Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А.Н. Котов, А.А. Шалыто

Сравнение различных вариантов реализации на примере задачи о декодировании файлов формата *GIF*

Программирование с явным выделением состояний

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2007

Содержание

1. Введение ... 3
2. Описание формата *GIF* ... 4
  2.1. Общие сведения ... 4
  2.2. Структура формата *GIF* ... 5
    2.2.1. Структура файла ... 5
    2.2.2. Кодирование цепочек с использованием *LZW*-кодов ... 6
  2.3. Пример файла формата *GIF* ... 7
3. Описание интерфейса программы ... 8
4. Классическая реализация ... 10
5. Автоматная реализация ... 11
  5.1. Представление программы в виде трех автоматов ... 11
  5.2. Описание автомата *A1* ... 12
  5.3. Описание автомата *A2* ... 14
  5.4. Описание автомата *A3* ... 16
  5.5. Реализация автоматов ... 18
  5.6. Различное взаимодействие автоматов ... 19
Заключение ... 21
Список источников ... 22
Приложение 1. Исходный код программы с классической реализацией ... 23
Приложение 2. Исходный код программы с автоматной реализацией ... 25

Потоковое программирование нужно тем, кто не умеет
использовать конечные автоматы
Алан Кокс, ведущий разработчик *Linux*-ядра в Великобритании

1. Введение

Графические файлы формата *GIF* [1] получили широкое распространение. Однако свободное использование инструментов для работы с этим форматом при разработке оригинального прикладного программного обеспечения до последнего времени было затруднено из-за ряда патентных ограничений [2]. В частности, свободная библиотека *GD* (www.boutell.com/libgd/) не включала поддержку этого формата вплоть до последнего времени. Поэтому представляется целесообразным написание независимой от используемых платформ и библиотек программы, которая преобразует поток битов формата *GIF* в двумерную матрицу изображения.

В данной работе представлены две различные реализации такой программы на языке *Perl* (www.perl.org). Этот современный кроссплатформенный язык программирования высокого уровня удобен для написания подобных программ. Первая реализация представляет собой классический вычислительный алгоритм декодирования [3], который закодирован на требуемом языке программирования традиционным образом. Вторая реализация сделана с применением автоматного подхода [4].

В автоматной реализации программа декодирования файлов формата *GIF* построена на основе трех конечных автоматов. Первый (главный) автомат преобразует бинарный поток битов из файла в соответствии со структурой формата *GIF*. Он имеет пять состояний. Второй автомат работает параллельно с первым и отвечает за преобразование входных *LZW*-кодов в выходные цепочки изображения. Он вызывается из первого автомата, как только тот распознает очередной *LZW*-код в потоке битов. Второй автомат имеет семь состояний. Третий автомат является вложенным в первый, и имеет четыре состояния. Он вызывается только один раз и отвечает за чтение *GIF*-заголовков.

Наибольший интерес представляет исследование различных способов взаимодействия автоматов друг с другом, а также сравнение соответствующих реализаций этой программы. В работе рассмотрены три способа взаимодействия автоматов, два из которых **естественным образом вводят параллелизм в реализуемую программу, которую традиционным путем распараллелить весьма трудно**. Полученный результат является особенно ценным при использовании современных многоядерных процессоров.

2. Описание формата *GIF*

2.1. Общие сведения

Формат *GIF* был разработан компанией *CompuServe* в 80-х годах прошлого века для хранения в сжатом двоичном виде цифровых изображений [1]. Формат *GIF* позволяет хранить изображения с глубиной цвета не более восьми бит на пиксель – не более чем 256-цветное изображение. В настоящее время этот формат используется в основном для представления графики в *WEB* с большим количеством однотонных элементов (линии, кнопки, элементы оформления и т.д.) и не приспособлен для хранения фотографических изображений с большим количеством деталей. С момента своего появления формат *GIF* входит в число самых распространенных графических форматов, наряду с *JPEG*, *PNG*, *TIFF* и *BMP*. Несмотря на то, что популярность рассматриваемого формата падает с каждым годом в пользу более нового, обеспечивающего лучшее сжатие, универсального, открытого и поддерживаемого всеми разработчиками формата *PNG*, игнорировать поддержку *GIF* в программах, работающих с изображениями, невозможно.

Некоторые общие характеристики этого формата:

- размер изображения - от 1x1 до 65535x65535 пикселей;
- количество цветов в палитре - от 2 до 256;
- каждый цвет палитры имеет 24-хбитную глубину (выбор из 16 миллионов цветов).

2.2. Структура формата GIF

2.2.1. Структура файла

Файлы формата *GIF* имеют блочную структуру. Минимально необходимый набор блоков для простейшего не анимированного файла формата *GIF* приведен на рис.1.

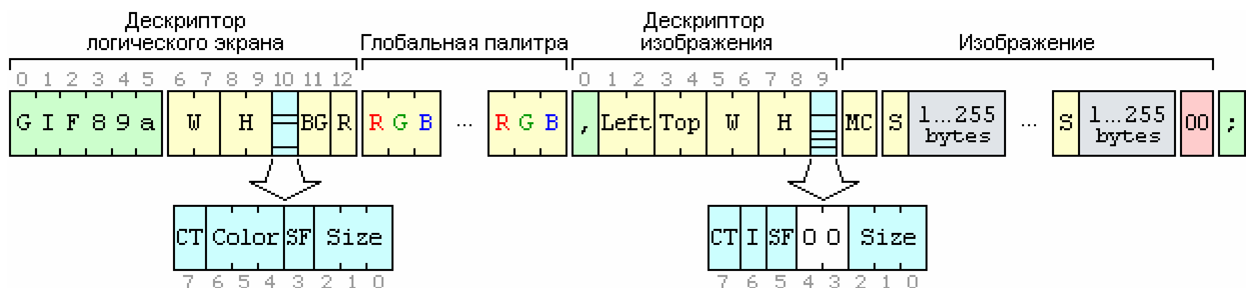


Рис. 1. Набор блоков файла формата *GIF*

Зеленым цветом обозначены текстовые константы;

желтым – переменные;

голубым – байты с упакованными в них полями и флагами;

розовым – шестнадцатеричные константы;

серым – блоки данных переменной длины,

белым – зарезервированные байты или биты.

Следующие четыре блока являются основными для файлов формата *GIF*:

- дескриптор логического экрана – описывает параметры логического устройства отображения графики;
- глобальная палитра – сопоставляет каждому номеру свой конкретный цвет;
- дескриптор изображения – описывает параметры закодированного изображения, которые, как правило, совпадают с параметрами логического экрана;
- изображение – цепочка с номерами цветов, соответствующая последовательности пикселей на экране, закодированная с применением алгоритма *LZW*.

Дескриптор логического экрана, глобальная палитра и дескриптор изображения, в совокупности образуют заголовок изображения в файле формата *GIF*, а блок изображения – само закодированное изображение.

2.2.2. Кодирование цепочек с использованием *LZW*-кодов

В исходном виде изображение представляет собой матрицу размером W x H, состоящую из номеров цветов. Каждый номер цвета кодируется с использованием $size + 1$ бита. Таким образом, можно закодировать изображение с максимальным количеством цветов, равным $2^{size + 1}$. Поскольку размер поля size в заголовке изображения формата *GIF* составляет ровно три бита, то максимальное количество цветов не может превышать числа $256 = 2^{7 + 1}$.

Цепочка длиной W x H из номеров цветов кодируется с применением алгоритма сжатия *LZW*, который подробно описан в работе [3]. В алгоритме используются коды переменной длины. Коды со значениями от 0 до $2^{size + 1} - 1$ соответствуют номерам цветов, код со значением $2^{size + 1}$ является кодом очистки, код со значением $2^{size + 1} + 1$ соответствует окончанию изображения, а все последующие коды предназначены для кодирования цепочек из номеров цветов в соответствии с алгоритмом *LZW*. Первоначально первым свободным кодом является код со значением $2^{size + 1} + 2$. При этом каждый из кодов занимает в сжатом виде MC + 1 бит. Когда следующий свободный код перестает умещаться в этот размер (его значение должно быть равно $2^{MC + 2}$), текущее значение размера кода автоматически увеличивается на один бит. Это обеспечивает реализацию использования кодов переменной длины. Код очистки указывает на то, что таблица *LZW*-кодов должна быть сброшена, и первым свободным кодом должен снова стать код размером MC + 1 бит со значением $2^{size + 1} + 2$.

Закодированная цепочка из номеров цветов упаковывается в поток байтов. Этот поток разбивается на блоки, содержащие до 255-ти байт закодированной цепочки. В начале каждого блока присутствует байт-счетчик, в котором хранится число содержащихся в нем байтов из потока. Такие блоки из байта-счетчика и потока данных и образуют сжатое изображение в файлах формата *GIF*.

2.3. Пример файла формата GIF

В качестве примера файла формата *GIF* в работе используется изображение `_test.gif`. В графическом (декодированном) виде оно представлено на рис. 2.

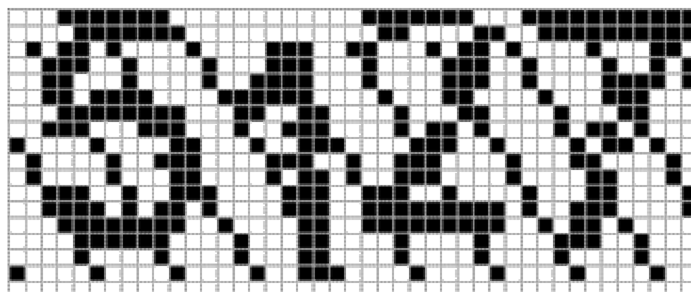


Рис. 2. Пример файла формата *GIF* в графическом виде

В исходном виде (в качестве потока битов, который необходимо декодировать) это изображение представлено на рис. 3.

```
0000000000:  47 49 46 38 39 61 2B 00 | 12 00 F0 00 00 FF FF FF   GIF89a• ↕ E
0000000010:  00 00 00 2C 00 00 00 00 | 2B 00 12 00 00 02 7A 84      ,    + ↕  ☻zД
0000000020:  83 19 C6 CD 9A 1E 84 6D | 2E 87 51 BD 36 2E 7D 1D   Г↓╞═Ъ▲Дm.3Q╜6.}↔
0000000030:  1B 17 7E 5F 25 49 1C 7A | 68 A2 FA BE 27 E0 D2 AC   ←↨~_%I∟zhв·╛'рπм
0000000040:  F2 8C 31 ED B5 7C FD 8B | E4 7C C3 5F C6 92 E2 B1   ЄМ1э╡|¤Лф|├_╞Тт▒
0000000050:  94 45 A0 B1 55 82 A2 40 | D3 99 AD C8 BC 0A A5 57   ФEа▒UВв@╙Щн╚╝◙еW
0000000060:  11 75 47 0A 3D C3 4B E0 | AA 29 36 69 CD 1B F4 79   ◄uG◙=├Kрк)6i═←Їy
0000000070:  3B 76 A5 93 B2 DE 97 DB | 99 C1 4B C8 2E 33 EF 65   ;veУ▓▐Ч█Щ┴K╚.3яe
0000000080:  E5 30 51 36 87 25 24 76 | 58 B8 E4 67 D8 C8 48 A6   х0Q63%$vX╕фg╪╚Hж
0000000090:  28 89 41 59 69 79 C9 50 | 00 00 3B                  (ЙAYiy╔P  ;
```

Рис. 3. Пример файла формата *GIF* в виде потока битов

В данном примере байты со смещениями `00-1D` описывают заголовок изображения, а байты со смещениями `1E-9A` – само изображение. Байт со смещением `1E` соответствует счетчику байтов в одном блоке *LZW*-кодов. Его значение определяет разницу между конечным и начальным смещениями байтов в конкретном блоке (`7A = 99 – 1F`). Следующий за этим блоком байт-счетчик со смещением `9A` равен нулю. Поэтому других блоков в демонстрационном изображении нет, а байт со смещением `9B` (`';'`) является признаком конца *GIF*-файла.

3. Описание интерфейса программы

В приложениях 1, 2 представлены исходные коды для классической и одной из автоматных реализаций программы, которая декодирует графические файлы формата *GIF*. В этих программах процесс запуска, входные аргументы и интерфейсы идентичны. Поэтому для демонстрации выполнения декодирования может быть использована любая из них. Опишем интерфейс программы.

Программа представляет собой скрипт на языке программирования *Perl*. На входе она получает файл формата `.gif`, а на выходе возвращает результат работы в виде текстового файла с декодированным изображением.

Для запуска программы необходим интерпретатор языка *Perl*. В операционных системах на основе ОС *UNIX* (*Linux*, *FreeBSD*) интерпретатор языка *Perl* обычно установлен по умолчанию. Для операционной системы *Windows* можно установить интерпретатор *ActivePerl* – компании *ActiveState* (http://www.activestate.com/Products/ActivePerl/?mp=1).

Формат запуска программы из командной строки:

```
<perl> gif_parser.pl <input.gif> <output.txt>
```

Под ОС *Windows* программа может запускаться так:

```
C:\perl\bin\perl gif_parser.pl file.gif file.txt
```

Под ОС *Unix* – примерно так:

```
./gif_parser.pl file.gif file.txt
```

Здесь `file.gif` – файл с графическим изображением формата *GIF*, а `file.txt` – файл, создаваемый программой, в который она будет записывать результаты своей работы.

При запуске программы на демонстрационном файле `_test.gif` (разд. 2.3), результатом ее работы является текстовый файл следующего содержания:

```
43 x 18 x 2
0001111111000000000000111111100011111111111
0000111111000000000000011000011001111111111
0101101000010000111001100010110100001000110
0011100100001000111000100001110010000100101
0011000100001001111000100001110010000111101
0011011110000111111000010001100001000111000
0001111111100011001100001000110000100011000
0011100011100010011100001011010000101101000
1000010000110001001110000111001000010110100
0100001001110000111001001110000100011000010
0100001000110000101101001110000100001100010
0011100100111000011100111001000010001100001
0011110101101000011100111111111010011100001
0001111111100100001100111111111001011010000
0000111111000010001100001000010000111001000
0000100001000010001100001000010000100001000
1000010000100001001110000100001000010000100
0000000000000000000000000000000000000000000
```

В первой строке этого файла указаны размеры изображения (43x18) и количество цветов (2). В следующих строках представлено само декодированное изображение. В данном примере цифра “1” соответствует пикселям черного цвета, а цифра “0” – пикселям белого цвета.

4. Классическая реализация

В приложении 1 приведен исходный код для классической реализации демонстрационной программы, которая декодирует файлы формата *GIF* без использования автоматов.

Текст программы состоит из двух частей: вызывающей (интерфейсной) части и процедуры декодирования `parse_gif_classic($image)`.

Вызывающая часть – это все строки программы, начиная с первой, и заканчивающиеся терминатором `exit 0`. Вызывающая часть имеет следующую функциональность.

1. Открывает исходный демонстрационный файл `_test.gif` для чтения и копирует его содержимое в строку – последовательность байтов `$image`.
2. Вызывает процедуру декодирования последовательности байтов `$image`. В классической реализации этой процедурой является процедура `parse_gif_classic($image)`. В описываемой ниже автоматной реализации это процедура `parse_gif_automata($image)`. Если рассматривать эти процедуры в качестве черных ящиков, то они идентичны: их интерфейсы, входные параметры и результаты работы полностью совпадают. Две процедуры отличаются друг от друга лишь внутренней реализацией механизма декодирования `.gif`-строки.
3. Записывает результат работы декодирующей процедуры в выходной файл `_test.txt`.

Процедура декодирования `parse_gif_classic($image)` представляет собой *Perl*-код, соответствующий описанию этого процесса [5]. Процедура имеет следующую функциональность:

1. Инициализирует различные заголовки изображения, работая с соответствующими подстроками строки `$image`.
2. Создает, путем последовательного чтения байтов-счетчиков из блока изображения, единую строку `$rString` с цепочкой *LZW*-кодов, которые описывают это изображение.
3. При помощи стандартного вычислительного алгоритма декодирования *LZW*, реализованного на языке *Perl*, преобразует (строки, начиная с '`# LZW decode data`') цепочку *LZW*-кодов `$rString` в последовательность номеров цветов (массив `@c`).

5. Автоматная реализация

В приложении 2 приведен исходный код для демонстрационной программы, которая декодирует файлы формата *GIF* с применением конечных автоматов.

Текст этой программы состоит из вызывающей части (описана в разделе 4), декодирующей процедуры `parse_gif($image)`, реализованной в виде конечного автомата, а также процедур `process_lzw()` и `get_config($image)`, которые реализуют еще два конечных автомата. Программа содержит также ряд вспомогательных вычислительных процедур.

5.1. Представление программы в виде трех автоматов

Для декодирования используются три конечных автомата. Главный автомат *A1* преобразует бинарный поток битов из входного файла в поток *LZW*-кодов. Автомат *A2* вызывается из автомата *A1* по событию `e8`, которое «оповещает», что на выходе автомата *A1* появился новый *LZW*-код. Автомат *A2* расшифровывает этот код и добавляет расшифровку в цепочку номеров цветов декодированного изображения. Автомат *A3* является вложенным в автомат *A1*. Он вызывается только один раз и отвечает за чтение заголовка *GIF*-файла. Модель декодирующей процедуры, реализованной в виде трех автоматов, представлена на рис. 4.

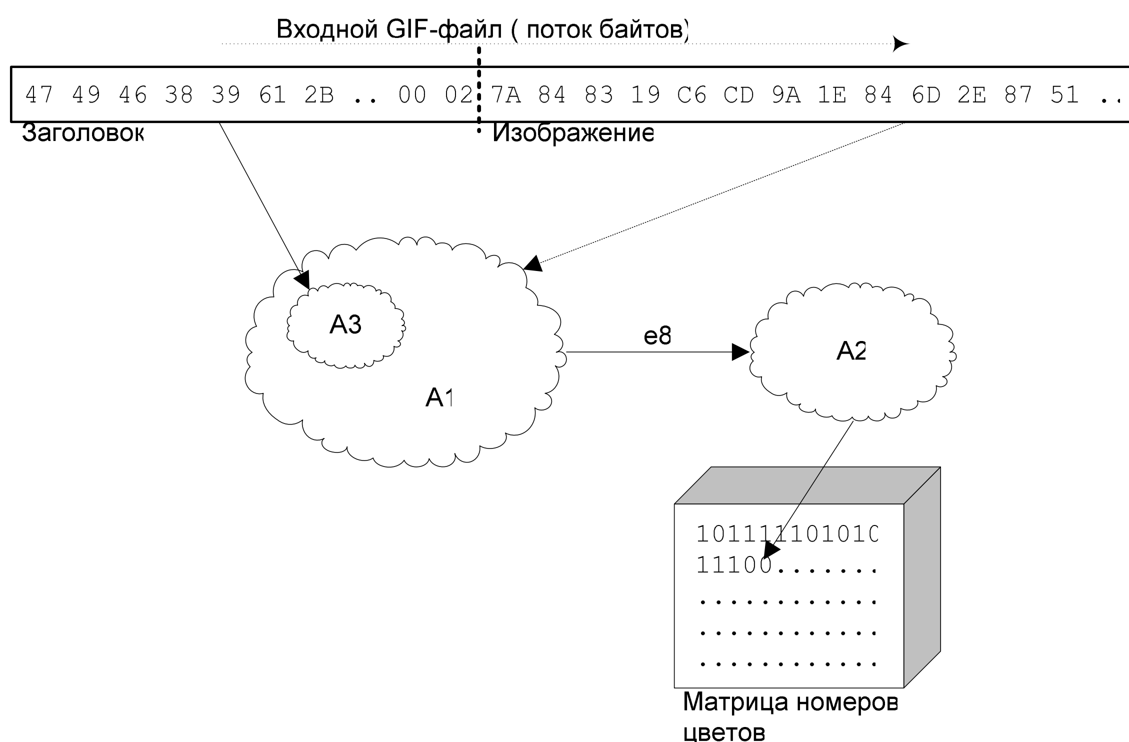


Рис. 4. Автоматная модель декодирующей процедуры

5.2. Описание автомата A1

Автомат *A1* – главный автомат процедуры декодирования. Он имеет пять состояний. При вызове процедуры декодирования автомат *A1* сразу оказывается в начальном состоянии 1.

Состояние «1. Инициализация». В этом состоянии происходит вызов вложенного автомата *A3*, который считывает заголовок *GIF*-файла и возвращает автомату *A1* структуру данных, соответствующую этому заголовку. Два выходных воздействия (`z2` и `e3`) завершают инициализацию автомата *A1* и генерируют событие, инициализирующее автомат *A2*, который может таким образом быть запущен в параллельном потоке. После завершения воздействий `z2` и `e3`, автомат *A1* переходит из состояния 1 в состояние 2.

Состояние «2. Получение счетчика». Выходные воздействия `z4` и `z5` считывают из файла очередной байт и инициализируют его значением переменную-счетчик `$counter` байтов в блоке изображения. Логическая переменная `x1` принимает истинное значение, если в блоке изображения еще присутствуют данные (`$counter > 0`). В этом случае автомат *A1* переходит из состояния 2 в состояние 3. Если данных больше нет (`$counter == 0`), то логическая переменная `x1` принимает ложное значение, и автомат *A1* переходит из состояния 2 в терминальное состояние 0.

Состояние «3. Продолжение чтения». Выходные воздействия `z4` и `z6` в состоянии 3 считывают из файла очередной байт и добавляют его к текущему значению буфера данных. Логическая переменная `x3` проверяет длину этого буфера. Если уже прочитано данных не меньше, чем требуется для печати очередного *LZW*-кода (`$data->{bit} >= $data->{digits}`), то переменная `x3` принимает истинное значение и автомат *A1* должен перейти в состояние 4. Если условие `x3` не выполнено (длина буфера прочитанных данных все еще мала), то автомат *A1* должен остаться в состоянии 3, для того, чтобы снова выполнить воздействия `z4` и `z6` и пересчитать значения логических переменных.

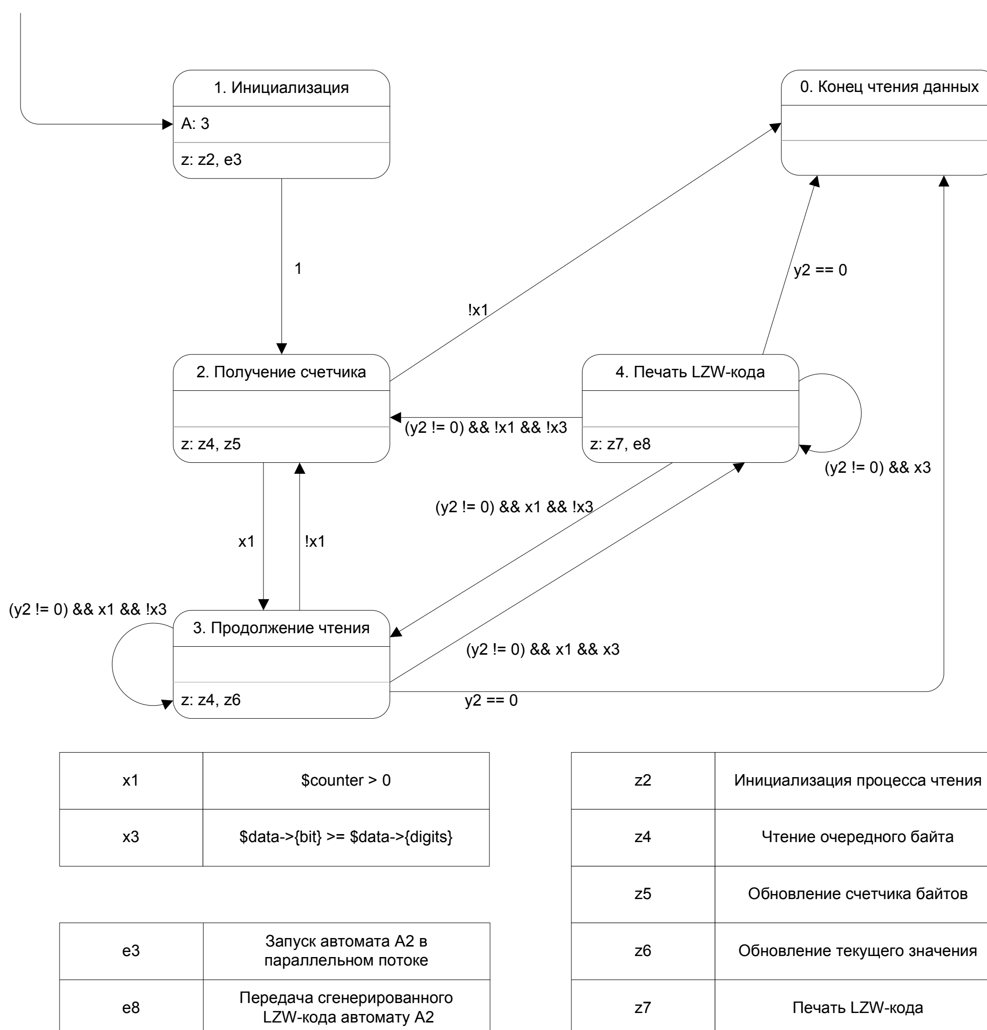
Состояние «4. Печать *LZW*-кода». Выходные воздействия `z7` и `e8` получают из буфера прочитанных данных очередной *LZW*-код, подают его на вход автомату *A2* и генерируют событие, сообщающее этому автомату о том, что на его входе появилась новая информация. Если после этого логическая переменная `x3` все еще принимает истинное значение, то это означает, что в буфере прочитанных данных содержится еще как минимум один *LZW*-код, ожидающий печати. Поэтому автомат *A1* должен остаться в

состоянии 4. Если же в буфере стало снова недостаточно данных (переменная `x3` принимает отрицательное значение), то автомату *A1* необходимо прочитать еще один байт данных. Для этого он должен вернуться либо в состояние 3, если значение счетчика байтов в блоке еще не обнулилось (`x1 = true`), либо в состояние 1, если следующим байтом в файле должен быть байт-счетчик (`x1 = false`).

Кроме того, когда автомат *A1* находится в состоянии 3 или в состоянии 4, он проверяет, не завершил ли свою работу по печати декодированного изображения автомат *A2*. Если автомат *A2* находится в терминальном состоянии (`y2 = 0`), то продолжать чтение оставшихся данных из *GIF*-файла бессмысленно, и автомат *A1* сам переходит в терминальное состояние 0.

Состояние «0. Конец чтения данных». Терминальное состояние 0 не имеет никаких выходных воздействий. В этом состоянии автомат *A1* завершает свою работу и возвращает управление вызывающей части.

На рис.5 приведена диаграмма состояний автомата *A1*.



x1	$counter > 0
x3	$data->{bit} >= $data->{digits}

e3	Запуск автомата A2 в параллельном потоке
e8	Передача сгенерированного LZW-кода автомату A2

z2	Инициализация процесса чтения
z4	Чтение очередного байта
z5	Обновление счетчика байтов
z6	Обновление текущего значения
z7	Печать LZW-кода

Рис. 5. Диаграмма состояний автомата *A1*

5.3. Описание автомата A2

Автомат *A2* реализует вычислительный алгоритм преобразования потока *LZW*-кодов в цепочку номеров цветов выходного изображения. Он имеет семь состояний. Когда автомат *A1* генерирует событие *e3*, автомат *A2* получает сигнал на запуск своей работы и переходит в начальное состояние 1.

Состояние «1. Ожидание *LZW*-кода». В этом состоянии автомат *A2* не совершает никаких выходных воздействий. Его задача – дождаться получения на вход нового *LZW*-кода (события *e8*) и перейти в состояние 2 для его обработки.

Состояние «2. Обработка *LZW*-кода». Автомат *A2* оказывается в этом состоянии после поступления события *e8*, которое означает что автомат *A1* сгененерировал новый *LZW*-код для автомата *A2*. Если выполнено логическое условие *x21* (этот код является кодом очистки), то автомат *A2* переходит в состояние 6. Если выполнено логическое условие *x22* (этот код является кодом завершения потока данных), то автомат *A2* переходит в терминальное состояние 0 и завершает свою работу. Если же ни одно из условий (ни *x21*, ни *x22)* не выполнено (*LZW*-код не является ни кодом очистки, ни кодом завершения потока данных), то автомат *A2* переходит в состояние 3 для продолжения обработки полученного кода.

Состояние «3. Начало генерации». Выходное воздействие *z23* в этом состоянии начинает генерацию значения нового *LZW*-кода во внутренней таблице автомата *A2*, исходя из кода, полученного автоматом *A2* по событию *e8* и обработанного в состоянии 2. Если по результатам воздействия *z23* оказывается, что полученный код является новым для автомата *A2* (логическая переменная *x23* принимает истинное значение), то этот автомат переходит из состояния 3 в состояние 4. Если же полученный код уже существует во внутренней таблице автомата *A2* (логическая переменная *x23* принимает ложное значение), то этот автомат переходит из состояния 3 в состояние 5.

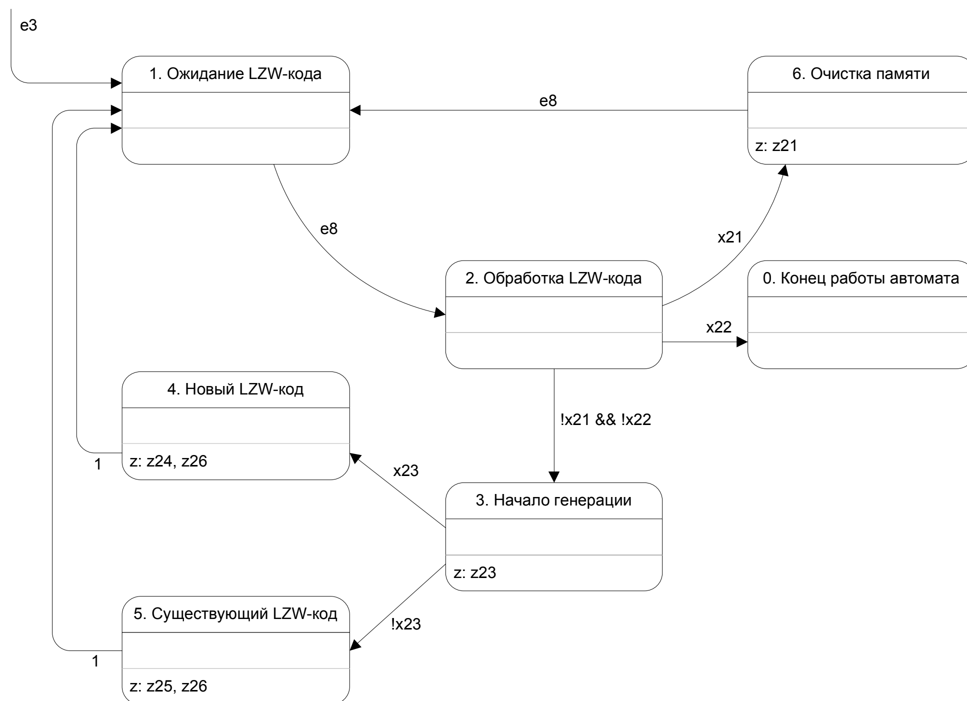
Состояние «4. Новый *LZW*-код». Выходные воздействия *z24* и *z26* в этом состоянии завершают обработку полученного *LZW*-кода как нового кода для автомата *A2*, обновляют внутреннюю таблицу и выводят на печать цепочку с номерами цветов, соответствующую этому коду. После этого автомат *A2* переходит в состояние 1 и готов к обработке следующего *LZW*-кода.

Состояние «5. Существующий *LZW*-код». Состояние 5 отличается от состояния 4 тем, что полученный *LZW*-код обрабатывается как уже существующий во внутренней таблице автомата *A2* (выходное воздействие *z25*, вместо воздействия *z24*).

Состояние «6. Очистка памяти». Выходное воздействие `z21` в этом состоянии производит начальную инициализацию внутренней таблицы *LZW*-кодов автомата *A2* и всех необходимых вычислительных переменных. Поскольку следующий за кодом очистки *LZW*-код в соответствии со стандартом *GIF* должен быть проигнорирован [5], то автомат *A2* покидает состояние 6 только по событию `e8`. При этом он не переходит в состояние 2, соответствующее обработке полученного кода, а переходит в состояние 1.

Состояние «0. Конец работы автомата». В этом терминальном состоянии автомат *A2* завершает свою работу, так как после получения кода завершения потока данных, изображение должно быть уже полностью декодировано этим автоматом.

На рис.6 приведена диаграмма состояний автомата *A2*.



x21	$data->{lzw} == $cc->{CLEAR_CODE}
x22	$data->{lzw} == $cc->{END_CODE}
x23	$data->{lzw} == $data->{FREE}

e3	Автомат A1 запускает в параллельном потоке автомат A2
e8	Автомат A1 передает очередной символ на вход автомату A2

z21	Очистка памяти
z23	Начало генерации нового LZW-кода
z24	Обработка LZW-кода как нового
z25	Обработка LZW-кода как существующего
z26	Обновление памяти

Рис. 6. Диаграмма состояний автомата *A2*

5.4. Описание автомата A3

Вспомогательный автомат *A3* является вложенным в автомат *A1* и отвечает за правильное чтение заголовков *GIF*-файла. Он имеет четыре состояния. Его начальным состоянием является состояние 1.

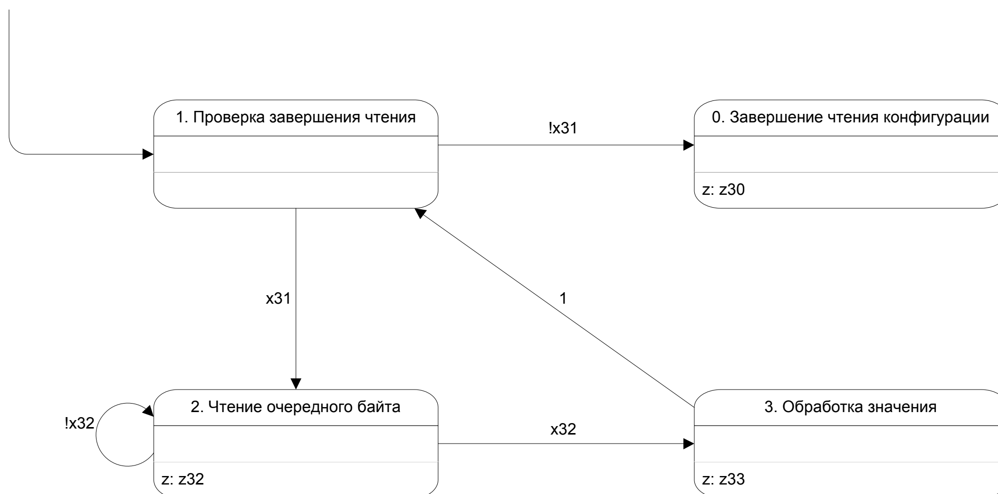
Состояние «1. Проверка завершения чтения». В этом состоянии автомат *A3* проверяет, остались ли в заголовке *GIF*-файла блоки данных, которые нужно прочитать и обработать. Если логическая переменная `x31` принимает истинное значение (такие блоки данных остались), то автомат *A3* приступает к чтению следующего блока, переходя в состояние 2. Если же данных в заголовке больше нет (переменная `x31` принимает ложное значение), то автомат *A3* переходит в терминальное состояние 0.

Состояние «2. Чтение очередного байта». Выходное воздействие `z32` в этом состоянии производит чтение очередного байта из текущего блока данных в заголовке *GIF*-файла и проверяет, не закончился ли текущий блок данных. Если логическая переменная `x32` принимает истинное значение (текущий блок данных закончился), то автомат *A3* приступает к обработке значения прочитанного блока данных, переходя в состояние 3. Если же в текущем блоке данных прочитано еще не все (переменная `x32` принимает ложное значение), то автомат *A3* остается в состоянии 2 для того чтобы повторить выходное воздействие `z32` необходимое число раз.

Состояние «3. Обработка значения». Выходное воздействие `z33` в этом состоянии производит все необходимое для преобразования прочитанного блока данных во внутреннюю структуру данных автомата *A3*. После этого автомат *A3* возвращается в состояние 1 для того чтобы проверить, остались ли еще в заголовке *GIF*-файла данные, которые необходимо прочитать и обработать.

Состояние «0. Завершение чтения конфигурации». Находясь в терминальном состоянии 0, автомат *A3* производит выходное воздействие `z30`, которое сохраняет прочитанную из заголовка *GIF*-файла конфигурацию, и возвращает управление вызывающему автомату *A1*.

На рис. 7 приведена диаграмма состояний автомата *A3*.



x31	$reader < @readers
x32	$readers[$reader]->[0]->($read, $value) != 0

z30	Инициализация данных о конфигурации
z32	Чтение очередного байта
z33	Обработка значения

Рис. 7. Диаграмма состояний автомата *A3*

5.5. Реализация автоматов

В демонстрационной программе конечные автоматы *A1*, *A2* и *A3* закодированы в процедурах `parse_gif($image)`, `process_lzw()` и `get_config($image)` соответственно.

Ключевой составляющей каждой из этих процедур является конструкция вида
`if ($y1 == 1) { } elsif ($y1 == 2) { } elsif … { }`,
аналогичная конструкции `switch` в языках *C++* и *Java*. Эта конструкция определяет следующее состояние конечного автомата на основе его текущего состояния, значения булевских переменных `$x[]` и состояния других автоматов, запущенных параллельно. После выполнения перехода в новое состояние, выполняются все выходные воздействия в этом состоянии, и осуществляется пересчет булевских переменных. Если текущее состояние автомата не является терминальным, то после пересчета булевских переменных происходит переход автомата в новое состояние.

Каждую из трех процедур, реализующих конечные автоматы, можно схематично представить следующим образом:

```
инициализация автомата;
пока (состояние не терминальное) {
      переход в следующее состояние;          // switch (y)
      выполнение выходных воздействий;        // z1, z2,…
      пересчет булевских переменных;          // x1, x2
}
возврат управления;
```

5.6. Различное взаимодействие автоматов

Поскольку автоматы *A1* и *A2* работают относительно независимо друг от друга, представляется интересным рассмотреть различные способы машинной интерпретации их взаимодействия.

Способ 1. Передача управления вместе с *LZW*-кодом. В этом случае автомат *A1*, находясь в состоянии 4, извещает автомат *A2* о том, что был прочитан новый *LZW*-код ***c*** (событие *e8*), и одновременно с этим передает управление автомату *A2*, приостанавливая на время его работы свое собственное выполнение. После того, как автомат *A2* выводит декодированное значение кода ***c*** в цепочку номеров цветов матрицы изображения, **управление снова возвращается автомату *A1***. Данный способ позволяет осуществить автоматную реализацию программы на системах с последовательной архитектурой, когда в один момент времени выполняется только одна машинная инструкция. **Схематично** такой способ представлен на рис. 8.

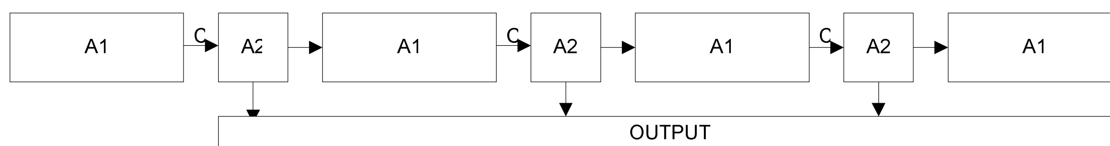


Рис. 8. Выполнение автоматов A1 и A2 в одном потоке

Способ 2. Параллельно-событийная реализация. В этом случае автоматы *A1* и *A2* запускаются одновременно в двух параллельных независимых друг от друга потоках операционной среды. По событию *e8* происходит синхронизация работы двух автоматов – автомат *A2* забирает очередной *LZW*-код со своего входа и начинает процесс его расшифровки. При этом вход автомата *A2* представляет собой не что иное, как выход автомата *A1*, реализованный в виде одиночного передаваемого из потока в поток символов. **Схематично** такой способ представлен на рис. 9.

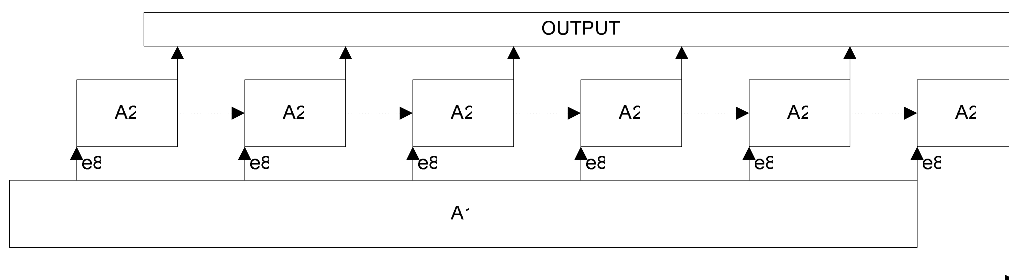


Рис. 9. Синхронизация автоматов *A1* и *A2* событием *e8*

Способ 3. Параллельная реализация с использованием очереди. В этом случае, в отличие от предыдущего способа, автоматы *A1* и *A2* выполняют свою работу в параллельных потоках асинхронно. Выход автомата *A1* и вход автомата *A2* реализуются в виде ***общей очереди*** (PIPE), специальной структуры данных. Находясь в состоянии 4, автомат *A1* добавляет очередной *LZW*-код в хвост этой очереди, а автомат *A2* забирает коды для последующей обработки из головы очереди в удобное этому автомату время. Такая реализация имеет смысл в том случае, если поток, в котором выполняется автомат *A2*, работает существенно медленнее потока, в котором выполняется автомат *A1*, либо же в случае, когда передать управляющий сигнал *e8* затруднительно, что возможно в некоторых типах распределенных систем.

Схематично такой способ представлен на рис. 10. Символы ***c1***, ***c2***, … соответствуют нескольким первым байтам, которые были считаны автоматом *A1* в блоке изображения.

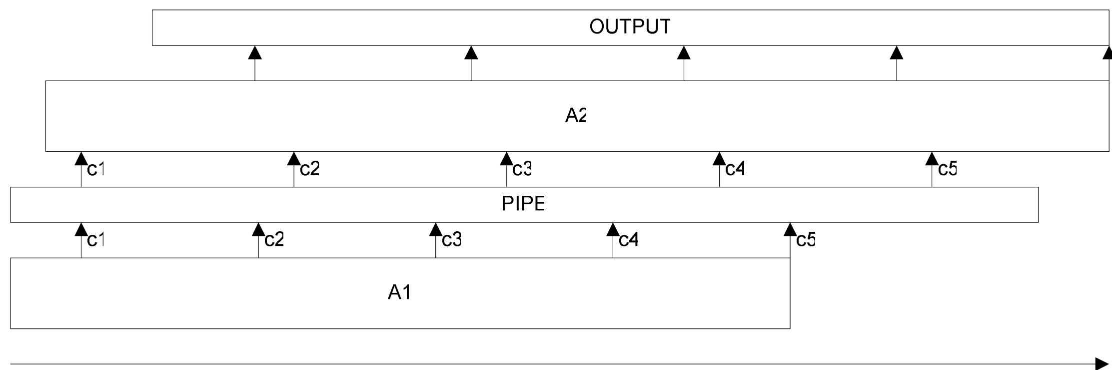


Рис. 10. Асинхронное выполнение автоматов *A1* и *A2*

В демонстрационной программе, исходные коды которой представлены в приложении 2, реализован первый (последовательный) способ взаимодействия автоматов *A1* и *A2*, так как такой способ является наиболее универсальным и машинно-независимым. В то же время, способы 2 и 3 являются достаточно перспективными, так как позволяют ввести в алгоритме декодирования файлов формата *GIF* естественную параллелизацию. В свете все возрастающей роли многопоточного программирования в мире теоретической информатики, представляется целесообразным использовать для параллелизации кода тот потенциал, который несет в себе применение конечных автоматов.

Заключение

Начавшись с идеи просто применить автоматный подход для галочки в списке курсовых работ, данное исследование привело в свою очередь к интересным научным результатам! Казалось бы, алгоритм декодирования потока битов в цепочку с номерами цветов представляет собой типичную вычислительную задачу. Однако, применение автоматного подхода к этой задаче позволило:

- получить возможность параллельной реализации там, где первоначально это даже не предусматривалось;
- разбить задачу на три относительно независимые подзадачи (три автомата);
- упростить понимание общей схемы декодирования за счет использования понятия «состояние».

Особенно ценным представляется провести дальнейшее и более глубокое исследование первой возможности. Очевидно, что ввести естественную параллелизацию в алгоритмах путем применения подхода автоматного программирования возможно далеко не всегда. В то же время, если задача выполняется при помощи нескольких конечных автоматов, то иногда становится возможным некоторые из них запускать в параллельных потоках.

В целом, в данной работе рассмотрены четыре возможные реализации задачи декодирования файлов формата *GIF*. Для двух последовательных реализаций, классической и автоматной, приведены работающие исходные коды. Два других параллельных решения на основе конечных автоматов (синхронное и асинхронное) представлены теоретически. Каждая из четырех реализаций данной задачи обладает своими достоинствами и недостатками. Классическая реализация быстро кодируется, но для ее возможной модификации требуется немало усилий. Код автоматной реализации изоморфен трем диаграммам состояний, поэтому его проще читать и модифицировать.

Кроме того, в данной работе представлен способ записи автоматных программ на языке программирования *Perl*, в котором отсутствует конструкция `switch`. Вместо нее используется эквивалентный набор условных операторов `if … elsif … else`.

Данная работа – это всего лишь небольшой шаг на пути к созданию современной теории о возможности распараллеливания произвольных алгоритмов. Однако этот шаг одновременно развивает и теорию автоматного программирования. Возможно, на стыке этих двух теорий нас ещё ожидает знатный урожай новых и полезных результатов!

Список источников

1. *GIF in Wikipedia*. http://en.wikipedia.org/wiki/GIF.
2. *Почему* на веб-страницах GNU не используется формат *GIF*. http://www.gnu.org/philosophy/gif.ru.html.
3. *LZW* in Wikipedia. http://en.wikipedia.org/wiki/LZW.
4. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1.
5. *GIF89a* specification. http://www.w3.org/Graphics/GIF/spec-gif89a.txt.
6. *Клименко В.В. , Хазановский А.Ю., Шалыто А.А.* Система управления автомобилем на шоссе. http://is.ifmo.ru/projects/carpilot/.

Приложение 1.
Исходный код программы с классической реализацией

```
#!/usr/bin/perl

#       Classic realization of Gif-file parser
#

use strict;
$| = 1;


my $image = '';
open(FILE, "_test.gif");
$image .= $_ while(<FILE>);
close(FILE);

my ($width, $height, $colors, $matrix) = parse_gif($image);

open(OUT, ">_test.txt");
binmode OUT;
print OUT "$width x $height x $colors\n";
for my $y (1..$height) {
        print OUT $matrix->[$_][$y] for (1..$width);
        print OUT "\n";
}
close(OUT);

exit 0;




sub parse_gif_classic {
        my $source            = shift;
        my @good_pixel = @_;
        my $width             = ord(substr($source, 6, 1)) + ord(substr($source, 7, 1)) * 256;
        my $height            = ord(substr($source, 8, 1)) + ord(substr($source, 9, 1)) * 256;
        my $bpp               = (ord(substr($source, 10, 1)) % 16) + 1;
        my $colors            = 1;
        $colors               *= 2 for (1..$bpp);
        unless(@good_pixel) {
                $good_pixel[$_] = $_ for (0..($colors - 1));
        }
        my $byte              = 13 + $colors * 3;
        $byte++
                while (substr($source, $byte, 1) ne ',');
        my $lzw_len           = ord(substr($source, $byte + 10, 1));
        my $first_over = 1;
        $first_over           *= 2 for (1..$lzw_len);
        my ($clear_code, $end_code, $free_code) =
                ($first_over, $first_over + 1, $first_over + 2);
        $first_over           *= 2;
        $byte                 += 11;

                # Get LZW sequence
        my $rString = '';
        my $counter = ord(substr($source, $byte, 1));
        while ($counter > 0) {
                $rString = $rString.substr($source, $byte + 1, $counter);
                $byte += $counter + 1;
                $counter = ord(substr($source, $byte, 1));
        }

                # LZW decode data
        my ($cByte, $cBit) = (0, 1);                        # current processed address
        my $cValue = ord(substr($rString, $cByte, 1));      # current processed value
        my (@code, @symbols, @c);
        for (0..($colors - 1)) {
                $code[$_][0] = $_;                          # decoded symbol[0] of @code[$_]
                $symbols[$_] = 0;                           # last array index for @code[$_]
        }
        my ($ffx, $ffy) = (1, 1);                   # current position in decoded matrix @c
        my $free = $free_code;                      # first current free code
        my $digits = $lzw_len + 1;                  # digits for code
        my $over_code = $first_over;                # first code with length > digits
        my ($old_lzw, $lzw) = (-1, 0);              # just readed codes
```

```
	while ($lzw != $end_code) {
			# enlarge digits for code if no more old free codes
		if ($free == $over_code) {
			$digits++;
			$over_code *= 2;
		}

			# Read new LZW code
		my $cM;
		($lzw, $cM) = (0, 1);
		for (1..$digits) {
			$lzw += $cM * ($cValue % 2);
				# Move one bit right
			$cValue = int($cValue / 2);
			$cM *= 2;
			$cBit++;
				# Touch new byte
			if ($cBit == 9) {
				$cBit = 1;
				$cByte++;
				$cValue = ord(substr($rString, $cByte, 1));
			}
		}

		if ($lzw == $clear_code) {
				($free, $digits, $over_code, $old_lzw) =
			($free_code, $lzw_len + 1, $first_over, -1);
		} elsif ($lzw != $end_code) {
				# Create new code
			if ($old_lzw > -1) {
				$symbols[$free] = $symbols[$old_lzw] + 1;
				$code[$free][$_] = $code[$old_lzw][$_] for (0..$symbols[$old_lzw]);
				$code[$free][$symbols[$free]] =
					($lzw == $free) ? $code[$old_lzw][0] : $code[$lzw][0];
				$free++;
			}

			$old_lzw = $lzw;

				# Put decoded $lzw to output matrix @c
			for (0..$symbols[$lzw]) {
				$c[$ffx][$ffy] = $good_pixel[$code[$lzw][$_]];
				$ffx++;				# Calculate next coordinates
				if ($ffx > $width) {
					$ffx = 1;
					$ffy++;
				}
			}
		}
	}
	return ($width, $height, $colors, \@c);
}
```

Приложение 2.
Исходный код программы с автоматной реализацией

```
#!/usr/bin/perl

#       Gif-file parser based on automata
#

use strict;
$| = 1;


my $image = '';
open(FILE, "_test.gif");
$image .= $_ while(<FILE>);
close(FILE);

my $index = 0;                # for next_symbol()
my ($y1, $y2, $y3);
my ($width, $height, $colors, $matrix) = parse_gif($image);

open(OUT, ">_test.txt");
binmode OUT;
print OUT "$width x $height x $colors\n";
for my $y (1..$height) {
        print OUT $matrix->[$_][$y] for (1..$width);
        print OUT "\n";
}
close(OUT);

exit 0;





# Automata-1 realization
#
sub parse_gif {
        my $img        = shift;
        my $buffer;                    # Readed, but not yet proceeded byte
        my $counter    = 0;            # Number of extra bytes in lzw-flow

        my $pict;                      # Global picture properties (hashref)
        my $cc;                        # Read-only LZW properties (hashref)
        my $data;                      # LZW properties, including Automata-2 state (hashref)
        my $code;                      # LZW code table (arrayref)
        my $matrix;                    # Output matrix (reference to array of array)


        my $y1  = -1;                  # Automata-1 state number
        my (@x, @z);                   # Automata-1 input and output signals
        @z[1..9] = ( sub {     ($pict, $cc) = get_config($img); },         # Call Automata-3
                              \&init,
                               sub { clear_code($data, $cc); },
                               sub { $buffer = ord(next_symbol($image)); },
                               sub { $counter = $buffer; },
                              \&update_value,
                              \&generate_lzw,
                               sub { process_lzw($cc, $data, $code); },    # Start Automata-2
                               sub { to_output($pict, $code->[$data->{lzw}], $matrix); } );
        my @actions = ( [], [1, 2, 3],                         # List of Automata-1 actions
                           [4, 5],
                           [4, 6],
                           [7, 8, 9] );


                       # 0 - end state
                       # 1 - init config
                       # 2 - get counter
                       # 3 - continue read input
                       # 4 - print lzw code
        while ($y1 != 0) {
                       # Jump :)
               if      ($y1 == 1) {
                               $y1 = 2;
```

```
			} elsif ($y1 == 2) {
				if (!$x[1]) {
					$y1 = 0;
				} else {
					$y1 = 3;
				}
			} elsif ($y1 == 3) {
				if ($y2 == 0) {
					$y1 = 0;
				} elsif (!$x[1]) {
					$y1 = 2;
				} elsif ($x[1] && ($y2 != 0) && !$x[3]) {
					$y1 = 3;
				} elsif ($x[1] && ($y2 != 0) && $x[3]) {
					$y1 = 4;
				}
			} elsif ($y1 == 4) {
				if ($y2 == 0) {
					$y1 = 0;
				} elsif (!$x[1] && ($y2 != 0) && !$x[3]) {
					$y1 = 2;
				} elsif ($x[1] && ($y2 != 0) && !$x[3]) {
					$y1 = 3;
				} elsif (($y2 != 0) && $x[3]) {
					$y1 = 4;
				}
			} else {
					$y1 = 1;
			}

				# Action at state
			$z[$_]->() for (@{$actions[$y1]});

				# Check for input signals
			$x[1] = ($counter > 0);
#			$x[2] = ($data->{state} != 0);
			$x[3] = ($data->{bit} >= $data->{digits});
		}

		return (@$pict{'width', 'height', 'colors'}, $matrix);



		sub init {				# Z2 @ state_1
				# begin lzw_decode
			for (0..($pict->{colors} - 1)) {
				$code->[$_]	= $_;		# decoded symbol[0] of @code[$_]
			}
			$pict->{ffx} = 1;			# current position in decoded matrix
			$pict->{ffy} = 1;

			$data->{bit} = 0;			# current processed address
			$data->{z} = 1;
			$data->{value} = 0;			# current processed value
			$y2 = 1;				# start state of Automata-2
		}


		sub update_value {		# Z6 @ state_3
			$data->{value} = $data->{value} + ($data->{z}) * $buffer;
			$data->{bit} += 8;
			$data->{z} *= 256;
			$counter--;
		}


		sub generate_lzw {		# Z7 @ state_4
			$data->{old_lzw} = $data->{lzw};
			$data->{lzw} = 0;
			$data->{m} = 1;
			for (1..$data->{digits}) {
				$data->{bit}--;
				$data->{z} = $data->{z} / 2;
				$data->{lzw} += $data->{m} * ($data->{value} % 2);
				$data->{m} = $data->{m} * 2;
				$data->{value} = int($data->{value} / 2);
			}
		}
}
```

```
# Automata-2 realization
#
sub process_lzw {
	my ($cc, $data, $code) = @_;

	my $new_iteration = 1;
	my (@z, @actions);
	@z[20, 21, 23, 24, 25, 26] = (				# Automata-2 actions at state
			sub { $new_iteration = 0; },			# Stop Automata-2
			sub { clear_code($data, $cc); },
			sub { $code->[$data->{free}] = $code->[$data->{old_lzw}]; },
			sub { $code->[$data->{free}] .= substr($code->[$data->{old_lzw}], 0, 1); },
			sub { $code->[$data->{free}] .= substr($code->[$data->{lzw}], 0, 1); },
			\&release_code );
	@actions[1..6] = ( [20], [], [23],				# List of Automata-2 actions
			   [24, 26], [25, 26], [21, 20] );

	while (($y2 != 0) && ($new_iteration)) {
			# Jump :)
		if ($y2 == 1) {
				$y2 = 2;
		} elsif ($y2 == 2) {
			if ($data->{x}->[21]) {
				$y2 = 6;
			} elsif ($data->{x}->[22]) {
				$y2 = 0;
			} else {
				$y2 = 3;
			}
		} elsif ($y2 == 3) {
			if ($data->{x}->[23]) {
				$y2 = 4;
			} else {
				$y2 = 5;
			}
		} elsif ($y2 == 4) {
				$y2 = 1;
		} elsif ($y2 == 5) {
				$y2 = 1;
		} elsif ($y2 == 6) {
				$y2 = 1;
		}

			# Action at state
		$z[$_]->() for (@{$actions[$y2]});

			# Check for input signals
		$data->{x}->[21] = ($data->{lzw} == $cc->{clear_code});
		$data->{x}->[22] = ($data->{lzw} == $cc->{end_code});
		$data->{x}->[23] = ($data->{lzw} == $data->{free});
	}
	return;



	sub release_code {
		$data->{free}++;

			# enlarge digits for code if no more old free codes
		if ($data->{free} == $data->{over_code}) {
			$data->{digits}++;
			$data->{over_code} *= 2;
		}
	}
}
```

```
# Automata-3 realization
#
sub get_config {
	my $img = shift;
	my $pict;					# Global picture properties (hashref)
	my $cc;						# Read-only LZW properties (hashref)

		# Initialize read-only %$pict
	my @readers = ([sub { $_[0] == 7 }, sub { $pict->{width}   =  $_[0];              } ],
	               [sub { $_[0] == 1 }, sub { $pict->{width}   += $_[0] * 256;        } ],
	               [sub { $_[0] == 1 }, sub { $pict->{height}  =  $_[0];              } ],
	               [sub { $_[0] == 1 }, sub { $pict->{height}  += $_[0] * 256;        } ],
	               [sub { $_[0] == 1 }, sub { $pict->{bpp} = ($_[0] % 16) + 1;
	                                          $pict->{colors}      = 1;
	                                          $pict->{colors}*= 2 for (1..$pict->{bpp});  } ],
	               [sub { $_[0] == (2 + $pict->{colors} * 3) },sub {   } ],
	               [sub { $_[1] == 44}, sub {           } ], # eq ','
	               [sub { $_[0] == 9 }, sub {           } ],
	               [sub { $_[0] == 1 }, sub { $cc->{lzw_len} = $_[0];             } ] );
	my $reader = 0;
	my ($read, $value) = (0, 0);


	my $y3 = -1;
	my (@x, @z);					# Automata-3 input and output signals
	@z[30, 32, 33] = (\&init_cc, \&get_symbol, \&process_value);
	my @actions = ( [30], [], [32], [33] );		# List of Automata-3 actions

							# 0 - end state
							# 1 - check for readers
							# 2 - read next_symbol
							# 3 - process value
	while ($y3 != 0) {
			# Jump :)
		if		($y3 == 1) {
			if ($x[31]) {
				$y3 = 2;
			} else {
				$y3 = 0;
			}
		} elsif ($y3 == 2) {
			if ($x[32]) {
				$y3 = 3;
			} else {
				$y3 = 2;
			}
		} else {
				$y3 = 1;
		}

			# Action at state
		$z[$_]->() for (@{$actions[$y3]});

			# Check for input signals
		$x[31] = ($reader < @readers);
		$x[32] = (($readers[$reader]) && ($readers[$reader]->[0]->($read, $value)));
	}

	return ($pict, $cc);


	sub init_cc {		# Z30 @ state_0
		# Initialize %$cc
		$cc->{first_over} = 1;
		$cc->{first_over} *= 2 for (1..$cc->{lzw_len});
		@$cc{'clear_code', 'end_code', 'free_code'} =
		         ($cc->{first_over}..($cc->{first_over} + 2));
		$cc->{first_over} *= 2;
	}

	sub get_symbol {	# Z32 @ state_2
		$read++;
		$value = ord(next_symbol($img));
	}
	sub process_value { # Z33 @ state_3
		$readers[$reader]->[1]->($value);
		$read = 0;
		$reader++;
	}
```

```
}
# clear_code($data, $cc);
#
sub clear_code {
	$_[0]->{free}		= $_[1]->{free_code};		# first current free code
	$_[0]->{digits}		= $_[1]->{lzw_len} + 1;		# digits for code
	$_[0]->{over_code}	= $_[1]->{first_over};		# first code with length > digits
	$_[0]->{lzw}		= $_[1]->{clear_code};		# current lzw code
	$_[0]->{old_lzw}	= -1;				# old lzw code
}



sub to_output {
	my ($pict, $symbols) = @_;

	foreach (split(//, $symbols)) {
		$_[2]->[$pict->{ffx}][$pict->{ffy}] = $_;
		$pict->{ffx}++;				# Calculate next coordinates
		if ($pict->{ffx} > $pict->{width}) {
			$pict->{ffx} = 1;
			$pict->{ffy}++;
		}
	}
}


sub next_symbol {
	return ($index >= length($_[0])) ? -1 : substr($_[0], $index++, 1);
}
```