настоящему времени основные достижения в проектировании визуализаторов относятся к сфере педагогики [3], а достижения в сфере технологии создания визуализаторов весьма скромны. В частности, отсутствует метод, позволяющий по алгоритму формально и единообразно создавать логику работы визуализатора. В работе [6] был предложен метод преобразования итеративных алгоритмов в автоматные. При этом было высказано предположение, что трудности с формальным построением визуализаторов связаны с тем, что в традиционном процедурном программировании понятие "состояние" явно не применяется, и поэтому "что" и "как" визуализировать каждый программист каждый раз решает заново и эвристически.

В автоматном программировании [7] состояния используются явно, и поэтому такое программирование было названо "программирование с явным выделением состояний" (http://is.ifmo.ru).

В работе [8] было предложено использовать особенности автоматных программ для построения визуализаторов.

В настоящей работе предлагается метод построения логики работы визуализатора по заданному алгоритму. Метод позволяет представить логику работы визуализатора системой взаимосвязанных конечных автоматов. Система состоит из пар автоматов, каждая из которых содержит “прямой” и “обратный” автоматы, которые обеспечивают пошаговое движение по алгоритму вперед и назад соответственно.

Некоторые шаги предлагаемого метода являются неформальными, например, реализация визуализируемого алгоритма на языке программирования.

Для описания логики работы визуализаторов выбраны автоматы Мура, в которых действия выполняются только в состояниях. Это позволяет отображать операции, выполняемые алгоритмом, наиболее естественным образом. При построении обратного автомата возможен переход к смешанному автомату, на ребрах которого выполняются только вспомогательные действия, но не шаги алгоритма.

2. Описание метода

Предлагаемый метод состоит из четырех шагов.

1. По визуализируемому алгоритму пишется процедурная программа, его реализующая.

2. Используемые переменные (в том числе, переменные циклов) выносятся в модель данных (например, в структуру, класс или запись). При этом процедурам, применяемым в программе, параметры должны передаваться через модель данных. Результаты работы процедур также должны передаваться через модель.
3. Программа преобразуется с целью использования в ней следующих операторов, последние пять из которых являются управляющими:
   - оператор присваивания;
   - последовательность операторов (составной оператор);
   - укороченный оператор ветвления (`if-then`);
   - полный оператор ветвления (`if-then-else`);
   - цикл с предусловием (`while`);
   - вызов процедуры.
4. Программа преобразуется в систему взаимосвязанных автоматов. При этом для каждой процедуры строится пара из прямого и обратного автоматов.

Прокомментируем шаги этого метода.

Первый шаг является неформальным. При построении программы не накладываются ограничения на используемые операторы.

Второй шаг необходим для того, чтобы полученную программу можно было разбить на систему автоматов, не заботясь о передаче значений локальных переменных между автоматами.

В результате третьего шага получается программа, более удобная для преобразования в систему автоматов, так как она не содержит громоздких синтаксических конструкций. Данное преобразование основано на теореме структурирования [9], в соответствии с которой любую программу можно преобразовать так, что она будет содержать только три типа управляющих конструкций: последовательность операторов, один из операторов ветвления и один из операторов цикла. Набор используемых управляющих конструкций может быть, как сокращен до последовательности операторов и оператора цикла с предусловием, так и расширен, например, как в настоящей работе, за счет введения процедур (также с одним входом и одним выходом) и применения двух типов оператора ветвления: полного и укороченного. Цикл с предусловием (while) выбран потому, что он является наиболее универсальным.

В результате применения описанного метода получается модель данных и система взаимосвязанных автоматов, управляющих этой моделью.

Далее подробно рассматривается выполнение четвертого шага предложенного метода, являющегося основным. При этом для каждой процедуры строится пара автоматов (прямой и

обратный). В разд. 3, 4 по процедуре создается прямой автомат. В разд. 5, 6 прямой автомат модифицируется и строится обратный автомат, а в разд. 7 рассматривается взаимодействие автоматов, соответствующих различным процедурам.

3. Преобразование процедуры в автомат

В результате выполнения третьего шага описываемого метода процедура содержит только следующие операторы: оператор присваивания, последовательность операторов, полный и укороченный операторы ветвления, цикл с предусловием и вызов процедуры.

Построение автомата по процедуре осуществляется последовательно. При этом для каждого оператора строится фрагмент автомата с одним входом и одним выходом, которые связывают его с другими фрагментами. На приведенных ниже рисунках входы и выходы фрагмента автомата будем представлять в виде входящей и исходящей стрелок, у которых начало (для входа) и конец (для выхода) не связаны ни с одним состоянием. Из фрагментов автомата, построенных для отдельных операторов, строятся блоки, соответствующие последовательности операторов.

3.1. Оператор присваивания

Оператор присваивания преобразуем в состояние автомата, в котором в качестве действий, выполняются присваивания. Например, оператор

```
d.m = max(d.m, d.a[d.i]);
```

преобразуется в состояние, как изображено на рис. 1.

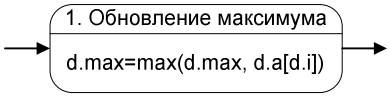


Рис. 1. Оператор присваивания

Здесь и далее над чертой, разделяющей на две части изображения состояния (вершины графа переходов), записаны номер состояния и его название, а под ней — действия, выполняемые в состоянии.

3.2. Последовательность операторов

Преобразование последовательности из двух операторов осуществляется “связыванием” стрелки, выходящей из фрагмента автомата, соответствующего первому оператору, и стрелки, входящей во фрагмент автомата, построенного для второго оператора. Преобразование последовательности из большего числа операторов осуществляется аналогично.

Например, последовательность операторов

```
m = max(m, a[i]);
m = min(m, b[i]);
```

преобразовывается во фрагмент автомата, изображенный на рис. 2.

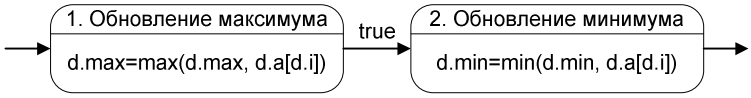


Рис. 2. Последовательность состояний

Обратим внимание, что дуга между вершинами помечена условием перехода `true`, что обозначает безусловный переход. В дальнейшем для экономии места и улучшения читаемости везде, где это целесообразно, условие `true` будем опускать.

Так как во фрагмент автомата, представляющей оператор или последовательность операторов, входит и выходит только одна стрелка, то такой фрагмент можно обозначить, как показано на рис. 3.

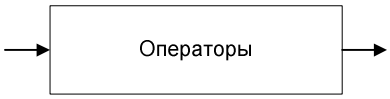


Рис. 3. Блок операторов

3.3. Оператор вызова процедуры

Оператор вызова процедуры преобразуем в состояние, в котором осуществляется вызов автомата, соответствующего ей. Например, вызов процедуры

```
findMin()
```

преобразуем, как показано на рис. 4.

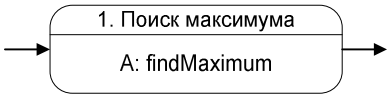


Рис. 4. Вызов процедуры

Здесь префикс “`A:`” обозначает вызов автомата. Более подробно вызовы процедур и автоматов будут рассмотрены в разд. 7.

3.4. Укороченный оператор ветвления

Укороченный оператор ветвления

```
if (expr) {
    Операторы
}
```

будем преобразовывать во фрагмент автомата, изображенный на рис. 5.

Рис. 5. Укороченный оператор ветвления

Переход происходит по той ссылке, условие для которой истинно. Здесь и далее `expr` обозначает условие, а `¬expr` — его отрицание.

3.5. Полный оператор ветвления

Полный оператор ветвления

```
if (expr) {
    Операторы 1
} else {
    Операторы 2
}
```

преобразуем, как показано на рис. 6.

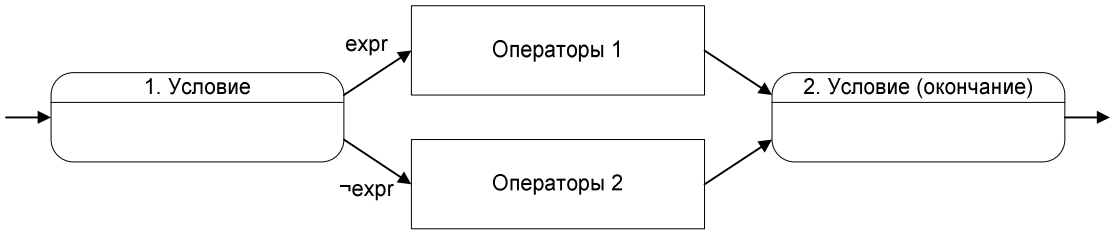


Рис. 6. Полный оператор ветвления

Обоснование введения состояния, помеченного как “Условие (окончание)”, будет приведено в разд. 5.2.

3.6. Цикл с предусловием

Цикл с предусловием

```
while (expr) {
    Операторы
}
```

преобразуем, как показано на рис. 7.

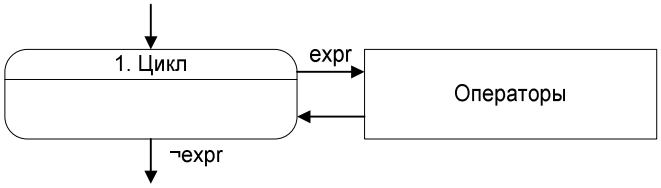


Рис. 7. Цикл с предусловием

3.7. Завершение построения автомата

Для завершения построения автомата к фрагменту, соответствующему телу процедуры, добавим начальное и конечное состояния. Начальное состояние связывается со свободной стрелкой, входящей во фрагмент автомата, а конечное — со свободной стрелкой, выходящей из него. Таким образом, получаем прямой автомат, соответствующий процедуре.

Для пошагового выполнения процедуры можно использовать управляемый тактовый генератор. При этом автомат совершает переход только при импульсе тактового генератора. В частности, можно применить тактовый генератор, производящий импульс только при нажатии пользователем кнопки “Переход к следующему шагу”.

4. Пример построения прямого автомата по процедуре

Несмотря на то, что визуализируемые алгоритмы являются процедурными, в целом визуализатор удобно реализовать как систему взаимосвязанных классов. При этом логика работы визуализатора реализуется по принципу: один автомат — один класс.

В этом и последующих примерах будем использовать язык *Java*.

Проиллюстрируем применение метода, изложенного в предыдущем разделе, на примере построения прямого автомата для процедуры поиска максимума в массиве натуральных чисел.

На первом шаге метода напишем следующую программу:

```
int max = 0;
for (int i = 0; i < a.length; i++) {
    if (max < a[i]) {
        max = a[i];
    }
}
```

В соответствии со вторым шагом предложенного метода, вынесем используемые переменные в модель данных — класс `Data`:

```
public final static class Data {
    public int max;
    public int i;
    public int a[];
}
```

Считая, что экземпляр класса `Data` доступен как переменная `d`, получим:

```
d.max = 0;
for (d.i = 0; d.i < d.a.length; d.i++) {
    if (d.max < d.a[d.i]) {
        d.max = d.a[d.i];
    }
}
```

В данной процедуре применен цикл `for`, который в соответствии с третьим шагом предлагаемого метода необходимого преобразовать в цикл с предусловием `while`:

```
d.max = 0;
d.i = 0;
while (d.i < d.a.length) {
    if (d.max < d.a[d.i]) {
        d.max = d.a[d.i];
    }
    d.i++;
}
```

В этой процедуре части выделенные курсивом соответствуют частям заголовка исходного цикла.

Преобразуем полученную процедуру в автомат. Два первых оператора превращаются во фрагмент автомата, изображенный на рис. 8.

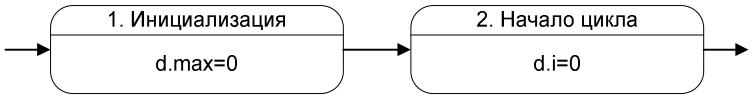


Рис. 8. Начальная инициализация

Прежде чем преобразовать цикл `while` во фрагмент автомата, необходимо построить фрагмент, соответствующий телу цикла. Для этого преобразуем оператор ветвления `if` (в данном случае — укороченный). Ему соответствует фрагмент автомата, изображенный на рис. 9.

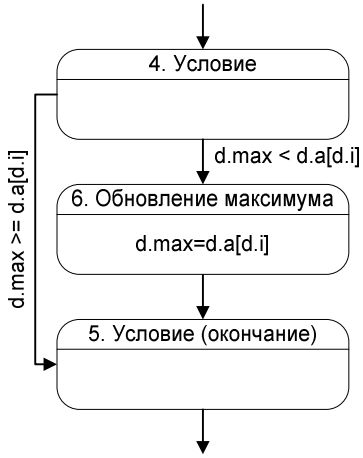


Рис. 9. Оператор ветвления

Присоединив к полученному фрагменту (состояния 4 – 6 на рис. 10) состояние 7, соответствующее оператору `d.i++`, получим преобразованное тело цикла (состояния 4 – 7). Заканчивая преобразование цикла `while`, добавим состояние 3.

Присоединяя к фрагменту, соответствующему циклу `while` (рис. 7), состояния 1 и 2, а также, вводя начальное и конечное состояния (0 и 8), получим автомат, изображенный на рис. 10.

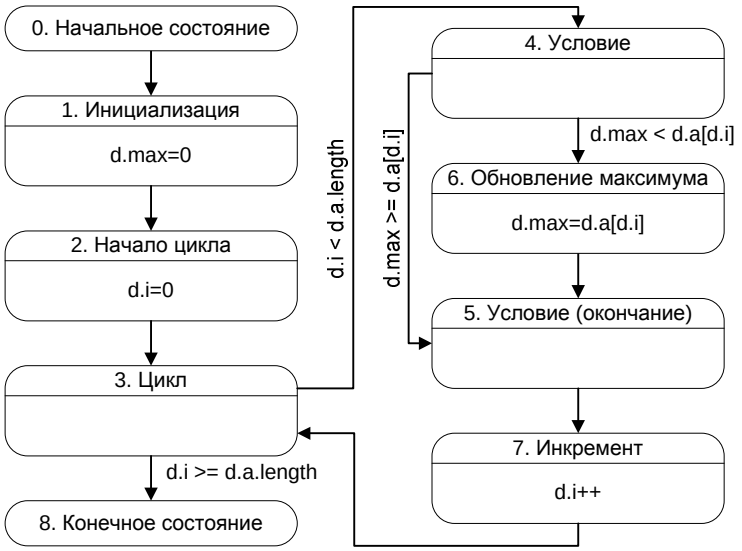


Рис. 10. Автомат поиска максимума

Полученный автомат Мура эквивалентен исходной процедуре по построению.

5. Построение обратного автомата

В предыдущих разделах было показано, как по процедуре получить эквивалентный автомат. Основная функция прямого автомата — перемещение по алгоритму вперед.

Существует несколько методов обеспечения возможности передвижения по алгоритму в обратную сторону.

1. *Метод пошагового сохранения*. Для каждого шага сохраняется значение всех переменных модели данных и состояние прямого автомата. При совершении шага назад значения, сохраненные на предыдущих шагах, восстанавливаются. Данный метод требует большей памяти по сравнению с методами 2 и 3.
2. *Метод пересчета*. При движении вперед запоминается количество шагов, сделанное автоматом. Для совершения шага назад прямой автомат перезапускается сначала и останавливается, сделав на один шаг меньше. Этот метод требует существенных затрат времени на повторные вычисления.
3. *Метод обращения*. Для передвижения в обратную сторону строится “обратный” автомат. Данный метод не требует ни большой памяти, ни дополнительных затрат времени на повторные вычисления, и будет рассмотрен ниже.

В предлагаемом подходе используется *метод обращения*, как наиболее эффективный. Заметим, что хотя этот метод внешне напоминает обращение программ, описанное в [10], однако эти методы различаются по сути.

Обратный автомат имеет одноименные состояния с прямым автоматом. Все переходы прямого автомата сохраняются, но направляются в противоположную сторону. Пометки переходов модифицируются, как будет изложено ниже. Построенную таким образом пару автоматов можно рассматривать, как обобщенный автомат с двумя функциями переходов и двумя наборами действий. При этом одна функция переходов и набор действий соответствуют движению по алгоритму вперед, а вторая функция и второй набор действий — движению назад.

В дальнейшем движение по алгоритму вперед будем называть *прямым проходом*, а движение назад — *обратным проходом*.

При *прямом проходе* остановка автомата для визуализации осуществляется перед каждым переходом (сразу после выполнения действия в состоянии). Соответственно, при *обратном проходе* остановку необходимо осуществлять непосредственно перед каждым состоянием. При использовании этого соглашения текущее состояние прямого и обратного автоматов можно описывать одним номером состояния.

Перейдем к изложению метода построения обратного автомата.

Последовательность операторов обращается путем выполнения обращений операторов в противоположном порядке. Таким образом, построение обратного автомата сводится к обращению отдельных операторов.

Обращение оператора можно произвести двумя способами.

1. *Непосредственный способ.* Состояние автомата и значения переменных модели данных вычисляются непосредственно по их значениям на последующем шаге.
2. *Косвенный способ.* Состояние автомата и значения переменных модели данных восстанавливаются по информации, запомненной при прямом проходе, как описано ниже.

Обращение *непосредственным способом* требует неформального подхода, но позволяет экономить память.

Обращение *косвенным способом* может быть произведено формально. При этом необходимо производить модификацию прямого автомата, с тем, чтобы он сохранял информацию, необходимую при *обратном проходе*.

В дальнейшем, вместо выражений *обращение непосредственным способом* и *обращение косвенным способом*, будем применять выражения *непосредственное обращение* и *косвенное обращение* соответственно.

5.1. Обращение оператора присваивания

В различных ситуациях выгодно использовать тот или иной метод. Проиллюстрируем это на примере обращения оператора присваивания.

Косвенное обращение оператора присваивания осуществляется помещением замещаемого значения переменной в стек при *прямом проходе* и извлечением его из стека при *обратном проходе*. Стек подходит для сохранения замещаемых значений, так как они используются в обратном порядке по сравнению с порядком запоминания.

В дальнейшем этот стек будем называть *общим стеком*, так как он будет использован при обращении и других операторов. На рисунках *общий стек* будем обозначать как `stack`.

Для стеков будем применять следующие операции:

- `push(expr)` — поместить значение выражения `expr` на вершину стека;
- `peek()` — прочитать значение на вершине стека;
- `pop()` — прочитать значение на вершине стека и удалить его.

Таким образом, при *прямом проходе* прямой автомат помещает замещаемое значение в *общий стек*, а при *обратном проходе* сохраненное значение извлекается из *общего стека*.

Приведем пример косвенного обращения оператора присваивания для оператора, осуществляющего обновление максимума (из процедуры поиска максимума в разд. 4):

```
d.max = d.a[d.i];
```

Этот оператор обращается, как показано на рис. 11.

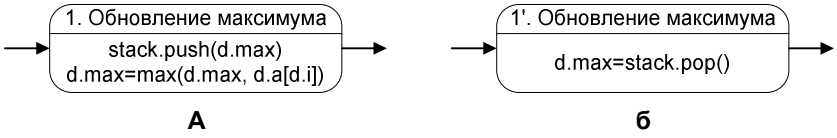


Рис. 11. Оператор присваивания (а) и его косвенное обращение (б)

Здесь и далее будем помечать состояния обращенного автомата добавлением штриха к его номеру.

Отметим, что обращение данного оператора *непосредственным способом* затруднено тем, что для вычисления предыдущего значения максимума необходимо знать все предыдущие значения в массиве, а если бы процедура обнуляла просмотренный элемент массива, то это было бы вообще невозможно.

В тоже время *непосредственное обращение* оператора

```
d.i++;
```

осуществляющего переход к следующему элементу (из процедуры поиска максимума в разд. 4), выполняется оператором

```
d.i--;
```

Таким образом, при прямом проходе единица прибавляется, а при обратном проходе — вычитается. Обращение данного оператора выглядит, как показано на рис. 12.

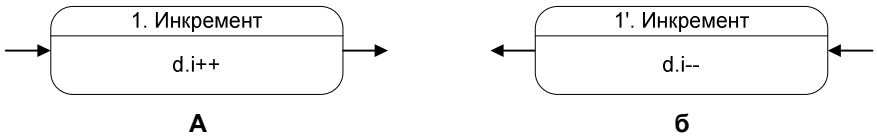


Рис. 12. Оператор присваивания (а) и его непосредственное обращение (б)

5.2. Обращения операторов ветвления

При обращении оператора ветвления возникает следующая проблема: решение о выборе ветви для обратного прохода невозможно принять в состоянии, в котором принимается решение при прямом проходе, так как в этот момент обращение ветви уже завершено.

Данная проблема решается введением состояния “Условие (окончание)” во фрагменты автоматов, соответствующих оператору ветвления. Здесь и далее будем называть состояние, помеченное как “Условие”, *открывающим* состоянием оператора ветвления, а состояние, помеченное как “Условие (окончание)” — *закрывающим*. Таким образом, решение о выборе ветви при обратном проходе принимается в *закрывающем состоянии*. Так же, как и при обращении оператора присваивания, существует два способа произвести этот выбор: *непосредственный* и *косвенный*.

При *непосредственном способе* после обращении операторов ветвления получаются фрагменты автоматов, приведенные на рис. 13, рис. 14.

Рис. 13. Непосредственное обращение укороченного оператора ветвления

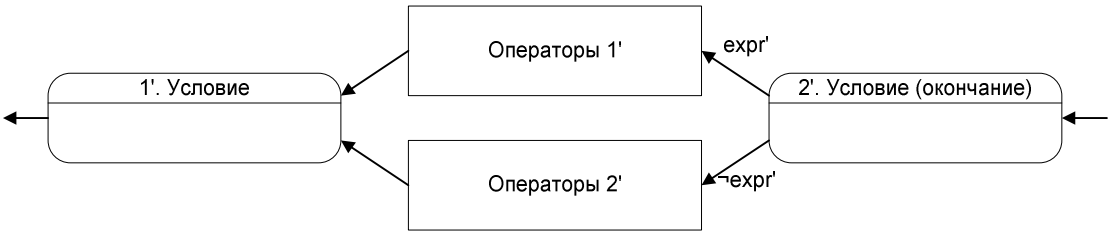


Рис. 14. Непосредственное обращение полного оператора ветвления

Здесь и далее под `expr'` подразумевается условие, позволяющее выбрать ветвь для обратного прохода.

При *косвенном способе* необходимо запоминать ветвь, выбранную при прямом проходе. Существует два варианта сохранения данной информации. В *первом варианте* обратный автомат является смешанным [8] (действия могут выполняться не только в состояниях, но и на переходах). При этом прямой автомат также преобразуется в смешанный. *Второй вариант* требует применения двух стеков, но получаемые автоматы являются автоматами Мура.

Рассмотрим *первый вариант*. Будем запоминать информацию о выбранной ветви при переходе в состояние, соответствующее окончанию оператора ветвления. Таким образом, данную информацию можно будет использовать для принятия решения при обратном проходе.

Фрагменты прямого и обратного автоматов для *первого варианта косвенного обращения* оператора ветвления приведены на рис. 15, рис. 16.

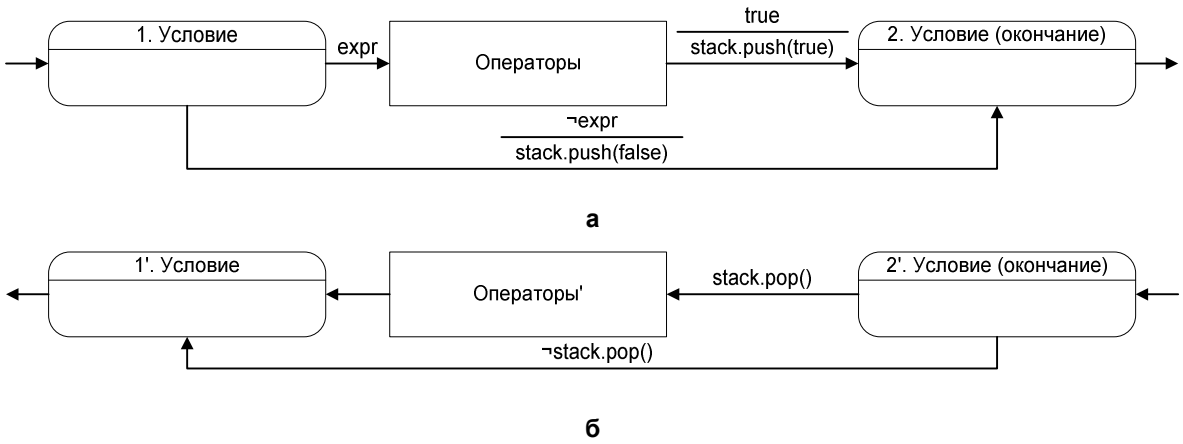


Рис. 15. Укороченный оператор ветвления (а) и первый вариант его косвенного обращения (б)

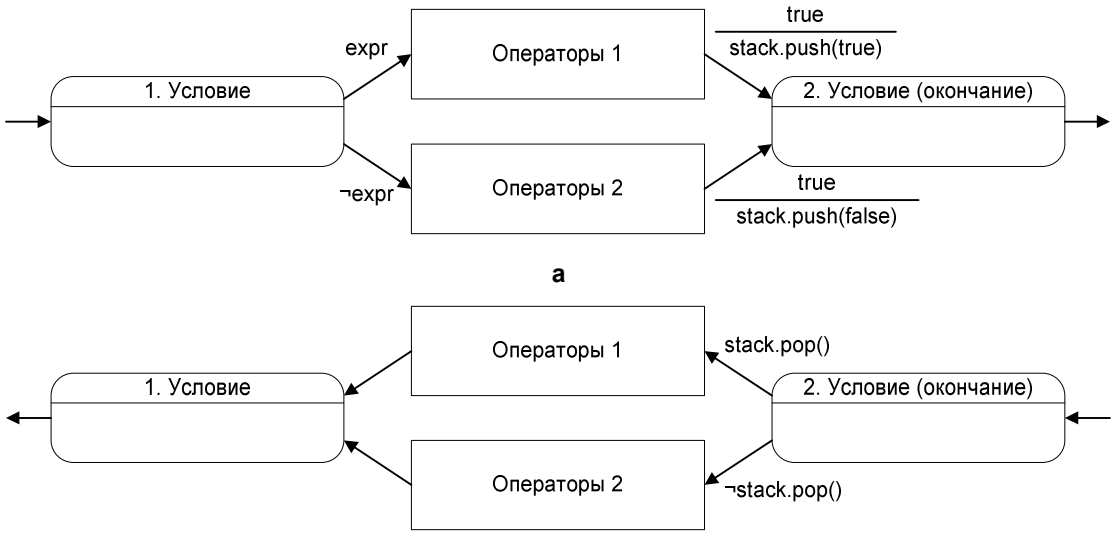


Рис. 16. Полный оператор ветвления (а) и первый вариант его косвенного обращения (б)

Здесь и далее на переходах под чертой указываются выполняемые действия.

Функция `stack.pop()` при работе удаляет значение из вершины стека. Если выполнять действие при проверке условия нежелательно, то применение функций `stack.pop()` и `¬stack.pop()` следует заменить на $\frac{\text{stack.peek()}}{\text{stack.pop()}}$ и $\frac{\neg\text{stack.peek()}}{\text{stack.pop()}}$ соответственно.

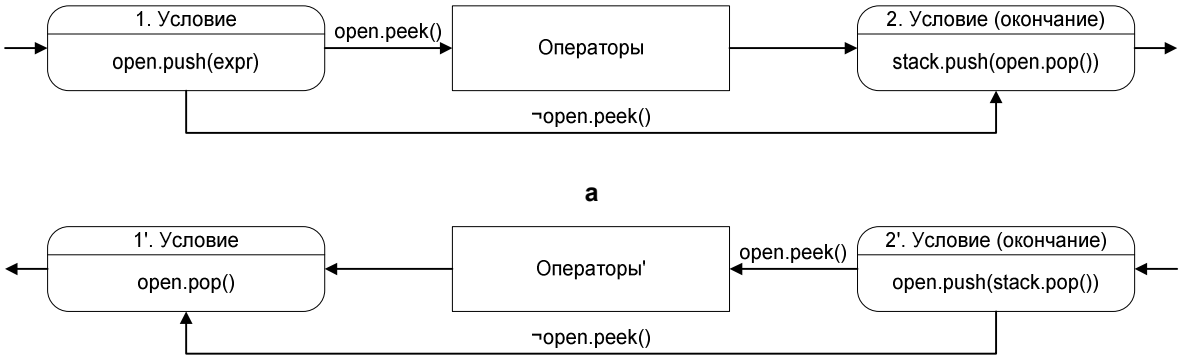
Таким образом, в *первом варианте косвенного обращения* операторов ветвления строятся фрагменты смешанного автомата.

Перейдем ко *второму варианту*. Информацию о ветви, выбранной при прямом проходе, нельзя запоминать в стеке непосредственно при ее выборе, так как тогда сохранять и использовать эту информацию необходимо в различных состояниях автомата (в *открывающем* и *закрывающем* соответственно), что недопустимо. Данную проблему можно

обойти, используя *стек открытых ветвлений* — ветвлений, для которых пройдено *открывающее* и еще не пройдено *закрывающее* состояние. На рисунках *стек открытых ветвлений* будем обозначать как `open`.

При *прямом проходе* в *открывающем* состоянии оператора ветвления в *стек открытых ветвлений* помещается флаг, соответствующий выбранной ветви (`true`, если условие выполнено, и `false`, если не выполнено). В *закрывающем* состоянии оператора ветвления данное значение переносится из *стека открытых ветвлений* в *общий стек*. При *обратном проходе* в *закрывающем* состоянии оператора ветвления значение из вершины *общего стека* переносится в *стек открытых ветвлений*. Если перенесенное значение равно `true`, то для обратного прохода выбирается положительная ветвь оператора ветвления, а если `false` — отрицательная.

Фрагменты прямого и обратного автоматов для *второго варианта косвенного обращения* оператора ветвления приведены на рис. 17, рис. 18.



а

Б

Рис. 17. Укороченный оператор ветвления (а) и второй вариант его косвенного обращения (б)

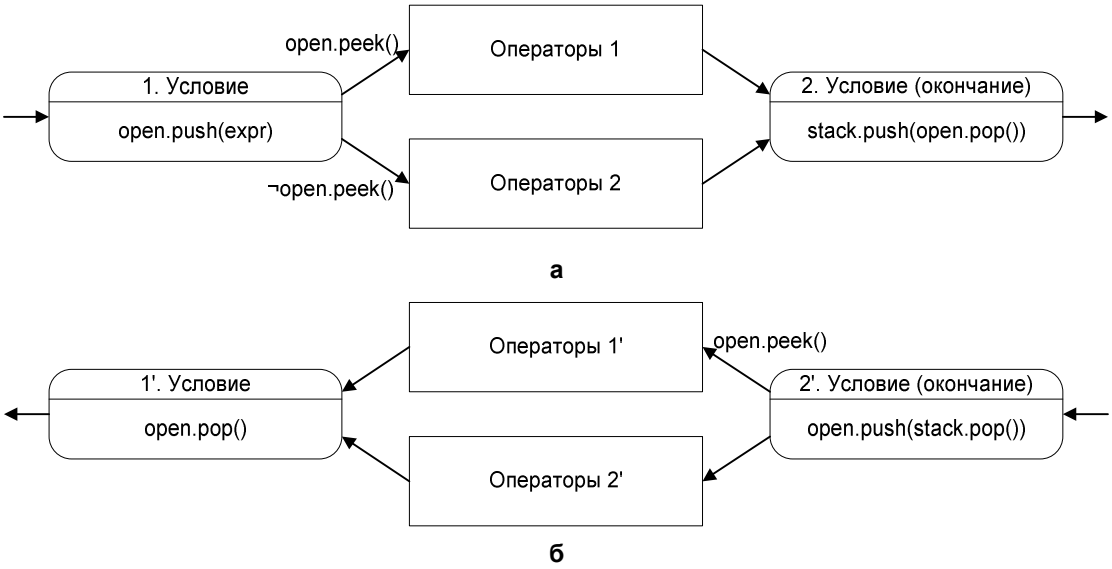


Рис. 18. Полный оператор ветвления (а) и второй вариант его косвенного обращения (б)

Таким образом, во *втором варианте косвенного обращения* операторов ветвления строятся фрагменты автоматов Мура.

5.3. Обращение оператора цикла с предусловием

Так же, как и для других операторов, обращение цикла с предусловием можно производить *непосредственным* и *косвенным способами*.

Обращение цикла с предусловием *непосредственным способом* приведено на рис. 19.

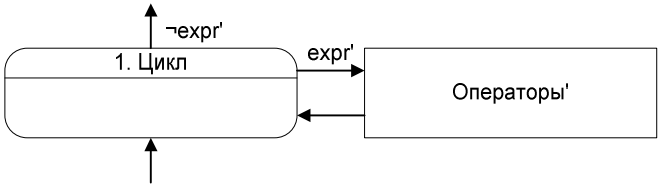
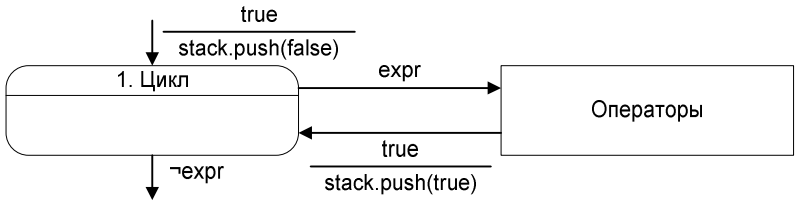


Рис. 19. Непосредственное обращение цикла с предусловием

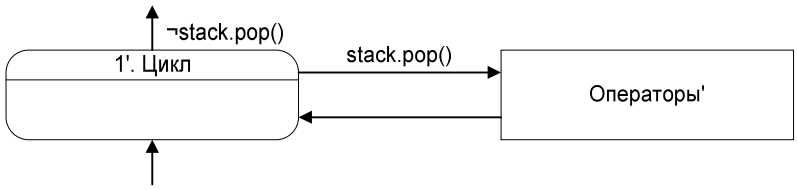
Косвенное обращение цикла с предусловием также можно осуществить в двух вариантах: с образованием фрагментов смешанных автоматов и с образованием фрагментов автоматов Мура, но требующим двух стеков.

Рассмотрим *первый вариант косвенного обращения* цикла с предусловием. При *прямом проходе* будем сохранять в стеке информацию о том, был ли переход в состояние цикла совершен извне (`false`) или из тела цикла (`true`). Соответственно, при *обратном проходе* данную информацию можно будет использовать для определения было ли тело цикла выполнено при *прямом проходе*. При этом определяется надо ли выполнить “обращенное” тело цикла еще раз или следует выйти из цикла.

Фрагменты прямого и обратного автоматов для *первого варианта косвенного обращения* цикла с предусловием приведены рис. 20.



а



б

Рис. 20. Цикл с предусловием (а) и первый вариант его косвенного обращения (б)

Второй вариант косвенного обращения оператора цикла с предусловием требует введения двух дополнительных состояний: “Цикл (начало)” и “Цикл (окончание)”. В дальнейшем эти состояния будем назвать *открывающее* и *закрывающее* соответственно, а состояние “Цикл” будем называть *основным*.

Рассмотрим *прямой проход*. В начальном состоянии в *стек открытых ветвлений* помещается значение `false`. В *основном* состоянии значение из вершины *стека открытых ветвлений* переносится в *общий стек*, а в *стек открытых ветвлений* помещается значение `true`. В *заключительном* состоянии значение из вершины *стека открытых ветвлений* удаляется.

Перейдем к рассмотрению *обратного прохода*. В *конечном* состоянии в *стек открытых ветвлений* заносится значение `true`. В *основном* состоянии значение из вершины *стека открытых ветвлений* удаляется. После этого значение из вершины *общего стека* переносится в *стек открытых ветвлений*. Затем для определения необходимости выполнения “обращения” тела цикла анализируется значение в вершине *стека открытых ветвлений*. Если это значение `true`, то осуществляется переход к “обращению” тела цикла, в противном случае осуществляется переход в *начальное* состояние. В *начальном* состоянии значение `false` удаляется из вершины *стека открытых ветвлений*.

Фрагменты прямого и обратного автоматов для *второго варианта косвенного обращения* цикла с предусловием приведены на рис. 21.

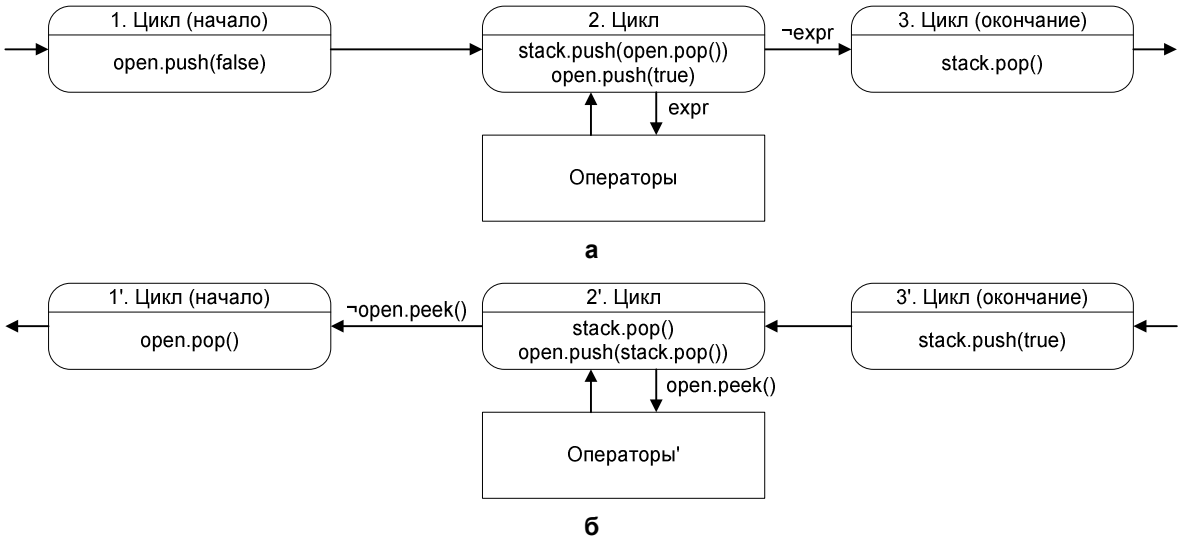


Рис. 21. Цикл с предусловием (а) и второй вариант его косвенного обращения (б)

5.4. Классификация вариантов построения обратного автомата

В подразделах 5.1 – 5.3 рассмотрены различные варианты построения обратного автомата. Обобщим сведения о предложенных вариантах в таблице.

Таблица

Особенности обращения	**Непосредственное обращение**	**Косвенное обращение**		
		Оператор присваивания	**Первый вариант**	**Второй вариант**
1	**2**	**3**	**4**	**5**
Операторы	Все	Присваивание	Ветвления и цикл	
Формализм	Неформальный	Формальный	Формальный	Формальный
Тип автоматов	Мура	Мура	Смешанный	Мура
Количество стеков	0	1	1	2
Модификация прямого автомата	Не требуется	Требуется		

Построить обратный автомат по процедуре можно, как формальным, так и неформальным способом. Формальное построение может быть осуществлено в двух вариантах. В первом случае для обращения операторов используются столбцы 3 и 4 таблицы, а во втором — столбцы 3 и 5. Соответственно, в первом случае получается пара смешанных автоматов, использующих *общий стек*, а во втором варианте — пара автоматов Мура, использующих *общий стек* и *стек открытых ветвлений*.

6. Пример построения обращенного автомата

В разд. 4 был построен прямой автомат для процедуры поиска максимума. Модифицируем его и построим обратный автомат, как изложено в разд. 5.

Оператор, осуществляющий обновление максимума, обратим *косвенным способом* (столбец 3 таблицы), так как его обращение *непосредственным способом* требует просмотра всех предыдущих значений массива.

По тем же причинам оператор ветвления обратим *косвенным способом*. Для обращения выберем вариант с образованием смешанного автомата (столбец 4 таблицы). Обращенный оператор ветвления приведен на рис. 22.

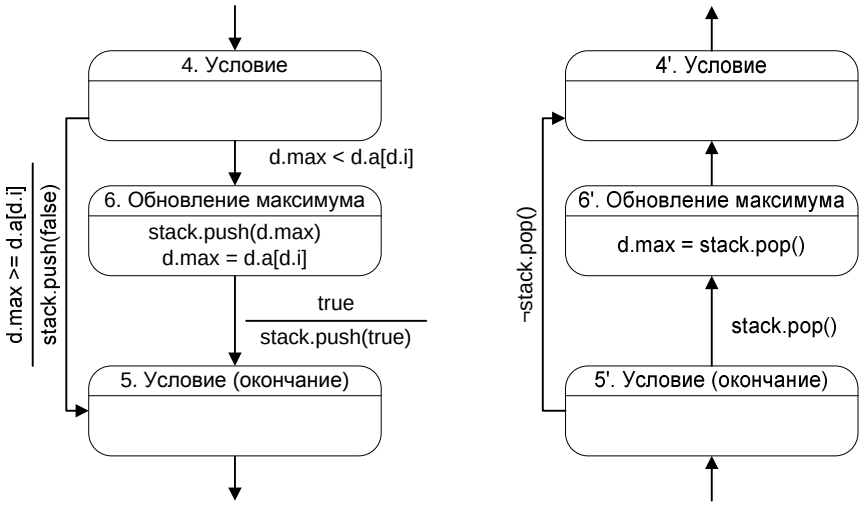


Рис. 22. Оператор ветвления (а) и его обращение (б)

Оператор, осуществляющий переход к следующему элементу, обратим *непосредственным способом* (столбец 2 таблицы). Также *непосредственным способом* обратим цикл (обращение условия продолжения цикла `d.i > 0`). Обращенный цикл соответствует состояниям 3 – 7 на рис. 23, рис. 24.

Заметим, что при *обратном проходе* после прохождения цикла значения переменных `d.max` и `d.i` равны нулю. Воспользовавшись этим, обратим операторы, осуществляющие инициализацию, *непосредственным способом*. При этом в соответствующих состояниях обратного автомата действия выполняться не будут.

Завершим построение обратного автомата добавлением начального и конечного состояний. Полученная пара автоматов изображена на рис. 23, рис. 24.

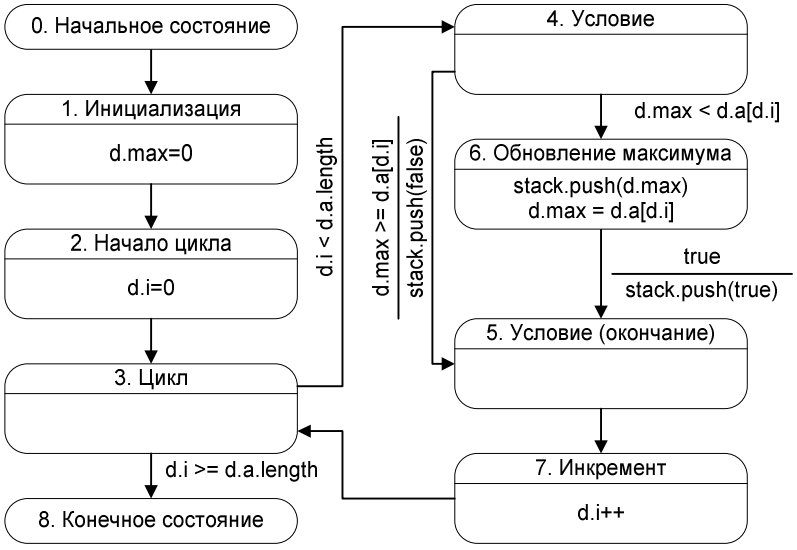


Рис. 23. Прямой автомат поиска максимума

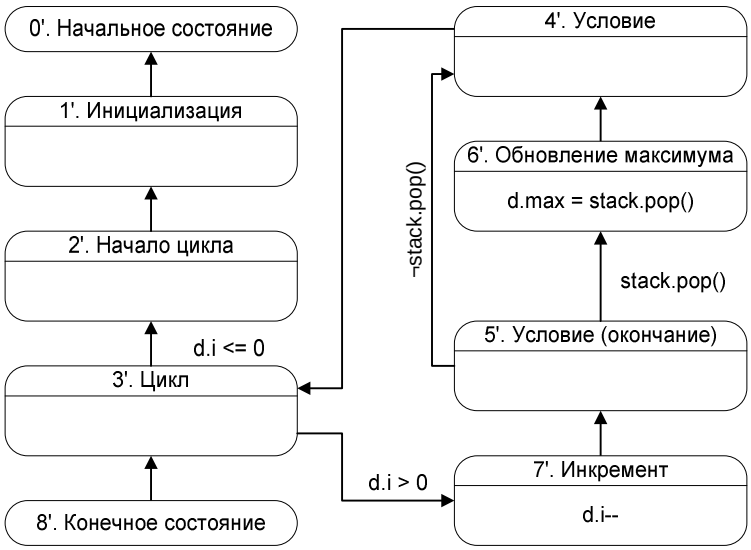


Рис. 24. Обратный автомат поиска максимума

Покажем, как произвести построение обратного автомата формально. Для этого будем обращать все операторы косвенным способом (столбцы 3 – 5 таблицы).

Пара автоматов, построенная при использовании столбцов 3 и 4 таблицы, приведена на рис. 25 и рис. 26.

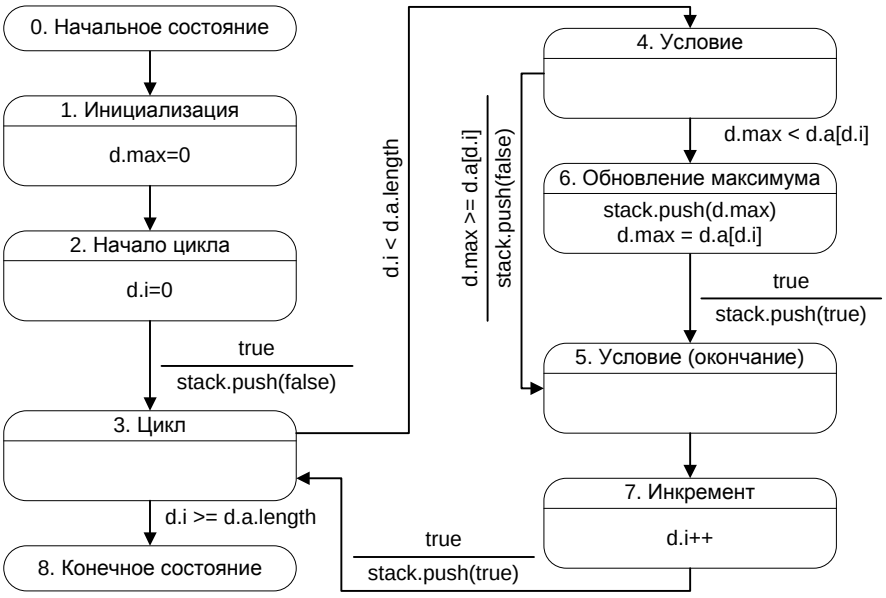


Рис. 25. Прямой автомат поиска максимума (второй вариант)

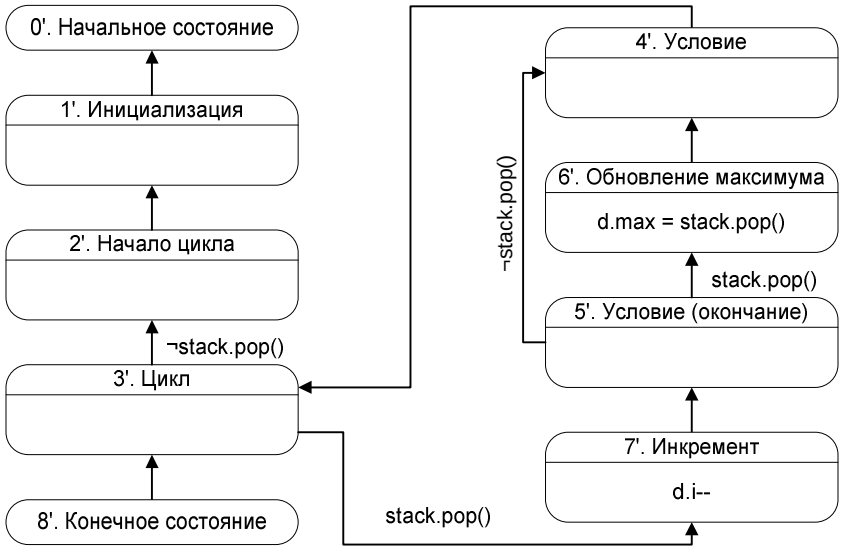


Рис. 26. Обратный автомат поиска максимума (второй вариант)

В заключение раздела отметим, что в рассмотренном примере при разных видах обращения получаются различные пары из автоматов, что является спецификой предлагаемого метода.

7. Процедуры и вызовы автоматов

Как было отмечено выше, если реализация визуализируемого алгоритма представляет собой набор процедур, то каждая из них преобразуется отдельно. При этом полученные автоматы взаимодействуют между собой посредством *вызовов* друг друга.

Так как автоматы для разных процедур строились независимо, то для осуществления взаимодействия их необходимо модифицировать. При этом возможны два случая:

- для итеративных (не рекурсивных) программ;
- для рекурсивных программ.

Первый случай более прост, но имеет существенное ограничение на класс преобразуемых программ. Второй случай сложнее, но позволяет преобразовать как итеративные, так и рекурсивные программы.

7.1. Итеративные программы

7.1.1. Преобразование итеративной программы

Пронумеруем процедуры некоторым способом. Обозначим прямой и обратный автоматы для процедуры номер *i* через *Ai* и *Ai'* соответственно.

Введем несколько условий для связи между автоматами:

- `state(Ai)==j` — автомат Ai находится в состоянии с номером j;
- `isAtStart(Ai)` — автомат Ai находится в начальном состоянии;
- `isAtEnd(Ai)` — автомат Ai находится в конечном состоянии.

Так как автоматы *Ai* и *Ai'* всегда находятся в одном и том же состоянии (разд. 5), то аналогичные условия для обратного автомата не вводятся.

Как было сказано выше, каждая пара автоматов соответствует одной процедуре, а каждому вызову этой процедуры (кроме главной) соответствует некоторое состояние. Составим *список вызовов* автомата *Ai*. Он состоит из пар (*Aj*, *k*), где *Aj* — автомат, который осуществляет вызов автомата *Ai*, а *k* — номер состояния, в котором осуществляется вызов. Заметим, что если автомат *Aj* содержит несколько вызовов автомата *Ai*, то ему соответствует несколько пар указанного вида. Для каждой пары из этого списка построим условие `state(Aj)==k`. Таким образом, всему *списку вызовов* будет соответствовать дизъюнкция условий, построенных для каждой пары. Обозначим полученное условие как *R(Ai)*.

Модифицируем автоматы, соответствующие всем процедурам, кроме главной, следующим образом. Для прямого автомата *Ai*:

- к переходу из *начального состояния* добавим условие *R(Ai)*;
- введем безусловный переход из *конечного состояния* в *начальное*.

Для обратного автомата *Ai'*:

- к переходу из *конечного состояния* добавим условие *R(Ai)*;
- введем безусловный переход из *начального состояния* в *конечное*.

После этого преобразуем состояния, в которых осуществляется вызов автоматов. Пусть в состоянии *j* автомата *Ai* осуществляется вызов автомата *Ak* (рис. 27).

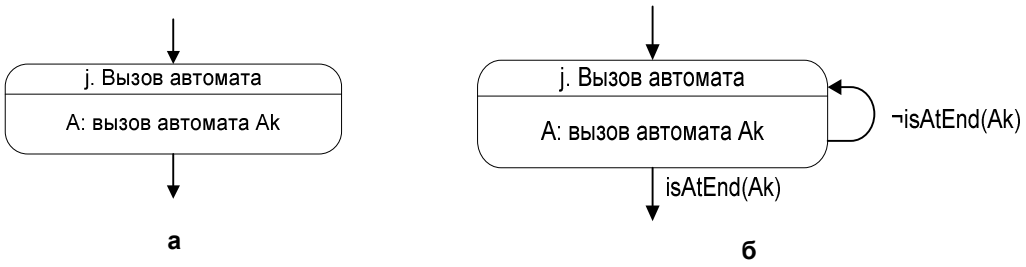


Рис. 27. Состояние прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

Преобразуем их, как показано на рис. 28.

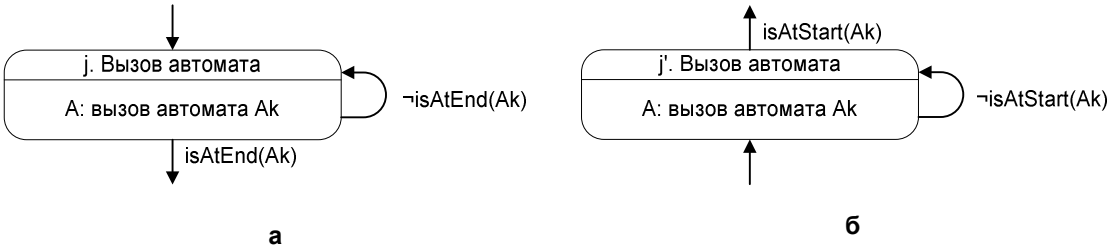


Рис. 28. Преобразованные состояния прямого (а) и обратного (б) автоматов, из которых вызывается другой автомат

Преобразовав все состояния, в которых осуществляются вызовы процедур изложенным выше образом, получим итоговый набор автоматов (по два автомата на процедуру).

Рассмотрим работу построенных автоматов. Они должны осуществлять переходы одновременно. Таким образом, сначала вычисляются все условия на переходах, а затем каждый автомат осуществляет соответствующий переход.

7.1.2. Пример преобразования итеративной программы

Проиллюстрируем рассмотренные преобразования на следующем (весьма надуманном) примере (рис. 29).

```
void main() {
    inc();
    dec();
}
```

а

```
void inc() {
    d.i += 2;
    dec();
}
```

б

```
void dec() {
    d.i--;
}
```

в

Рис. 29. Преобразуемая программа. Главная процедура (а), процедура inc (б), процедура dec (в)

Им соответствуют прямые автоматы, изображенные на рис. 30.

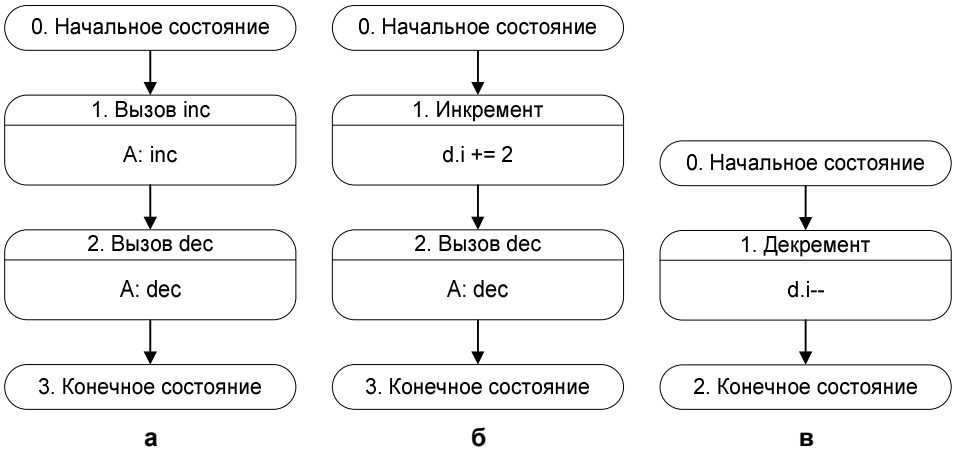


Рис. 30. Прямые автоматы для процедур main (а), inc (б) и dec (в)

Соответствующие обратные автоматы изображены на рис. 31.

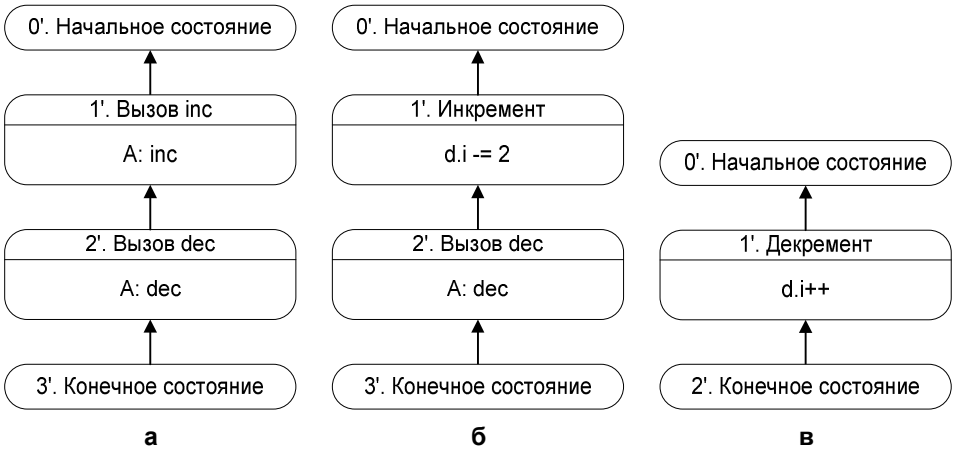


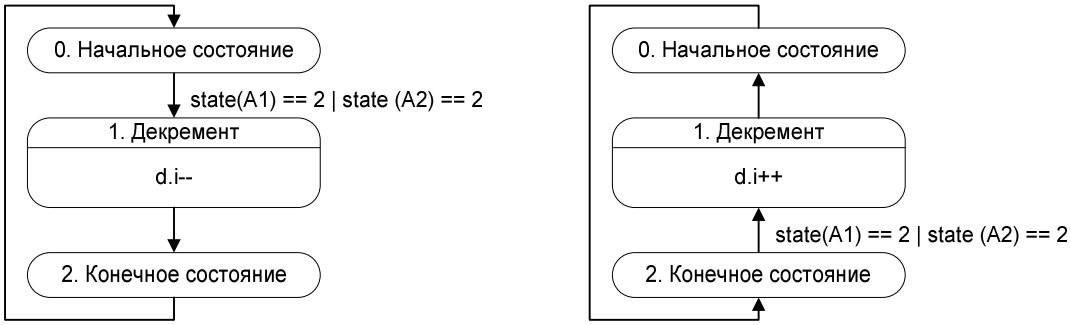
Рис. 31. Обратные автоматы для процедур main (а), inc (б) и dec (в)

Обозначим автоматы для процедур `main`, `inc` и `dec` через *A1*, *A2* и *A3* соответственно.

Рассмотрим преобразование автомата *A3*. *Список вызовов* для автомата *A3* состоит из двух пар: (*A1*, *2*) и (*A2*, *2*). При этому условие *R(A3)* записывается следующим образом:

```
state(A1) == 2 | state (A2) == 2,
```

где "`|`" — дизъюнкция. Преобразованные автоматы *A3* и *A3'* приведены на рис. 32.



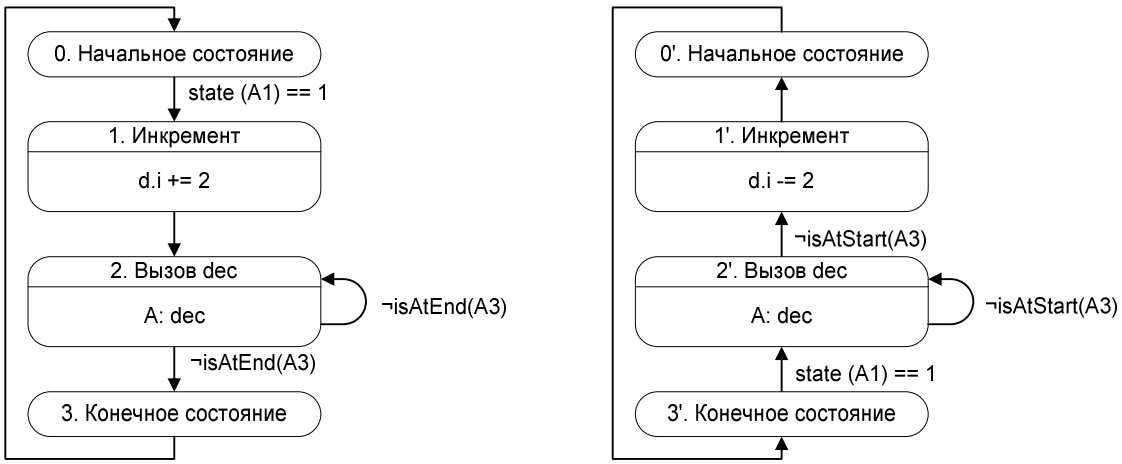
а **б**

Рис. 32. Преобразованные автоматы A3 (а) и A3’ (б)

Для автомата *A2* условие *R(A2)* имеет вид:

```
state(A1) == 1
```

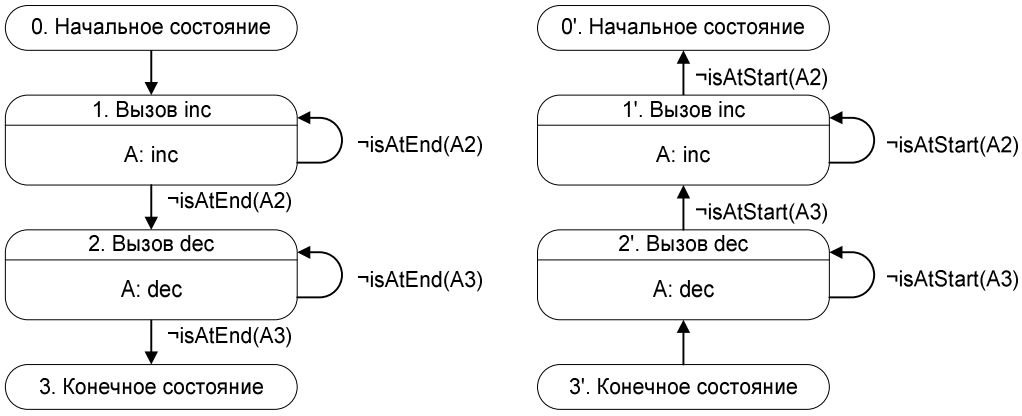
При этом в автоматах *A2* и *A2’* необходимо преобразовать состояния 2 и 2’, как показано выше. Таким образом, получаем автоматы, изображенные на рис. 33.



а **б**

Рис. 33. Преобразованные автоматы A2 (а) и A2’ (б)

Аналогично преобразуя автоматы *A1* и *A1’*, получим рис. 34.



а **б**

Рис. 34. Преобразованные автоматы A1 (а) и A1’ (б)

Система взаимодействующих конечных автоматов *A1* (*A1’*), *A2* (*A2’*) и *A3* (*A3’*) реализует исходную программу.

7.1.3. Некоторые замечания о корректности преобразования программ

Обозначим совместное состояние автоматов *A1*, *A2*, *A3* тройкой вида: *(состояние автомата A1, состояние автомата A2, состояние автомата A3)*. Тогда, при прямом проходе имеем следующую цепочку состояний:

```
(0, 0, 0) à (1, 0, 0) à (1, 1, 0) à (1, 2, 0) à
(1, 2, 1) à (1, 2, 2) à (1, 3, 0) à (2, 0, 0) à
(2, 0, 1) à (2, 0, 2) à (3, 0, 0),
```

а при обратном проходе:

```
(3, 0, 0) à (2, 3, 2) à (2, 3, 1) à (2, 3, 0) à
(1, 3, 2) à (1, 2, 2) à (1, 2, 1) à (1, 2, 0) à
(1, 1, 3) à (1, 0, 3) à (0, 3, 2).
```

Заметим, что при *обратном проходе* цепочка состояний не совпадает с перевернутой цепочкой состояний, пройденных при *прямом проходе*.

Пусть в некоторый момент времени имеем цепочку вызовов. Все процедуры, участвующие в этой цепочке, назовем э*мулируемыми* в рассматриваемый момент времени, а остальные процедуры — *не эмулируемыми*.

При *прямом проходе* автоматы, соответствующие *не эмулируемым* в данный момент процедурам, находятся в *начальном состоянии*, а при *обратном проходе* эти автоматы находятся в *конечном состоянии*. Заметим, что при выполнении шага вперед все “незадействованные” автоматы переходят в *начальное состояние*, а при выполнении шага назад они переходят в *конечное состояние*. Таким образом, как при прямом, так и при обратном проходе, после перехода в состояние, вызывающее один из “незадействованных”

автоматов, тот находится в состоянии, при котором будет осуществлена соответствующая эмуляция (либо прямого прохода, либо обратного). При рассмотрении “задействованных” автоматов такой проблемы не возникает, так как они и при прямом и при обратном проходе находятся в одном и том же состоянии.

Отдельного рассмотрения заслуживает случай, когда два вызова одной и той же процедуры идут подряд. При *прямом проходе* вызываемый автомат переходит из *конечного состояния* в *начальное* одновременно с переходом вызывающего автомата из одного состояния, осуществляющего вызов, в другое. Соответственно, когда вызывающий автомат находится в состоянии, соответствующем второму вызову, вызываемый автомат уже находится в начальном состоянии, и поэтому происходит повторная эмуляция процедуры. При *обратном проходе* одновременно с переходом вызывающего автомата выполняется переход вызываемого автомата в *конечное состояние*. Таким образом, и в этом случае преобразования выполненные на основе предлагаемого подхода корректны.

7.2. Рекурсивные программы

7.2.1. Преобразование рекурсивной программы

Вопрос о преобразовании рекурсивных программ в автоматные рассматривался в статье [11]. В данной работе предложен альтернативный метод, основанный на понятии *экземпляр автомата*.

В рекурсивной программе одна и та же процедура может встречаться в цепочке вызовов несколько раз. Назовем каждое вхождение процедуры в цепочку вызовов *экземпляром процедуры*. Проиллюстрируем это на примере функции, осуществляющей рекурсивное вычисление факториала:

```
1:      int factorial(int a) {
2:          int r = 1;
3:          if (a > 1) {
4:              r = a * factorial(a - 1);
5:          }
6:          return r;
7:      }
```

Данная функция вызывает себя в строке 4. При этом в нескольких *экземплярах процедуры* (тех, которые вызвали новый экземпляр этой процедуры) текущей будет строка 4, а в еще одном экземпляре (текущем) — некоторая другая строка.

Для отражения этого факта введем понятие *экземпляр автомата*. Как указывалось выше, пара из прямого и обратного автоматов может быть рассмотрена, как один автомат с

двумя функциями переходов. *Экземпляр автомата* — объект, хранящий состояние такого автомата.

Определим для *экземпляра автомата* следующие функции:

- `stepForward()` — сделать шаг вперед (изменить состояние и выполнить действия в соответствии с функцией переходов *прямого автомата*);
- `stepBackward()` — сделать шаг назад (изменить состояние и выполнить действия в соответствии с функцией переходов *обратного автомата*);
- `isAtStart()` — проверка, находится ли *экземпляр автомата* в *начальном состоянии*;
- `isAtEnd()` — проверка, находится ли *экземпляр автомата* в *конечном состоянии*;
- `new автомат(start)` — создать новый *экземпляр автомата*, находящийся в *начальном состоянии*;
- `new автомат(end)` — создать новый *экземпляр автомата*, находящийся в *конечном состоянии*.

Для каждого автомата можно создать несколько *экземпляров*, каждый из которых находится в своем состоянии.

При *прямом проходе* для каждого вызова процедуры создается новый *экземпляр автомата*, находящийся в *начальном состоянии*. Вызывающий автомат ожидает, пока созданный *экземпляр автомата* не закончит работу (перейдет в *конечное состояние*). Соответственно, при *обратном проходе* для каждого вызова процедуры создается новый *экземпляр автомата*, находящийся в *конечном состоянии*. Вызывающий автомат ожидает, пока созданный *экземпляр автомата* не закончит работу (перейдет в *начальное состояние*).

Фрагменты *прямого* и *обратного автоматов*, осуществляющих вызов автомата A, приведены на рис. 35.

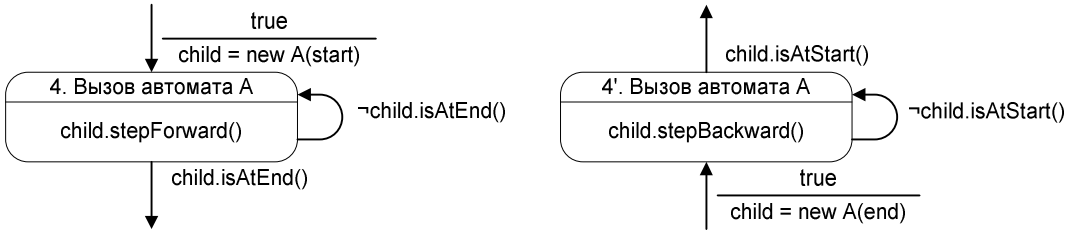


Рис. 35. Фрагменты прямого (а) и обратного (б) автоматов, вызывающие автомат A

Этими фрагментами можно заменить состояние, вызывающее автомат A в исходном автомате. Для получения автомата Мура следует использовать фрагменты, изображенные на рис. 36.

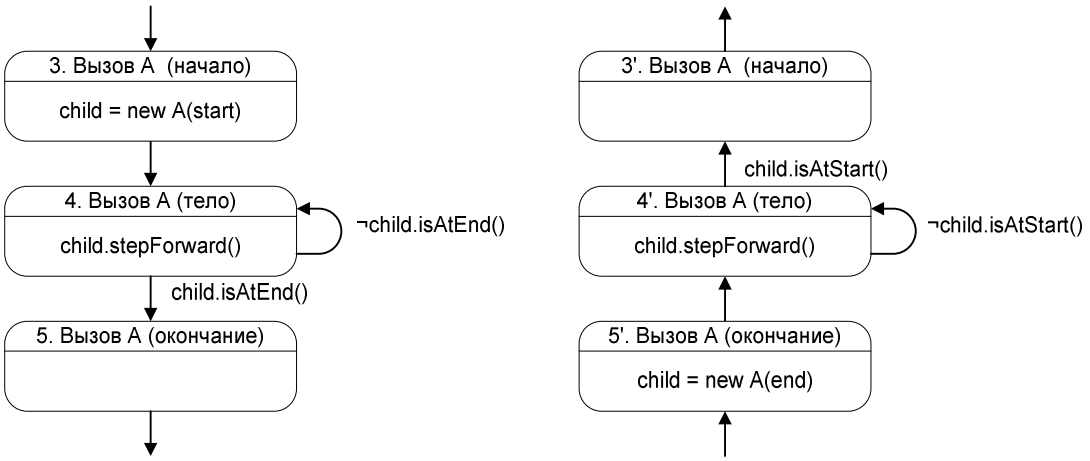


Рис. 36. Фрагменты прямого (а) и обратного (б) автоматов Мура, вызывающие автомат A

Таким образом, в этом случае к *прямому* и о*братному автоматам* необходимо добавить по два новых состояния.

7.2.2. Пример преобразования рекурсивной программы

Проиллюстрируем предложенный метод на примере функции, вычисляющей факториал (исходная программа приведена в разд. 7.2.1). Данная функция использует явную рекурсию. Преобразование программ с косвенной рекурсией производится аналогично.

В соответствии со вторым шагом предложенного метода (разд. 2) преобразуем программу к виду, использующему модель данных. Передачу параметра будем производить через переменную `d.a`, а возврат результата — через переменную `d.r`:

```
void factorial() {
    d.r = 1;
    if (d.a > 1) {
        d.a--;
        factorial();
        d.r = d.r * (d.a + 1);
    }
}
```

Прямой и обратные автоматы, построенные по данной программе, изображены на рис. 37.

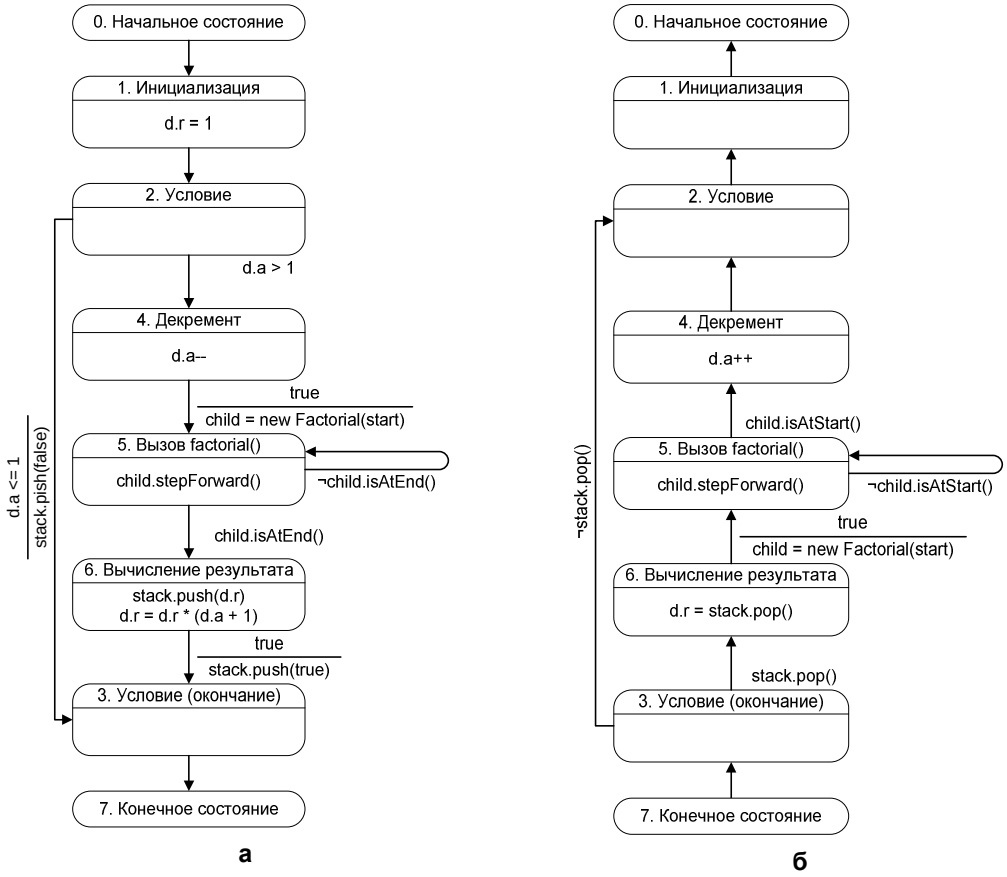


Рис. 37. Прямой (а) и обратный (б) автоматы для процедуры вычисления факториала

8. Заключение

Изложенный метод позволяет преобразовывать программы в набор конечных автоматов с возможностью их пошагового выполнения. Таким образом, предложено формальное описание логики визуализаторов на основе конечных автоматов. Предложенный метод был реализован в программном комплексе, автоматизирующем процесс написания логики визуализаторов, который будет описан в отдельной статье.

В настоящее время на основе предложенного подхода первокурсниками кафедры “Компьютерные технологии” разработано большое число визуализаторов различных алгоритмов, используемых при преподавании курса “Дискретная математика” и в уроках Интернет-школы программирования (http://ips.ifmo.ru/ru). Визуализаторы и их документация будут выложены на сайт http://is.ifmo.ru/ru в разделе “Визуализаторы”.

Работа выполнена при поддержке Российского фонда фундаментальных исследований по гранту 02-07-90114 “Разработка технологии автоматного программирования”.

Литература

1. *Кнут Д.* Искусство программирования. Том 1. Основные алгоритмы. М.: Вильямс, 2000.
2. *Кормен Т., Лейзерсон Ч., Ривес Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 1999.
3. *Казаков М.А., Столяр С.Е.* Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования //Тезисы докладов международной научно-методической конференции "Телематика-2000". СПб.: СПбГИТМО (ТУ), 2000.
4. *Корнеев Г.А., Парфенов В.Г., Столяр С.Е., Васильев В.Н.* Визуализаторы алгоритмов как основной инструмент технологии преподавания дискретной математики и программирования //Труды международной научно-методической конференции "Телематика-2001". СПб.: СПбГИТМО (ТУ), 2001.
5. *Казаков М.А., Мельничук О.П., Парфенов В.Г.* Интернет-школа программирования в СПбГИТМО. Реализация и внедрение //Труды Всероссийской научно-методической конференции "Телематика-2002". СПб.: СПбГИТМО (ТУ), 2002.
6. *Шалыто А.А., Туккель Н.И.* Преобразование итеративных алгоритмов автоматные //Программирование. 2002. № 5.
7. *Шалыто А.А., Туккель Н.И.* Программирование с явным выделением состояний //Мир ПК. 2001. № 8, № 9.
8. *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.
9. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982.
10. *Грис Д.* Наука программирования. М.: Мир, 1984.
11. *Туккель Н.И., Шалыто А.А., Шамгунов Н.Н.* Реализация рекурсивных алгоритмов на основе автоматного подхода //Телекоммуникации и информатизация образования. 2002. № 5.