Статья опубликована в журнале «Мир ПК», 2005, № 7, с. 66-70, а также «Мир ПК–Диск», 2005, № 6, 7.

Подсчет длин слов в строке на основе автоматного подхода

Павел Лобанов, Анатолий Шалыто

В статье рассматривается решение задачи подсчета длин слов в строке на основе автоматного подхода. Описан ряд методов реализации автоматов.

В книге [1] в главе “Методы программирования от состояний” предлагается несколько процедурных способов реализации автоматов, а в главе «Объектно-ориентированный подход» предлагаются идеи по реализации автоматов с помощью объектов. К сожалению, описание предложенных в книге методов реализации автоматов является недостаточно четким, что может создать сложности при их применении на практике.

Часть этих методов описана также и в книге [2], в которой появились главы, названные со ссылкой на работу [3], «Автоматное программирование».

В данной работе рассматривается пять методов реализации автоматов: четыре процедурных и один объектно-ориентированный. Объектно-ориентированный метод в книге [1] представляет собой лишь набор идей, применение которых невозможно без дополнительной доработки.

Авторы данной работы попытались устранить указанные недостатки. В работе предложены шаблоны для реализации автоматов различными методами. Кроме этого сделана попытка, выделить общую часть реализации автоматов, не зависящую от метода процедурной реализации.

Применение методов, как и в книге [1], иллюстрируется на примере подсчета длин слов в тексте. Примеры написаны на языке программирования C++.

Для изображения графа переходов автомата используется нотация, предложенная в работе [4].

1. Постановка задачи

На вход программы подается **строка символов**, состоящая из слов и разделителей. Она завершается **символом перевода строки** *‘\n’*. Под **словом** понимается непустая последовательность букв латинского алфавита. Его символы будем называть **символами слова**. **Разделителем** считается любой символ, не являющийся **символом слова** или **символом перевода строки**.

Программа должна преобразовать:

<строку символов>,

в последовательность строк вида:

<слово> - <длина слова>.

Например, если на вход программы подана строка

dasda fsfллfsdfs fsdf2dfsfs

то программа формирует следующую выходную последовательность строк:

dasda - 5
fsf - 3
fsdfs - 5
fsdf - 4
dfsfs - 5

Как отмечалось во введении, в работе строятся пять программ, в каждой из которых используется автоматный подход, однако методы реализации автомата в этих программах различны.

2. Автомат. Описание и формализация

Приведем словесное описание алгоритма подсчета длин слов в тексте.

При получении **символа слова**, он записывается в выходной поток. При этом счетчик символов в слове увеличивается на единицу. При получении **разделителя**, если перед ним поступил **символ слова**, то в выходной поток записывается строка вида:

-<длина слова><перевод строки>

и счетчик длины слова сбрасывается в ноль.

При получении **символа перевода строки** автомат завершает работу и, если перед ним поступил **символ слова**, в выходной поток записывается строка вида:

-<длина слова><перевод строки>

Для решения поставленной задачи построим автомат, в котором переходы выполняются по событию **e0 –** “Приход символа из входного потока”. Введем две булевы переменные **x1** и **x2**. Переменная **x1** принимает значение **истина,** если по событию **e0** пришел **символ слова**, а переменная **x2** принимает то же значение**,** если по событию **e0** пришел **символ перевода строки**. Заметим, что переменные **x1** и **x2** не могут принимать значение истина одновременно.

Построим автомат с тремя состояниями и опишем каждое из них.

0. Ожидания начала слова – автомат начинает работу в этом состоянии и переходит в состояние *1*, если **x1 -** истина (получен **символ слова**), и в состояние *2*, если **x2** –истина (получен **символ перевода строки**);

1. Обработка слова – автомат переходит из этого состояния в состояние *0*, если **x1 –** ложь и **x2** – ложь (получен **разделитель**), и в состояние *2*, если **x2 -** истина (получен **символ перевода строки**);

2. Работа завершена – при попадании в это состояние автомат завершает работу. Переходы из этого состояния в другие невозможны.

Опишем действия, выполняемые на переходах:

- **z1_0** (*записать символа в выходной поток*) – выполняется при переходе из состояния *0* в состояние *1*, а также когда автомат остается в состоянии *1* при приходе нового символа;
- **z1_1** (*записать в выходной поток «длину слова»*) – выполняется при переходе из состояния *1* в состояния *0* и *2*.

- **z2_0** (*установить счетчик символов в слове в **0***) – выполняется при переходе из состояния *0* в состояние *1*, а также в том случае, когда автомат остается в состоянии *1* при приходе нового символа.
- **z2_1** (*увеличить счетчик символов в слове на единицу*) – выполняется при переходе из состояния *1* в тоже состояние.

Схема связей автомата, описывающая его интерфейс, приведена на рис. 1, а граф переходов, описывающий поведение автомата – на рис. 2.

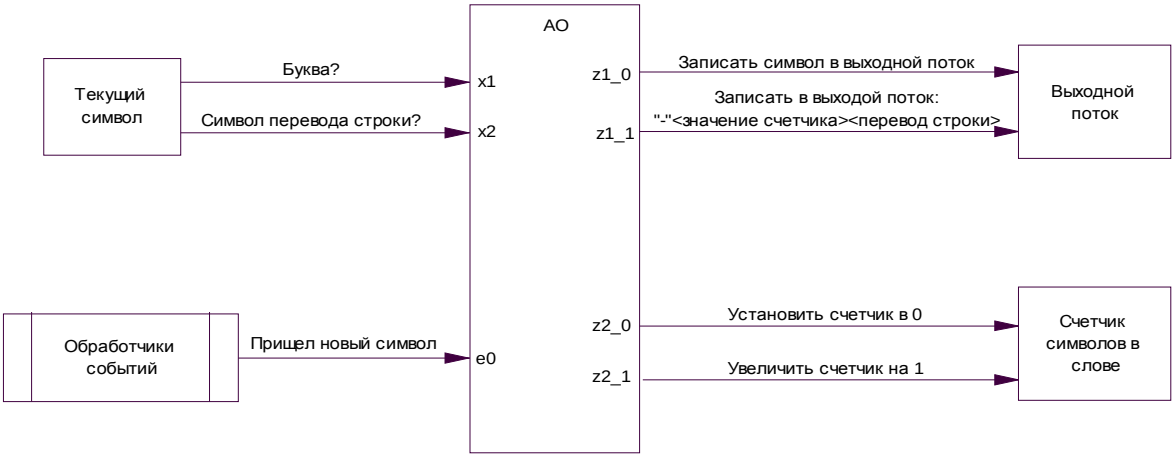

Рис 1. Схема связей автомата *A0*

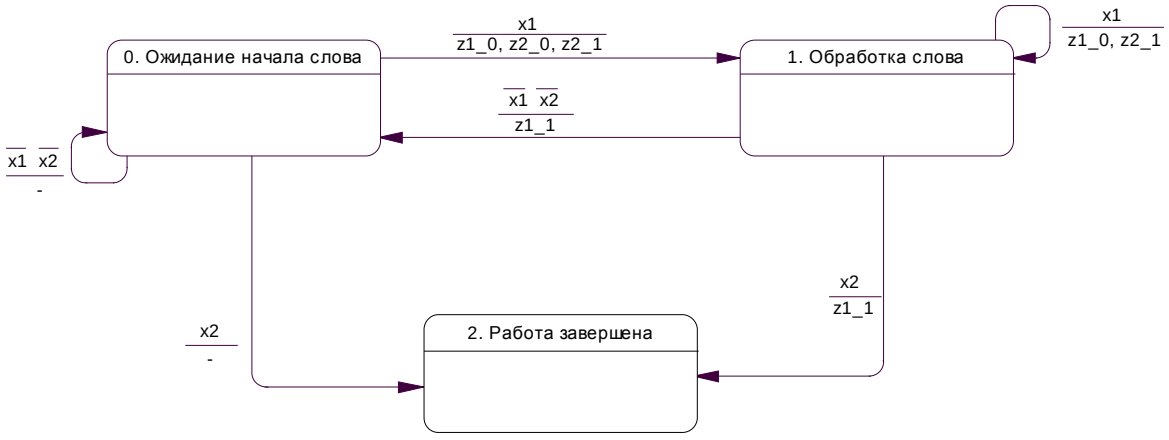

Рис 2. Граф переходов автомата *AO*

3. Методы реализации автомата

3.1. Общие функции и переменные для процедурных способов реализации автомата

Во всех рассматриваемых реализациях автомата с помощью процедурных методов, функции для входных переменных, выходных воздействий, внутренние переменные, а также функция, реализующая событие, не изменяются.

Объявление функций автомата *A0*

Объявление (Листинг 1) этих функций состоит из следующих частей:

- объявление функций входных переменных;
- объявление функций выходных воздействий;
- объявление функции, реализующей событие.

Листинг 1. Объявления функций автомата *A0*

```
////////////////////////////////////////////////////////
// Входные переменные

// Возвращает значение true, если текущий символ (curSymbol) -
// буква латинского алфавита, и значение false - в противном случае.
bool x1();

// Возвращает значение true, если текущий символ (curSymbol) -
// перевод строки '\n', и значение false - в противном случае.
bool x2();


////////////////////////////////////////////////////////
// Выходные воздействия

// Записывает текущий символ (curSymbol) в выходной поток.
void z1_0();

// Записывает в выходной поток:
// - symbolCount '\n'
void z1_1();


// Устанавливает счетчик символов в слове (symbolCount) в ноль.
void z2_0();

// Увеличивает значение счетчика символов в
// слове (symbolCount) на единицу.
void z2_1();

////////////////////////////////////////////////////////
// Функция, реализующая событие

// Событие - приход нового символа.
// Обновляется текущий символ.
// Перед первым вызов требуется вызвать функцию initInputStream().
void e0();

// Инициализация входного потока.
// Требуется вызвать перед первым вызовом функции e0().
void initInputStream();
```

Реализация функций и определение внутренних переменных для автомата *A0*

Реализация этих функций и определение переменных (Листинг 2) состоит из следующих частей:

- определение внутренних переменных;
- реализация функций входных переменных;

- реализация функций выходных воздействий;
- реализация функции, реализующей событие.

Листинг 2. Реализация функций и определение переменных автомата *A0*

```
//////////////////////////////////////////////////////////
// commonfunctions.cpp

#include "commonfunctions.h"
#include <stdio.h>

//////////////////////////////////////////////////////////
// Внутренние переменные

// Максимальный размер строки, которая будет использоваться
// в качестве входного потока (INPUT_STREAM_SIZE).
#define INPUT_STREAM_SIZE 16384

// Текущий символ.
char curSymbol;

// Счетчик символов в слове.
int symbolCount;

// Строка, которая используется в
// качестве входного потока.
char inputStream[16384];

// Указатель на текущий символ в inputStream.
char * curInputChar;

//////////////////////////////////////////////////////////
// Входные переменные.

// Возвращает значение true, если текущий символ (curSymbol) -
// буква латинского алфавита, и значение false - в противном случае.
bool x1() {

      return (curSymbol >= 'a' && curSymbol <= 'z') ||
               (curSymbol >= 'A' && curSymbol <= 'Z');
}

// Возвращает значение true, если текущий символ (curSymbol) -
// символ перевода строки '\n', и значение false в противном случае.
bool x2() {

      return curSymbol == '\n';
}

//////////////////////////////////////////////////////////
// Выходные воздействия.

// Записывает текущий символ (curSymbol) в выходной поток.
void z1_0() {

      printf("%c", curSymbol);
}

// Записывает в выходной поток:
// - symbolCount '\n'
void z1_1() {

    printf(" - %d\n", symbolCount);
```

```
}

// Устанавливает счетчик символов в слове (symbolCount) в ноль.
void z2_0() {

    symbolCount =  0;
}

// Увеличивает значение счетчика символов в
// слове (symbolCount) на единицу.
void z2_1() {

    symbolCount++;
}

//////////////////////////////////////////////////////
// Функция, реализующая событие.

// Событие - приход нового символа.
// Обновляется текущий символ.
// Перед первым вызов требуется вызнать initInputStream().
void e0() {

    if(curInputChar == inputStream + INPUT_STREAM_SIZE) {
        curSymbol = '\n';
        return;
    }

    curSymbol = *curInputChar++;
}

// Инициализация входного потока.
// Требуется вызвать перед первым вызовом e0().
void initInputStream() {

    curInputChar = inputStream;

    while ((*curInputChar++ = getchar()) != '\n') {
        if(curInputChar == inputStream + INPUT_STREAM_SIZE)
            break;
    }

    curInputChar = inputStream;
}
```

Дальнейшее изложение выполняется по следующей схеме:

- словесное описание метода;
- шаблон для реализации метода;
- реализация рассматриваемого примера.

3.2. Метод 1. Каждому состоянию сопоставляется рекурсивная функция

Для каждого состояния пишется функция, в которой реализуются все возможные действия в этом состоянии. Для запуска автомата в данном случае, требуется вызвать функцию, соответствующую начальному состоянию автомата.

Существенный недостаток данного метода состоит в том, что в нем используются рекурсивные функции. Поэтому количество переходов по дугам, которые может

совершить автомат, реализованный таким образом, ограничено размером стека. По этой причине метод нецелесообразно применять на практике.

При вызове каждой из этих функции выполняются следующие операции.

1. Ожидается событие.
2. Путем проверки входных переменных выбирается дуга перехода.
3. Выполняются выходные воздействия, соответствующие выбранной дуге перехода.
4. Осуществляется переход в новое состояние с помощью вызова функции, соответствующей новому состоянию.

Шаблон для функции, реализующей действия для состояния

```
void sI() {

    // Ждем, пока произойдет событие.
    e0();

    if(/* Поверка  входных переменных для первой дуги. */) {

        // Выполнение выходных воздействий.
        zN(); ... zM();

        // Переход в новое состояние J.
        sJ();
    } else if (/* Проверка  входных переменных для второй дуги */) {
        ...
    } else if ...
}
```

Реализуем пример с помощью данного метода (Листинги 3 и 4).

Листинг 3. Объявление функций, реализующих действия состояний для автомата *A0*

```
//////////////////////////////////////////////////////////
// 1.h

// Функция, соответствующая состоянию 0 ("Ожидание начала слова").
void s0();

// Функция, соответствующая состоянию 1 ("Обработка слова").
void s1();

// Функция, соответствующая состоянию 2 ("Работа завершена").
void s2();
```

Листинг 4. Определение функций, реализующих состояния для автомата *A0*, и главной функции программы

```
//////////////////////////////////////////////////////////
// 1.cpp

#include <stdio.h>

#include "../common/commonfunctions.h"
#include "1.h"

// Функция, соответствующая состоянию 0 ("Ожидание начала слова").
void s0() {
```

```
    e0();

    if(x1()) {
        z1_0();
        z2_0();
        z2_1();
        s1();
    } else if (x2()) {
        s2();
    } else {
        s0();
    }
}

// Функция, соответствующая состоянию 1 ("Обработка слова").
void s1() {

    e0();

    if(x1()) {
        z1_0();
        z2_1();
        s1();
    } else if(x2()) {
        z1_1();
        s2();
    } else {
        z1_1();
        s0();
    }
}

// Функция, соответствующая состоянию 2 ("Работа завершена").
void s2() {
}

// Главная функция программы.
int main() {
    initInputStream();
    s0();

    printf("Press \"Enter\" to exit.");
    while(getchar() != 10);
}
```

3.3. Метод 2. Реализация автомата с помощью оператора `switch`

Для того, что бы избавиться от рекурсивных вызовов, используемых в *методе 1*, можно ввести переменную, в которой хранить текущее состояние автомата, а вместо вызова функции, соответствующей состоянию автомата, изменять значение этой переменной.

Шаблон для реализации автомата

```
int main() {

    // Устанавливаем автомат в начальное состояние.
    State_t state = s0;
```

```
    // Цикл выполняется, пока автомат не придет в конечное состояние sX.
    while(state != sX) {

        // Ждем, пока не произойдет событие.
        e0();

        switch(state) {
            case s0:
                if(/* Проверка  входных переменных для первой дуги. */)
                {
                    // Выполнение выходных воздействий.
                    zN(); ... zM();

                    // Переход в новое состояние I.
                    state = sI;
                }
                else if(/* Проверка  входных переменных для второй дуги. */)
                {
                    state = s2;
                }
                else if ...
                break;

            case s1:
                ...
                break;

            ...

            case sX:
                ...
                break;
        }
    }

    return 0;
}
```

Реализуем пример с помощью данного метода (Листинги 5 и 6).

Листинг 5. Объявление перечислимого типа, определяющего состояния автомата *A0*

```
//////////////////////////////////////////////////////
// 2.h

// Состояния, в которых может находиться автомат A0.
typedef enum States {
    s0, // 0: "Ожидание начала слова".
    s1, // 1: "Обработка слова".
    s2  // 2: "Работа завершена".
} State_t;
```

Листинг 6. Реализация автомата *A0*

```
//////////////////////////////////////////////////////
// 2.cpp

#include <stdio.h>

#include "../common/commonfunctions.h"
#include "2.h"

// Главная функция автомата.
int main() {
```

```
    // Текущее состояние автомата.
    State_t state = s0;

    initInputStream();

    while(state != s2) {

        e0();

        switch(state) {
            case s0:
                if(x1()) {
                    z1_0();
                    z2_0();
                    z2_1();
                    state = s1;
                } else if (x2()) {
                    state = s2;
                }
                break;

            case s1:
                if(x1()) {
                    z1_0();
                    z2_1();
                } else if(x2()) {
                    z1_1();
                    state = s2;
                } else {
                    z1_1();
                    state = s0;
                }
                break;

            case s2:
                break;
        }
    }

    printf("Press \"Enter\" to exit.");
    while(getchar() != 10);

    return 0;
}
```

3.4. Метод 3. Каждому состоянию сопоставляется нерекурсивная функция

Этот метод основан на двух предыдущих. Для каждого состояния реализуется функция (в данном случае нерекурсивная), которая определяет операции – выходные воздействия для этого состояния и состояния, в которые автомат может перейти. В отличие от первого метода переход в новое состояние осуществляется не внутри функции, а в операторе `switch`. Функция только возвращает следующее состояние. Как и во втором методе для хранения текущего состояния вводится переменная, используемая в операторе `switch`.

Шаблон для функции, реализующей операции для состояния

```
State func_s1() {
```

```
    if(/* Проверка  входных переменных для первой дуги. */) {

        // Выполнение выходных воздействий.
        zN(); ... zM();

        // Возвращаем новое состояние I.
        return sI;
    } else if (/* Проверка  входных переменных для второй дуги. */) {
        ...
    } else if ...
}
```

Шаблон для реализации автомата

```
int main() {

    // Устанавливаем автомат в начальное состояние.
    State_t state = s0;

    // Цикл выполняется, пока автомат не придет в конечное состояние sX.
    while(state != sX) {

        // Ждем, пока не произойдет событие.
        e0();

        switch(state) {
            case s0:
                state = func_s0();
                break;

            case s1:
                state = func_s1();
                break;

            ...

            case sX:
                state = func_sX();
                break;
        }
    }

    return 0;
}
```

Реализуем пример с помощью данного метода (Листинги 7 и 8).

Листинг 7. Объявление перечислимого типа, определяющего состояния автомата *A0*, и функций, реализующих операции для его состояний

```
////////////////////////////////////////////////////////
// 3.h

// Состояния, в которых может находиться автомат A0.
typedef enum States {
    s0, // 0: "Ожидание начала слова".
    s1, // 1: "Обработка слова".
    s2  // 2: "Работа завершена".
} State_t;

// Функция, вызываемая по событию, если автомат
// находится в состоянии 0 ("Ожидание начала слова").
State_t func_s0();
```

```
// Функция, вызываемая по событию, если автомат
// находится в состоянии 1 ("Обработка слова").
State_t func_s1();

// Функция, вызываемая по событию, если автомат
// находится в состоянии 2 ("Работа завершена").
State_t func_s2();
```

Листинг 8. Определение функций, реализующих состояния автомата *A0*, и главной функции

```
////////////////////////////////////////////////////////
// 3.cpp

#include <stdio.h>

#include "../common/commonfunctions.h"
#include "3.h"

// Функция, вызываемая по событию, если автомат
// находится в состоянии 0 ("Ожидание начала слова").
State_t func_s0() {

    if(x1()) {
        z1_0();
        z2_0();
        z2_1();
        return s1;
    } else if (x2()) {
        return s2;
    } else {
        return s0;
    }
}

// Функция, вызываемая по событию, если автомат
// находится в состоянии 1 ("Обработка слова").
State_t func_s1() {

    if(x1()) {
        z1_0();
        z2_1();
        return s1;
    } else if(x2()) {
        z1_1();
        return s2;
    } else {
        z1_1();
        return s0;
    }
}

// Функция, вызываемая по событию, если автомат
// находится в состоянии 2 ("Работа завершена").
State_t func_s2() {

    return s2;
}

// Главная функция программы.
int main() {
```

```
    // Текущее состояние автомата.
    State_t state = s0;

    initInputStream();

    while(state != s2) {

        e0();
        switch(state) {
            case s0:
                state = func_s0();
                break;

            case s1:
                state = func_s1();
                break;

            case s2:
                state = func_s2();
                break;
        }
    }

    printf("Press \"Enter\" to exit.");
    while(getchar() != 10);

    return 0;
}
```

3.5. Метод 4. Реализация автомата с помощью безусловных переходов по меткам

В данном методе предлагается использовать метки и операторы безусловного перехода для управления состояниями автомата. При изменении состояния автомат переходит на метку, соответствую новому состоянию. Такой подход позволяет избавиться от необходимости хранить состояние автомата в переменной.

Шаблон для реализации автомата

```
// Главная функция программы.
int main() {

// Метка начального состояния.
s0:
    // Ожидаем, пока произойдет событие.
    e0();
    if(/* Проверка  входных переменных для первой дуги. */) {

        // Выполнение выходных воздействий.
        zN(); ... zM();

        // Переходим в состояние I.
        goto sI;
    } else if (/* Проверка  входных переменных для второй дуги. */) {
        ...
    } else if ...

s1:
  ...
```

```
...

// Метка заключительного состояния.
sX:
    return 0;
}
```

Реализуем пример с помощью данного метода (Листинг 9).

Листинг 9. Реализация автомата *A0*

```
//////////////////////////////////////////////////////////
// 4.cpp

#include <stdio.h>

#include "../common/commonfunctions.h"
#include "4.h"

// Главная функция программы.
int main() {

    initInputStream();

// Состояние 0: "Ожидание начала слова".
s0:
    e0();
    if(x1()) {
        z1_0();
        z2_0();
        z2_1();
        goto s1;
    } else if (x2()) {
        goto s2;
    } else {
        goto s0;
    }

// Состояние 1: "Обработка слова".
s1:
    e0();
    if(x1()) {
        z1_0();
        z2_1();
        goto s1;
    } else if(x2()) {
        z1_1();
        goto s2;
    } else {
        z1_1();
        goto s0;
    }

// Состояние 2: "Работа завершена".
s2:

    printf("Press \"Enter\" to exit.");
    while(getchar() != 10);

    return 0;
}
```

3.6. Метод 5. Объектно-ориентированный подход. Класс сопоставляется состоянию автомата

При применении данного метода предлагается описать все состояния автомата как наследники базового класса состояния. При этом в базовом классе определяются функции входных переменных, выходных воздействий и указатель на объект данных автомата (например, счетчик символов в слове), хранящий внутренние переменные автомата и объекты классов его состояний. Кроме этого, в базовом классе задается абстрактный метод *execute*, возвращающий указатель на объект базового класса состояния. Этот метод предназначен для определения нового состояния автомата и должен быть переопределен в классах наследниках.

Для работы автомата требуется вызывать указанный метод у текущего объекта класса состояния, изменяя текущее состояние на результат выполнения метода, до тех пор, пока не придем в конечное состояние.

В объекте данных автомата хранятся указатели на объекты классов состояний, внутренние данные автомата, методы реализующие события. Объекты классов состояний автомата создаются и разрушаются вместе с классом данных в его конструкторе и деструкторе соответственно.

Шаблон интерфейса для базового класса состояния

Интерфейс для базового класса автомата включает в себя следующие части:

- указатель на объект, содержащий внутренние данные автомата (*pMainData*);
- объявление функций входных переменных;
- объявление функций выходных воздействий;
- объявление конструктора и деструктора;
- объявление абстрактного метода реализующего действия для состояния (*execute*).

```
// Базовый класс для состояний автомата A0.
class S {

protected:

    // Указатель на объект данных, содержащий внутренние переменные и
    // объекты, соответствующие классам состояний автомата.
    MainData *pMainData;

    ////////////////////////////////////////////////////////
    // Входные переменные.
    bool x1();
    ...
    bool xN();

    ////////////////////////////////////////////////////////
    // Выходные воздействия.
    void z0();
    ...
    void zK();

public:
    // Конструктор.
    S(MainData *pMainData);
    virtual ~S() {}
    // Метод, реализующий действия для состояния.
    virtual S* execute()=0;
```

```
};
```

Шаблон реализации для базового класса состояния

Реализация для базового класса состояния включает в себя следующие части:

- реализацию функций входных переменных;
- реализацию функций выходных воздействий;
- реализацию конструктора.

```
//////////////////////////////////////////////////////
// Входные переменные.
bool S::x1() {

      return ...
}
...
bool S::xN() {

      return ...
}

//////////////////////////////////////////////////////
// Выходные воздействия.
void S::z0() {

      ...
}
...
void S::zK() {

      ...
}

// Конструктор.
S::S(MainData *pMainData):pMainData(pMainData){

}
```

Шаблон для интерфейса класса состояния

Интерфейс для класса состояния включает в себя следующие части:

- объявление конструктора;
- объявление метода, реализующего действия для состояния (*execute*).

```
class S0 : public S {
public:
    // Конструктор.
    S0(MainData *pMainData);

    // Метод, реализующий действия для состояния.
    virtual S* execute();
};
```

Шаблон для реализации класса состояния

Реализация для класса состояния включает в себя следующие части:

- определение конструктора;

- определение метода, реализующего действия для состояния (*execute*).

```
// Конструктор.
S0::S0(MainData *pMainData):S(pMainData) {

}

// Метод, реализующий действия для состояния.
S* S0::execute() {

    // Ожидаем событие.
    pMainData->e0();

    if(/* Проверка входных переменных для первой дуги. */) {

        // Выполнение выходных воздействий.
        zN(); ... zM();

        // Возвращаем новое состояние I.
        return pMainData->pSI;
    } else if (/* Проверка входных переменных для второй дуги. */) {
        ...
    } else if ...
}
```

Шаблон для интерфейса класса данных

Интерфейс для класса данных включает в себя следующие части:

- объявление конструктора;
- объявление деструктора;
- указатели на объекты классов состояний;
- определение внутренних переменных;
- объявление функций реализующих события.

```
class MainData {

public:
    // Конструктор.
    MainData();

    // Деструктор.
    virtual ~MainData();

    ////////////////////////////////////////////////////////
    // Указатели на объекты, соответствующие классам состояниям.
    S *pS0;
    ...
    S *pSL;

    ////////////////////////////////////////////////////////
    // Внутренние переменные.
    ...

    ////////////////////////////////////////////////////////
    // Функции, реализующие события.
    ...
};
```

Шаблон для реализации класса данных

Реализация для класса данных включает в себя следующие части:

- определение конструктора;
- определение деструктора;
- определение функций, реализующих события.

```
// Конструктор.
MainData::MainData() {

    // Инициализация автомата.
    ...

    // Создание объектов состояний.
    pS0 = new S0(this);
    ...
    pSL = new SL(this);
}

// Деструктор.
MainData::~MainData() {

    // Уничтожение объектов состояний.
    delete pS0;
    ...
    delete pSL;
}

//////////////////////////////////////////////////////////
// Функции, реализующие события.
...
```

Листинг 10. Интерфейсы для автомата *A0*

```
//////////////////////////////////////////////////////////
// 5.h

// Максимальный размер строки, которая будет использоваться
// в качестве входного потока (INPUT_STREAM_SIZE).
#define INPUT_STREAM_SIZE 16384

// Объявление класса данных автомата A0, содержащего внутренние переменные
// и указатели на объекты, соответствующие классам состояний.
class MainData;

// Базовый класс для состояний автомата A0
class S {

protected:

    // Указатель на объект данных, содержащий внутренние переменные и
    // объекты, соответствующие классам состояний автомата A0.
    MainData *pMainData;

    //////////////////////////////////////////////////////
    // Входные переменные

    // Возвращает true, если текущий символ (curSymbol) -
    // буква латинского алфавита и
    // false в противном случае.
    bool x1();

    // Возвращает true, если текущий символ (curSymbol) -
```

```
    // символ перевода строки '\n' и
    // false в противном случае.
    bool x2();

    ////////////////////////////////////////////////////////
    // Выходные воздействия

    // Записывает текущий символ (curSymbol) в выходной поток.
    void z1_0();

    // Устанавливает счетчик символов в слове (symbolCount) в ноль.
    void z2_0();

    // Увеличивает значение счетчика символов в
    // слове (symbolCount) на единицу.
    void z2_1();

    // Записывает в выходной поток:
    // -<значение счетчика символов в слове (symbolCount)>
    // <символ перевода строки '\n'>
    void z1_1();

public:
    // Конструктор и деструктор
    S(MainData *pMainData);
    virtual ~S() {}

    // Метод, реализующий действия для состояния.
    virtual S* execute()=0;
};

// Класс для состояний 0 ("Ожидание начала слова") автомата A0.
class S0 : public S {
public:
    //Конструктор
    S0(MainData *pMainData);

    // Метод, реализующий действия для состояния.
    virtual S* execute();
};

// Класс для состояний 1 ("Обработка слова") автомата A0.
class S1 : public S {
public:
    //Конструктор
    S1(MainData *pMainData);

    // Метод, реализующий действия для состояния.
    virtual S* execute();
};

// Класс для состояний 2 ("Работа завершена") автомата A0.
class S2: public S {
public:
    //Конструктор
    S2(MainData *pMainData);

    // Метод, реализующий действия для состояния.
    virtual S* execute();
};

// Определение класса данных содержащего внутренне переменные и
// объекты, соответствующие классам состояниям, автомата A0.
class MainData {
```

```
public:
    // Конструктор
    MainData();

    // Деструктор
    virtual ~MainData();

    ////////////////////////////////////////////////////////
    // Указатели на объекты, соответствующие классам состояниям

    // Указатель на объект класса для состояния 0
    S *pS0;

    // Указатель на объект класса для состояния 1 ("Обработка слова").
    S *pS1;

    // Указатель на объект класса для состояния 2 ("Работа завершена").
    S *pS2;

    ////////////////////////////////////////////////////////
    // Внутренние переменные

    // Текущий символ
    char curSymbol;

    // Счетчик символов в слове
    int symbolCount;

    // Строка, которая будет использоваться в
    // качестве входного потока
    char inputStream[16384];

    // Указатель на текущий символ в inputStream.
    char *curInputChar;

    ////////////////////////////////////////////////////////
    // Функции, реализующие события

    // Событие - приход нового символа.
    // Обновляется текущий символ.
    // Перед первым вызов нужно вызнать initInputStream().
    void e0();

    // Инициализация входного потока.
    // Необходимо вызвать перед первым вызовом e0().
    void initInputStream();
};
```

Реализуем пример с помощью данного метода (Листинг 9).

Листинг 11. Реализация автомата *A0*

```
////////////////////////////////////////////////////////
// 5.cpp

#include <stdio.h>

#include "5.h"

////////////////////////////////////////////////////////
// Определение методов класса S.

////////////////////////////////////////////////////////
```

```
// Входные переменные.

// Возвращает true, если текущий символ (curSymbol) -
// буква латинского алфавита и
// false в противном случае.
bool S::x1() {

      return (pMainData->curSymbol >= 'a' && pMainData->curSymbol <= 'z') ||
               (pMainData->curSymbol >= 'A' && pMainData->curSymbol <= 'Z');
}

// Возвращает true, если текущий символ (curSymbol) -
// символ перевода строки '\n' и
// false в противном случае.
bool S::x2() {

      return pMainData->curSymbol == '\n';
}

//////////////////////////////////////////////////////////
// Выходные воздействия.

// Записывает текущий символ (curSymbol) в выходной поток.
void S::z1_0() {

      printf("%c", pMainData->curSymbol);
}

// Устанавливает счетчик символов в слове (symbolCount) в ноль.
void S::z2_0() {

    pMainData->symbolCount =  0;
}

// Увеличивает значение счетчика символов в
// слове (symbolCount) на единицу.
void S::z2_1() {

    pMainData->symbolCount++;
}

// Записывает в выходной поток:
// -<значение счетчика символов в слове (symbolCount)>
// <символ перевода строки '\n'>
void S::z1_1() {

    printf(" - %d\n", pMainData->symbolCount);
}

// Конструктор.
S::S(MainData *pMainData):pMainData(pMainData){

}

//////////////////////////////////////////////////////////
// Определение методов класса S0.

// Конструктор.
S0::S0(MainData *pMainData):S(pMainData) {

}

// Метод, реализующий действия для состояния.
S* S0::execute() {
```

```
    pMainData->e0();

    if(x1()) {
        z1_0();
        z2_0();
        z2_1();
        return pMainData->pS1;
    } else if (x2()) {
        return pMainData->pS2;
    } else {
        return pMainData->pS0;
    }
}

////////////////////////////////////////////////////////
// Определение методов класса S1.

// Конструктор.
S1::S1(MainData *pMainData):S(pMainData) {

}

// Метод, реализующий действия для состояния.
S* S1::execute() {

   pMainData->e0();

   if(x1()) {
        z1_0();
        z2_1();
        return pMainData->pS1;
    } else if(x2()) {
        z1_1();
        return pMainData->pS2;
    } else {
        z1_1();
        return pMainData->pS0;
    }
}

////////////////////////////////////////////////////////
// Определение методов класса S2.

// Конструктор.
S2::S2(MainData *pMainData):S(pMainData) {

}

// Метод, реализующий действия для состояния.
S* S2::execute() {

    return pMainData->pS2;
}

////////////////////////////////////////////////////////
// Определение методов класса MainData.

// Конструктор.
MainData::MainData() {

    // Инициализация автомата.
    initInputStream();
```

```
    // Создание объектов состояний.
    pS0 = new S0(this);
    pS1 = new S1(this);
    pS2 = new S2(this);
}

// Деструктор.
MainData::~MainData() {

    // Уничтожение объектов состояний.
    delete pS0;
    delete pS1;
    delete pS2;
}

//////////////////////////////////////////////////////
// Функции, реализующие события.

// Событие - приход нового символа.
// Обновляется текущий символ.
// Перед первым вызов нужно вызнать initInputStream().
void MainData::e0() {

    if(curInputChar == INPUT_STREAM_SIZE + inputStream) {
        curSymbol = '\n';
        return;
    }

    curSymbol = *curInputChar++;
}

// Инициализация входного потока.
// Необходимо вызвать перед первым вызовом e0().
void MainData::initInputStream() {

    curInputChar = inputStream;

    while ((*curInputChar++ = getchar()) != '\n') {
        if(curInputChar == INPUT_STREAM_SIZE + inputStream)
            break;
    }

    curInputChar = inputStream;
}

// Главная функция программы.
int main() {
    // Создание внутренних переменных и состояний автомата.
    MainData mainData;

    // Устанавливаем начальное состояние.
    S *pS = mainData.pS0;

    // До тех пор, пока автомат A0 не перейдет
    // в состояние 2 ("Работа завершена").
    while(pS != mainData.pS2)
    {
        pS = pS->execute();
    }

    printf("Press \"Enter\" to exit.");
    while(getchar() != 10);

    return 0;
```

```
}
```

Заключение

В работе предложены пять шаблонов для реализации автоматов, позволяющих формально преобразовать графы переходов автоматов в код программы. Поэтому достаточно просто осуществить и обратное преобразование. Предложенные шаблоны можно использовать для автоматизации указанного процесса.

Как отмечалось выше, метод реализации автомата с помощью рекурсивной функции является учебным и заимствованным из работы [1] для полноты цитирования.

Исполняемые программы и исходные тексты опубликованы по адресу http://is.ifmo.ru/projects/length/.

Литература

1. *Непейвода Н.Н., Скопин И.Н.* Основания программирования. Москва-Ижевск: Институт компьютерных исследований, 2003. http://ulm.udsu.ru/~nnn/index.html
2. *Непейвода Н.Н.* Стили и методы программирования. М.: Интернет-Университет Информационных Технологий, 2005.
3. *Шалыто А.А.* SWITCH-технология. Алгоритмизация программирования задач логического управления. СПб.: Наука, 1998.
4. *Шалыто А.А, Туккель Н.И.* Программирование с явным выделением состояний // Мир ПК, 2001. № 8, С.116-121; № 9, С.132-138. http://is.ifmo.ru/works/mirk/

ОБ АВТОРАХ

Павел Геннадьевич Лобанов – магистрант кафедры «Компьютерные технологии» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СПбГУ ИТМО). E-mail: plobanov@rain.ifmo.ru.

Анатолий Абрамович Шалыто – докт. техн. наук, профессор, заведующий кафедрой «Технологии программирования» СПбГУ ИТМО. E-mail: shalyto@mail.ifmo.ru