

Автоматное программирование

Материал из Википедии — свободной энциклопедии

Автомáтное программи́рование — это парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. Известна также и другая "парадигма автоматного программирования, состоящая в представлении сущностей со сложным поведением в виде автоматизированных объектов управления, каждый из которых представляет собой объект управления и автомат". При этом о программе, как в автоматическом управлении, предлагается думать как о системе автоматизированных объектов управления.

В зависимости от конкретной задачи в автоматном программировании могут использоваться как конечные автоматы, так и автоматы с более сложным строением.

Определяющими для автоматного программирования являются следующие особенности:

- временной период выполнения программы разбивается на *шаги автомата*, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний
- передача информации между шагами автомата осуществляется только через явно обозначенное множество переменных, называемых *состоянием автомата*; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.; иначе говоря, состояние программы на любые два момента входа в *шаг автомата* могут различаться между собой только значениями переменных, составляющих *состояние автомата* (причём такие переменные должны быть явно обозначены в качестве таковых).

Полностью выполнение кода в автоматном стиле представляет собой цикл (возможно, неявный) шагов автомата.

Название **автоматное программирование** оправдывается ещё и тем, что стиль мышления (восприятия процесса исполнения) при программировании в этой технике практически точно воспроизводит стиль мышления при составлении формальных автоматов (таких как машина Тьюринга, автомат Маркова и др.)

Содержание

Пример с использованием конечного автомата

Императивная программа

Программа в автоматном стиле

Выделение шага автомата в отдельную функцию

Программа с явно заданной таблицей переходов

Использование объектно-ориентированных возможностей

[Сфера применения](#)

[История](#)

[Сравнение с императивным и процедурным программированием](#)

[Связь с объектно-ориентированным программированием](#)

[Специализированные языки программирования](#)

[Примечания](#)

[Литература](#)

[Ссылки](#)

[См. также](#)

Пример с использованием конечного автомата

Пусть, к примеру, требуется написать на языке Си программу, читающую из потока стандартного ввода текст, состоящий из строк, и для каждой строки печатающую первое слово этой строки и перевод строки. Ясно, что для этого во время чтения каждой строки следует сначала пропустить пробелы, если таковые есть в начале строки; затем читать буквы, составляющие слово, и печатать их, пока слово не кончится (то есть либо не кончится строка, либо не будет встречен пробельный символ); наконец, когда первое слово успешно считано и напечатано, необходимо дочитать строку до конца, ничего при этом не печатая. Встретив (на любой фазе) символ перевода строки, следует напечатать перевод строки и продолжить с начала. При возникновении (опять таки, на любой фазе) ситуации «конец файла» следует прекратить работу.

Императивная программа

Программа, решающая эту задачу в традиционном императивном стиле, может выглядеть, например, так (язык C):

```
#include <stdio.h>

int main() {
    int c;

    do {
        c = getchar();
        while (c == ' ') c = getchar();
        while (c != ' ' && c != '\n' && c != EOF) putchar(c), c = getchar();
        putchar('\n');
        while (c != '\n' && c != EOF) c = getchar();
    } while (c != EOF);

    return 0;
}
```

Программа в автоматном стиле

Ту же задачу можно решить, применив мышление в терминах конечных автоматов. Заметим, что разбор строки разделяется на три фазы: пропуск лидирующих пробелов, печать слова и пропуск символов остатка строки. Назовём эти три фазы **состояниями** before, inside и after. Программа теперь может выглядеть, например, так:

```

#include <stdio.h>

int main() {
    enum states { before, inside, after } state;
    int c;

    state = before;
    while ((c = getchar()) != EOF) {
        switch (state) {

            case before:
                if (c == '\n') putchar('\n');
                else if (c != ' ') putchar(c), state = inside;
                break;

            case inside:
                switch (c) {
                    case ' ':
                        state = after; break;
                    case '\n':
                        putchar('\n'), state = before;
                        break;
                    default: putchar(c);
                }
                break;

            case after:
                if (c == '\n') putchar('\n'), state = before;

        }
    }

    return 0;
}

```

или так:

```

#include <stdio.h>

void (*state)(int);

void before(int c);
void inside(int c);
void after(int c);

void before(int c) {
    if (c == '\n') putchar('\n');
    else if (c != ' ') putchar(c), state = inside;
}

void inside(int c) {
    switch (c) {
        case ' ':
            state = after; break;
        case '\n':
            putchar('\n'), state = before;
            break;
        default: putchar(c);
    }
}

void after(int c) {
    if (c == '\n') putchar('\n'), state = before;
}

int main() {
    int c;

    state = before;
    while ((c = getchar()) != EOF) {
        state(c);
    }
}

```

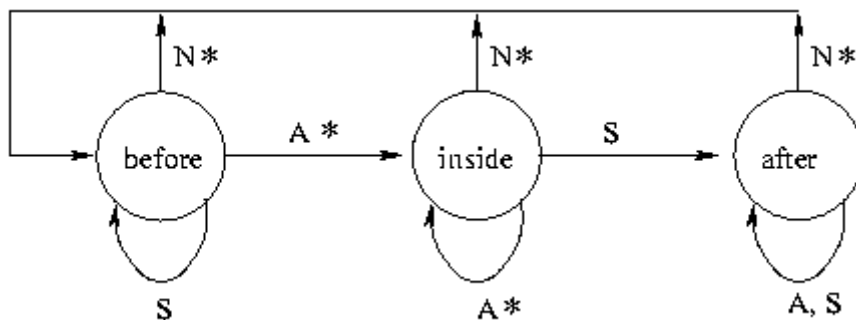
```

    return 0;
}

```

Несмотря на то, что код явно стал более длинным, у него имеется одно несомненное достоинство: чтение (то есть вызов функции `getchar()`) теперь выполняется **ровно в одном месте**. Кроме того, необходимо заметить, что вместо четырёх циклов, использовавшихся в предыдущей версии, цикл теперь используется только один. Тело цикла (за исключением действий, выполняемых в заголовке) представляет собой **шаг автомата**, сам же цикл задаёт **цикл работы автомата**.

Программа реализует (моделирует) работу конечного автомата, изображённого на рисунке. Буквой **N** на диаграмме обозначен символ конца строки, буквой **S** — символ пробела, буквой **A** — все остальные символы. За один шаг автомат делает ровно один переход в зависимости от текущего



состояния и прочитанного символа. Некоторые переходы сопровождаются печатью прочитанного символа; такие переходы на диаграмме обозначены звёздочками.

Строго соблюдать разделение кода на обработчики отдельных состояний, вообще говоря, не обязательно. Более того, в некоторых случаях само понятие состояния может складываться из значений нескольких переменных, так что учесть все возможные их комбинации окажется практически невозможно. В рассматриваемом примере можно сэкономить объём кода, если заметить, что действия, выполняемые по символу «конец строки», от состояния не зависят. Программа, эквивалентная предыдущей, но написанная с учётом такого замечания, будет выглядеть так:

```

#include <stdio.h>

int main() {
    enum states { before, inside, after } state;
    int c;

    state = before;
    while ((c = getchar()) != EOF) {
        if (c == '\n') putchar('\n'), state = before, continue;
        switch (state) {
            case before:
                if (c != ' ') putchar(c), state = inside;
                break;
            case inside:
                if (c == ' ') state = after;
                else putchar(c);
            case after:
                break;
        }
    }

    return 0;
}

```

Выделение шага автомата в отдельную функцию

Основополагающим в вышеприведённой программе является по-прежнему чёткое выделение кода, отвечающего за шаг автомата. Это обстоятельство можно подчеркнуть ещё сильнее, если выделить шаг автомата в отдельную функцию.

```
#include <stdio.h>

enum States { before, inside, after };

void step(enum States *state, int *c)
{
    if (*state == before)
    {
        if (*c == '\n')
            putchar('\n');
        else if (*c != ' ')
            *state = inside;
    }
    if (*state == inside)
    {
        if (*c == ' ')
            *state = after;
        else if (*c == '\n')
        {
            putchar('\n');
            *state = before;
        }
        else
            putchar(*c);
    }
    if (*state == after)
    {
        if (*c == '\n')
        {
            putchar('\n');
            *state = before;
        }
    }
}

int main() {
    int c; enum States state = before;

    while ((c = getchar()) != EOF) step(&state, &c);

    return 0;
}
```

Этот пример наглядно демонстрирует основное свойство, благодаря которому код можно считать оформленным в стиле автоматного программирования:

1. отдельные шаги автомата выполняются в неперекрывающиеся временные периоды
2. единственным средством передачи информации между шагами является явно определённое состояние (в данном случае переменная `state`)

Программа с явно заданной таблицей переходов

Конечный автомат, как известно, может быть задан и таблицей переходов. Вообще говоря, код программы, моделирующей конечный автомат, вполне может отражать и это свойство автомата. В следующей программе массив `the_table` задаёт таблицу переходов. Строки таблицы соответствуют трём состояниям автомата, столбцы — читаемым символам (первый столбец — пробел, второй столбец — перевод строки, третий столбец — все остальные символы). Каждая ячейка таблицы содержит номер нового состояния и признак необходимости печати символа (в приведённом коде используются битовые поля для экономии памяти). Конечно, в реальной задаче

могла бы потребоваться гораздо более сложная структура таблицы, содержащая, например, указатели на функции для выполнения каких-либо действий, связанных с переходами, но в рассматриваемом примере это не нужно:

```
#include <stdio.h>

enum states { before = 0, inside = 1, after = 2 };

typedef struct branch {
    enum states new_state:4;
    int should_putchar:4;
} branch;

branch the_table[3][3] = {
    /*      ' '      '\n'      others */
    /* before */ { {before, 0}, {before, 1}, {inside, 1} },
    /* inside */ { {after, 0}, {before, 1}, {inside, 1} },
    /* after  */ { {after, 0}, {before, 1}, {after, 0} }
};

void step(enum states *state, int c) {
    int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
    branch *b = & the_table[*state][idx2];

    *state = b->new_state;
    if (b->should_putchar) putchar(c);
}

int main() {
    int c; enum states state = before;

    while ((c = getchar()) != EOF) step(&state, c);

    return 0;
}
```

Использование объектно-ориентированных возможностей

Если используемый язык программирования поддерживает объектно-ориентированные возможности, логично будет инкапсулировать конечный автомат в объект, скрыв детали реализации. Например, аналогичная программа на языке C++ может выглядеть так:

```
#include <stdio.h>
class StateMachine {
    enum states { before = 0, inside = 1, after = 2 } state;
    struct branch {
        enum states new_state:4;
        unsigned should_putchar:4;
    };
    static struct branch the_table[3][3];
public:
    StateMachine() : state(before) {}
    void FeedChar(int c) {
        int idx2 = (c == ' ') ? 0 : (c == '\n') ? 1 : 2;
        struct branch *b = & the_table[state][idx2];
        state = b->new_state;
        if(b->should_putchar) putchar(c);
    }
};

struct StateMachine::branch StateMachine::the_table[3][3] = {
    /*      ' '      '\n'      others */
    /* before */ { {before, 0}, {before, 1}, {inside, 1} },
    /* inside */ { {after, 0}, {before, 1}, {inside, 1} },
    /* after  */ { {after, 0}, {before, 1}, {after, 0} }
};

int main()
```

```
{  
    int c;  
    StateMachine machine;  
    while((c = getchar()) != EOF)  
        machine.FeedChar(c);  
    return 0;  
}
```

Отметим, что в этом примере мы использовали для ввода-вывода библиотеку языка Си, чтобы избежать появления «лишних» (отвлекающих внимание) изменений в сравнении с предыдущим примером.

Сфера применения

Автоматное программирование широко применяется при построении лексических анализаторов (классические конечные автоматы) и синтаксических анализаторов (автоматы с магазинной памятью)^[1].

Кроме того, мышление в терминах конечных автоматов (то есть разбиение исполнения программы на *шаги автомата* и передача информации от шага к шагу через состояние) необходимо при построении событийно-ориентированных приложений. В этом случае программирование в стиле конечных автоматов оказывается единственной альтернативой порождению множества процессов или потоков управления (тредов).

Часто понятие состояний и машин состояний используется для спецификации программ. Так, при проектировании программного обеспечения с помощью UML для описания поведения объектов используются диаграммы состояний (state machine diagrams). Кроме того, явное выделение состояний используется в описании сетевых протоколов (см., например, RFC 793^[2]).

Мышление в терминах автоматов (шагов и состояний) находит применение и при описании семантики некоторых языков программирования. Так, исполнение программы на языке Рефал представляет собой последовательность изменений поля зрения Рефал-машины или, иначе говоря, последовательность шагов Рефал-автомата, состоянием которого является *содержимое поля зрения* (произвольное Рефал-выражение, не содержащее переменных).

Механизм продолжений языка Scheme для своей реализации также требует мышления в терминах состояний и шагов, несмотря на то что сам язык Scheme никоим образом не является автоматным. Тем не менее, чтобы обеспечить возможность «замораживания» *продолжения*, приходится при реализации вычислительной модели языка Scheme объединять все компоненты среды исполнения, включая *список действий, которые осталось выполнить для окончания вычислений*, в единое целое, которое также обычно называется *продолжением*. Такое *продолжение* оказывается состоянием автомата, а процесс выполнения программы состоит из шагов, каждый из которых выводит следующее значение *продолжения* из предыдущего.

Александр Оллонгрэн в своей книге^[3] описывает так называемый *Венский метод* описания семантики языков программирования, основанный целиком на формальных автоматах.

В качестве одного из примеров применения автоматной парадигмы можно назвать систему STAT ^[1] (<https://web.archive.org/web/20080202163122/http://www.cs.ucsb.edu/~seclab/projects/stat/index.html>); эта система, в частности, включает встроенный язык STATL, имеющий чисто автоматную семантику.

Существуют также предложения по использованию автоматного программирования в качестве универсального подхода к созданию компьютерных программ вне зависимости от предметной области. Так, авторы статьи^[4] утверждают, что автоматное программирование способно сыграть

роль легендарной серебряной пули.

История

Наиболее ранние случаи применения парадигмы автоматного программирования относятся, по-видимому, к предметным областям, в которых наработана алгоритмическая теория, основанная на теории автоматов, и прежде всего — к анализу текстов на формальных языках.^[1] В качестве одной из наиболее ранних работ на эту тему можно назвать статью.^[5]

Одним из первых упоминаний использования техники автоматного программирования независимо от теоретических наработок, основанных на конечных автоматах, является статья Питера Наупа.^[6] В этой статье автор называет применённый подход «подходом машины Тьюринга» (*Turing machine approach*), но реально никакая машина Тьюринга в статье не строится; приведённый в статье подход удовлетворяет вышеприведённому определению **автоматного программирования**.

Сравнение с императивным и процедурным программированием

Понятие **состояния программы** не является эксклюзивной особенностью автоматного программирования. Вообще говоря, **состояние** возникает при выполнении любой компьютерной программы и представляет собой совокупность всей информации, которая во время исполнения может изменяться. Так, **состояние** программы, выполненной в традиционном императивном стиле состоит из

1. совокупности значений всех глобальных переменных и содержимого динамической памяти
2. содержимого регистров общего назначения
3. содержимого стека (включая адреса возвратов и значения локальных переменных)
4. текущего значения счётчика команд (то есть текущей позиции в коде программы)

Составные части состояния можно разделить на **явные** (такие как значения переменных) и **неявные** (адреса возвратов и значение счётчика команд).

В контексте введённых определений программу, оформленную в виде модели конечного автомата, можно считать частным случаем императивной программы, таким, в котором роль неявной составляющей состояния сведена к минимуму. Если рассмотреть автоматную программу в моменты начала очередного шага автомата, то состояния программы в эти моменты будут различаться только явной составляющей. Это обстоятельство существенно упрощает анализ свойств программы.

Связь с объектно-ориентированным программированием

В теории объектно-ориентированного программирования считается, что **объект** имеет **внутреннее состояние** и способен получать **сообщения**, отвечать на них, отправлять сообщения другим объектам и в процессе обработки сообщений изменять своё внутреннее состояние. Более приближенное к практике понятие *вызова метода объекта* считается синонимом понятия *отправки сообщения объекту*.

Таким образом, с одной стороны, объекты в объектно-ориентированном программировании могут рассматриваться как конечные автоматы (или, если угодно, модели конечных автоматов), состояние которых представляет собой совокупность внутренних полей, в качестве же шага автомата могут

рассматриваться один или несколько методов объекта при условии, что эти методы не вызывают ни сами себя, ни друг друга ни прямо, ни косвенно.

С другой стороны, очевидно, что понятие объекта представляет собой удачный инструмент реализации модели конечного автомата. При применении парадигмы автоматного программирования в объектно-ориентированных языках обычно модели автоматов представляются в виде классов, состояние автомата описывается внутренними (закрытыми) полями класса, а код шага автомата оформляется в виде метода класса, причём такой метод скорее всего оказывается единственным открытым методом (не считая конструкторов и деструкторов), изменяющим состояние автомата. Другие открытые методы могут служить для получения информации о состоянии автомата, но не меняют его. Все вспомогательные методы (например, методы-обработчики отдельных состояний или их категорий) в таких случаях обычно убирают в закрытую часть класса.

Специализированные языки программирования

- Язык последовательных функциональных схем SFC (Sequential Function Chart) — графический язык программирования широко используется для программирования промышленных логических контроллеров PLC.


В SFC программа описывается в виде схематической последовательности шагов, объединённых переходами.

- Дракон-схемы — графический язык программирования, используется для программирования в ракетно-космической технике («Буря», «Морской старт», «Тополь»). Существует бесплатный Дракон-редактор.
- Язык Рефлекс (<http://reflex-language.narod.ru/>) — Си-подобный язык программирования, ориентированный на описание сложных алгоритмов управления в задачах промышленной автоматизации.



Примечания

1. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции = The theory of parsing, translation and compiling. — М.: МИР, 1978. — Т. 1. — 612 с.
2. Postel, J., ed., Transmission Control Protocol, RFC 793
3. А. Оллонгрен. Определение языков программирования интерпретирующими автоматами = Definition of programming languages by interpreting automata. — М.: МИР, 1977. — 288 с.
4. Туккель Н.И., Шалыто А.А. Программирование с явным выделением состояний (<http://is.ifmo.ru/works/mirk/>) // Мир ПК. — 2001. — № 9. — С. 132—138.
5. Johnson, W. L.; Porter, J. H.; Ackley, S. I.; Ross, D. T. Automatic generation of efficient lexical processors using finite state techniques (англ.) // Comm. ACM. — 1968. — Т. 11, № 12. — С. 805—813.
6. Naur, Peter. The design of the GIER ALGOL compiler Part II (англ.) // BIT Numerical Mathematics. — 1963. — Сентябрь (т. 3). — С. 145—166. — ISSN 0006-3835 (<https://www.worldcat.org/search?fq=x0:jrnl&q=n2:0006-3835>). — doi:10.1007/BF01939983 (<https://dx.doi.org/10.1007%2FBF01939983>).

Литература

- Поликарпова Н.И., Шалыто А.А. Автоматное программирование (<http://project.ifmo.ru/books/2/>) (http://is.ifmo.ru/books/_book.pdf) . — СПб.: Питер, 2009. — 176 с. — ISBN 978-5-

388-00692-9.

- *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. (<http://project.ifmo.ru/books/1>). — СПб.: Наука, 1998. — 628 с.
- *Practical UML Statecharts in C/C++* (https://www.amazon.com/Practical-UML-Statecharts-Second-Event-Driven/dp/0750687061/ref=ntt_at_ep_dpi_1) Книга о реализации UML2 State Machine на C/C++. Нацелена, главным образом, на программирование встроенных систем реального времени.
- Научно-технический вестник СПбГУ ИТМО. Выпуск 53. Автоматное программирование. 2008. <http://ntv.ifmo.ru/file/journal/61.pdf> 
- *Зюбин В. Е.* Программирование информационно-управляющих систем на основе конечных автоматов: учебное пособие. (<http://softcraft.ru/auto/etc/mpz/index.shtml>). — Новосибирск: Изд-во: Новосиб. гос. ун-т, 2006. — 96 с. — ISBN 5-94356-425-X.
- *Ненейвода Н.Н.* Стили и методы программирования. курс лекций. учебное пособие. — М.: Интернет-университет информационных технологий, 2005. — С. 145—212. — 316 с. — ISBN 5-9556-0023-X.
- *Harel, David.* Statecharts: A Visual Formalism for Complex Systems (<http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf>)  (англ.) // Sci. Comput. Programming : journal. — 1987. — No. 8. — P. 231—274.
- *Harel, David; Drusinsky, D.* Using Statecharts for Hardware Description and Synthesis (англ.) // IEEE Trans. Computer Aided Design of Integrated Circuits and Systems : journal. — 1989. — No. 8. — P. 798—807.

Ссылки

- «Автоматное программирование. Проекты» (архив проектов с открытой документацией) (<https://web.archive.org/web/20091227145751/http://project.ifmo.ru/>)
- Б. П. Кузнецов. «Психология автоматного программирования» (<http://www.softcraft.ru/design/ap>)
- Б. П. Кузнецов. «Настраиваемые автоматные программы» (<http://www.softcraft.ru/auto/other/itst>)

См. также

- Событийно-ориентированное программирование

Источник — https://ru.wikipedia.org/w/index.php?title=Автоматное_программирование&oldid=110183620

Эта страница в последний раз была отредактирована 29 октября 2020 в 18:44.

Текст доступен по лицензии Creative Commons Attribution-ShareAlike; в отдельных случаях могут действовать дополнительные условия.

Wikipedia® — зарегистрированный товарный знак некоммерческой организации Wikimedia Foundation, Inc.