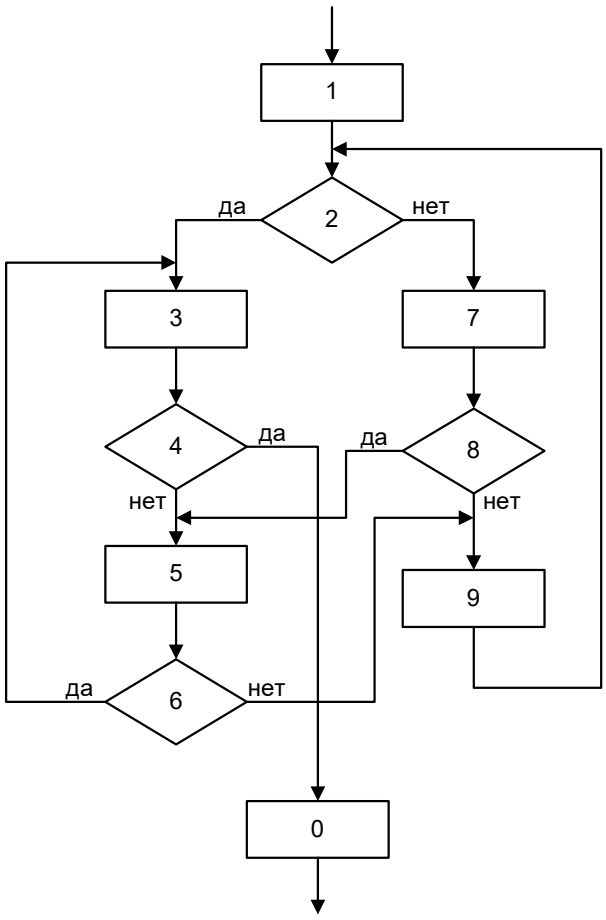
ТУБ №38:
Метод введения переменной состояния

Наверное, самым широко используемым методом преобразования неструктурированных программ в структурированные является метод введения переменной состояния. Более того, сама идея, положенная в основу метода, настолько универсальна, что позволяет даже самую сложную логику превратить в простой алгоритм, который, к тому же, будет структурированным. Разумеется, за это придётся чем-то заплатить, но давайте по порядку.

Рассмотрим такую схему алгоритма:



Невооружённым глазом видно, что о структурированности такого алгоритма речи идти не может: ни одной из базовых конструкций, предложенных Бомом и Джакопини, здесь найти не удастся. При этом метод дублирования кодов навести порядок не поможет, т.к. здесь явно присутствуют циклы, с которыми этот метод не работает.

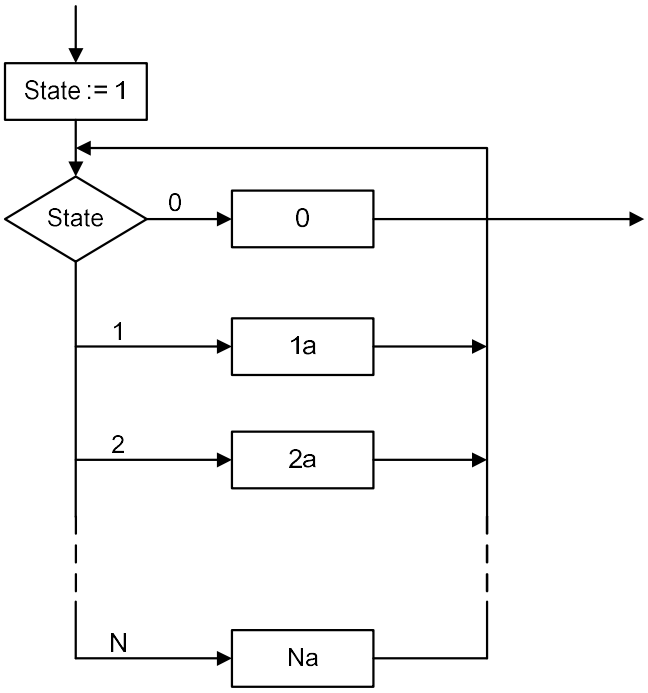
Безвыходная ситуация? Ни в коем случае!

Решение проблемы заключается в использовании *метода введения переменной состояния*. В соответствии с ним для преобразования алгоритма в структурированный нужно выполнить следующие действия:

- пронумеровать блоки на схеме (обычно последний блок, которым заканчивается алгоритм, получает номер 0);
- ввести дополнительную переменную, которая называется *переменной состояния*;

– перестроить схему алгоритма к виду, предложенному авторами метода — Ашкрофтом и Манной.

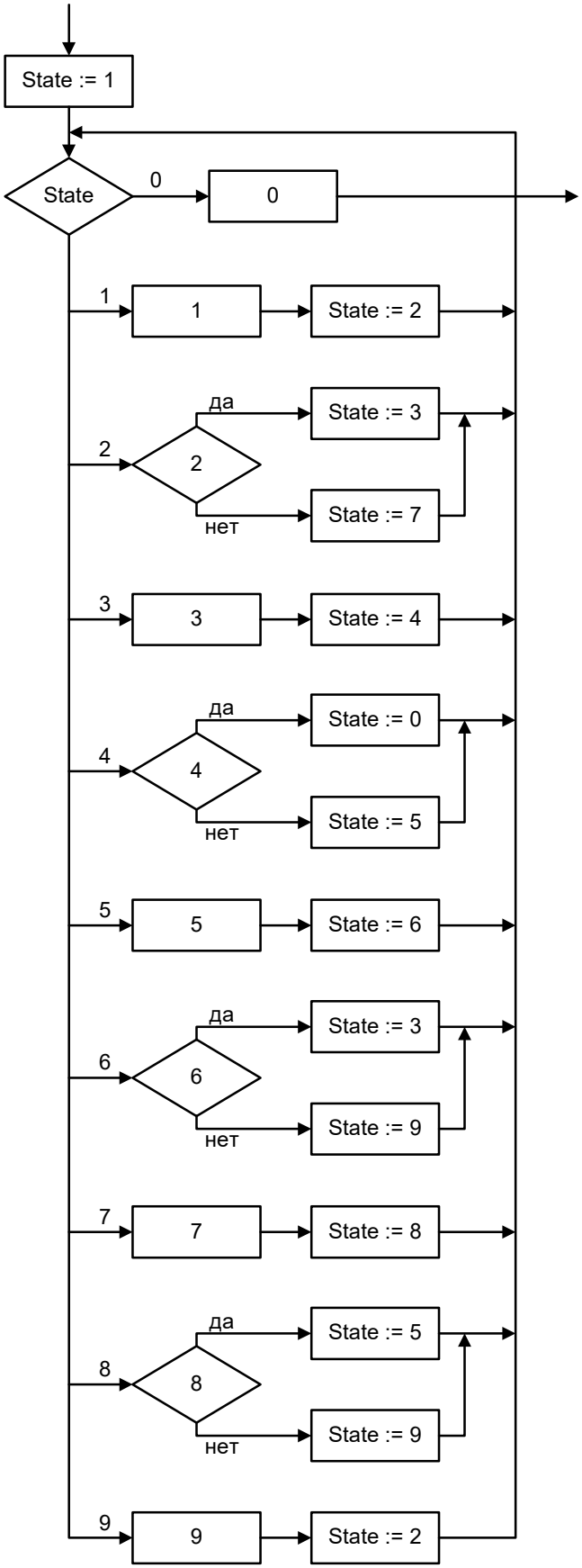
Для перестроения схемы необходимо каждый блок исходной схемы дополнить блоками, изменяющими значение переменной состояния (назовём её `State`), а затем включить их в следующую схему:



Блоки 1a, 2a, … Na представляют собой блоки исходной схемы, дополненные изменениями значения переменной состояния `State`.

Идея метода введения переменной состояния заключается в том, чтобы построить алгоритм, который отражает пошаговое выполнение исходного алгоритма. При этом каждой итерации цикла соответствует один шаг (блок) исходной схемы, а переменная `State` в начале каждой итерации должна содержать номер блока исходной схемы, который должен быть выполнен на соответствующем шаге.

Для исходной схемы полный результат преобразования будет иметь следующий вид:



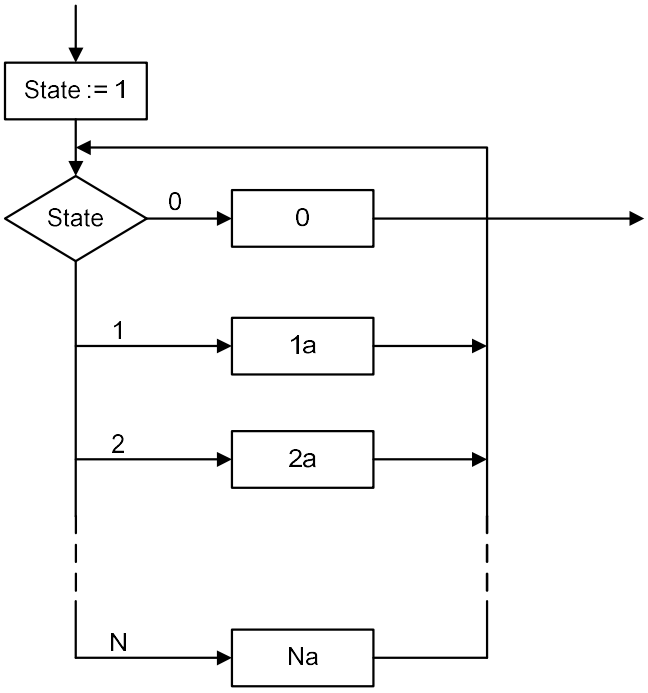
Принцип построения такой схемы достаточно прост:

– к каждому блоку «Процесс» исходной схемы (кроме последнего) добавляется блок, изменяющий значение переменной состояния на номер следующего блока в исходной схеме (например, в исходной схеме после блока 1 управление передаётся блоку 2 — значит, блок 1 на преобразованной схеме будет дополнен присваиванием значения 2 переменной State);
– к каждой ветви блока «Решение» исходной схемы добавляется по одному блоку, изменяющему значение переменной состояния на номер блока, который должен получать управление при выборе этой ветви в исходной схеме (например, в исходной схеме после блока 2 управление передаётся блоку 3 в ветви «да» и блоку 7 в ветви «нет» — это и отражается в преобразованной схеме); при этом после изменения значения переменной состояния ветви сводятся вместе.

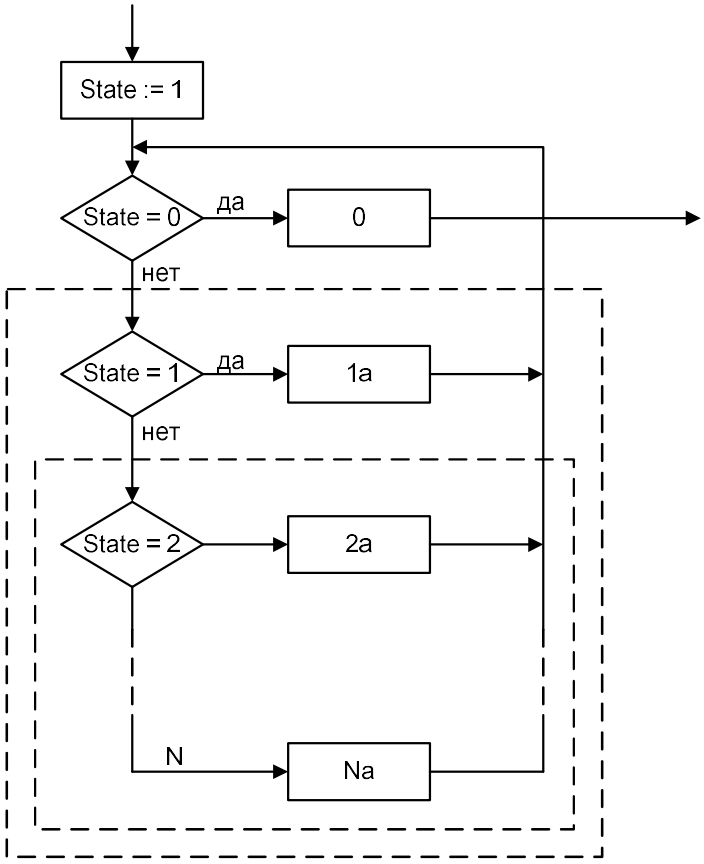
Очевидно, что получившаяся схема оказалась более громоздкой, чем исходная. Но стал ли алгоритм структурированным? Проверим это.

Нетрудно заметить, что фрагменты схемы, которыми заменяются исходные блоки, представляют собой либо конструкции следования (если заменяется блок «Процесс»), либо конструкции принятия двоичного решения (если заменяется блок «Решение»), что позволяет рассматривать их как функциональные блоки и возвращает нас к схеме в общем виде.

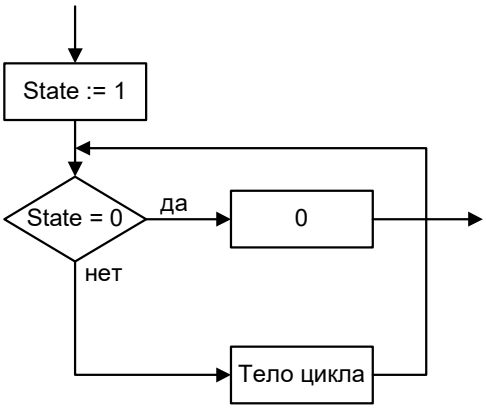
В теле цикла на этой схеме содержится единственная конструкция — конструкция множественного выбора, соответствующая оператору `case`:



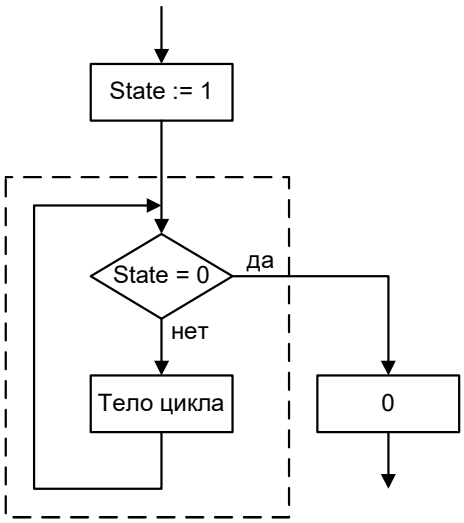
Эта конструкция может быть преобразована в набор конструкций принятия двоичного решения, после чего, двигаясь по схеме снизу вверх, можно показать, что они являются вложенными и последовательно заменить их функциональными блоками:



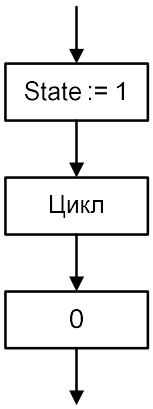
Результатом такого преобразования станет следующая схема:



Для наглядности немного переместим блоки, не меняя связей между ними:



Заменив конструкцию обобщённого цикла на функциональный блок, получим конструкцию следования из трёх блоков:



Таким образом, вся схема, получившаяся в результате преобразования по методу введения переменной состояния, может быть приведена преобразованиями Бома-Джакопини к единственному функциональному блоку, а значит, результат преобразования является структурированным алгоритмом.

Безусловным преимуществом метода введения переменной состояния является его применимость к абсолютно любым схемам. Особенно ценно то, что его применение может выполняться автоматически, т.к. процесс дополнения блоков исходной схемы и их подстановки в схему результирующую не содержит никаких творческих этапов: всё выполняется строго по заданным правилам.

В качестве недостатка можно назвать значительное изменение топологии (т.е. структуры) схемы алгоритма. Другими словами, схема «изменяется до неузнаваемости».

Другим слабым местом метода является громоздкость получающегося алгоритма и необходимость тратить время на проверку и изменение значения переменной состояния: оба этих факта могут негативно сказываться на быстродействии программы.

Подход, лежащий в основе метода, можно обобщить и вместо одной переменной состояния использовать целый набор переменных. При этом становится возможным использовать его не только для преобразования неструктурированного алгоритма в структурированный, но и для решения ряда сложных задач:

– разработка эмуляторов различных процессоров и контроллеров;
– поддержка асинхронных процедур/методов в некоторых современных языках программирования;
– разработка некоторых модулей компиляторов и интерпретаторов (лексический и синтаксический анализ).

Кроме того, сама идея построения алгоритма решения задачи вокруг явно выделенного состояния программы зачастую позволяет существенно упростить программу.

Предположим, перед нами поставлена задача: разработать обучающую программу для детей, рассказывающую о правилах дорожного движения. Один из элементов такой программы — светофор. Как организовать переключение его сигналов и их отображение? Как вообще хранить информацию о том, что в данный момент должен показывать светофор?

Простое и очевидное решение «в лоб» заключается в том, чтобы иметь для каждой секции светофора по одной переменной логического типа, которая будет означать, горит ли соответствующая секция:

```
var
  Red, Yellow, Green: Boolean;
```

Выглядит неплохой идеей, не так ли? Но у этого решения есть неприятный подвох. В какой-то момент нам понадобится показывать, как сменяются сигналы светофора, и вот тут-то окажется, что логика переключения, казавшаяся такой простой, на самом деле весьма запутанная:

```
if Red and not Yellow and not Green then
begin
  Yellow := True;
end
else if Red and Yellow and not Green then
begin
  Red := False;
  Yellow := False;
  Green := True;
end
else if not Red and not Yellow and Green then
begin
  Yellow := True;
  Green := False;
end
else if not Red and Yellow and not Green then
begin
  Red := True;
  Yellow := False;
end;
```

Многовато, не правда ли? И так много способов ошибиться и не заметить! А если где-то по ошибке эти переменные получат значения, которые не соответствуют ни одной из ветвей, переключение вообще не сможет быть выполнено.

Неужели нельзя проще? Конечно, можно!

Давайте пронумеруем возможные комбинации сигналов. 0 — «красный», 1 — «красный с жёлтым», 2 — «зелёный», 3 — «жёлтый». Как теперь будет выглядеть логика переключения сигналов?

Заметим, что в Pascal/Delphi для этих целей вместо чисел можно использовать перечислимый тип:

```
type
  TState = (stRed, stRedYellow, stGreen, stYellow);

var
  State: TState;
```

Теперь, если в переменной `State` содержится описание какого-либо из возможных состояний светофора, для получения следующего достаточно сделать так:

```
case State of
  stRed:
    State := stRedYellow;
  stRedYellow:
    State := stGreen;
  stGreen:
    State := stYellow;
  stYellow:
    State := stRed;
end;
```

или даже ещё проще:

```
if State <> stYellow then
  Inc(State)
else
  State := stRed;
```

Свойство перенумерованности перечислимого типа позволяет применить к переменной `State` процедуру `Inc`. Этот же код можно переписать чуть хитрее:

```
if State <> High(TState) then
  Inc(State)
else
  State := Low(TState);
```

Теперь, если появятся новые возможные состояния или поменяются местами существующие, код останется работоспособным.

Разумеется, реализация с процедурой `Inc` подойдёт только в том случае, если за одним из состояний всегда может следовать только одно из оставшихся. В более сложных случаях придётся возвращаться к реализации с оператором `case`, однако и она намного проще и нагляднее, чем набор переменных логического типа.

Но как теперь будет происходить отображение светофора? Ненамного сложнее:

```
case State of
  stRed:
    // Рисуем светофор в состоянии «Красный»
  stRedYellow:
    // Рисуем светофор в состоянии «Красный с жёлтым»
  ...
end;
```

Как видим, идея, лежащая в основе метода введения переменной состояния может быть полезна не только для исправления проблем в уже написанных программах, но и для написания новых программ сразу «начисто».

Дополнительные вопросы

1. Дана строка, содержащая фрагмент программы на языке Pascal. Удалить из этой строки все многострочные комментарии (по правилам языка они заключаются в фигурные скобки). Учесть, что в тексте программы могут встречаться строковые литералы:

```
WriteLn('Такой {фрагмент текста в фигурных скобках} — не комментарий!');
```

2. Написать программу, которая для введённой строки из открывающих и закрывающих круглых скобок проверяет правильность их расстановки.

Примеры правильных расстановок:

```
()
((())(()))
()()(())
```

Примеры неправильных расстановок:

```
(
()(
)
())(()
```