Automata-based programming

From Wikipedia, the free encyclopedia
(Redirected from Automata-Based Programming)

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other (often more complicated) formal automata (see automata theory). Sometimes a potentially-infinite set of possible states is introduced, and such a set can have a complicated structure, not just an enumeration.

FSM-based programming
is generally the same, but, formally speaking, doesn't cover all possible variants as FSM stands for finite state machine and **automata-based programming** doesn't necessarily employ FSMs in the strict sense.

The following properties are key indicators for automata-based programming:

1. The time period of the program's execution is clearly separated down to the *steps of the automaton*. Each of the *steps* is effectively an execution of a code section (same for all the steps), which has a single entry point. Such a section can be a function or other routine, or just a cycle body. The step section might be divided down to subsection to be executed depending on different states, although this is not necessary.
2. Any communication between the steps is only possible via the explicitly noted set of variables named *the state*. Between any two steps, the program (or its part created using the automata-based technique) can **not** have implicit components of its state, such as local (stack) variables' values, return addresses, the current instruction pointer etc. That is, the state of the whole program, taken at any two moments of entering the step of the automaton, can only differ in the values of the variables being considered as the state of the automaton.

The whole execution of the automata-based code is a (possibly explicit) cycle of the automaton's steps.

Another reason to use the notion of **automata-based programming** is that the programmer's style of thinking about the program in this technique is very similar to the style of thinking used to solve maths-related tasks using Turing machine, Markov algorithm etc.

Contents

Example

Consider we need a program in C that reads a text from *standard input stream*, line by line, and prints the first word of each line. It is clear we need first to read and skip the leading spaces, if any, then read characters of the first word and print them until the word ends, and then read and skip all the remaining characters until the end-of-line character is encountered. Upon the end of line character (regardless of the stage) we restart the algorithm from the beginning, and in case the *end of file* condition (regardless of the stage) we terminate the program.

Traditional (imperative) program

The program which solves the example task in traditional (imperative) style can look something like this:

```
#include <stdio.h>
int main()
{
    int c;
    do {
        c = getchar();
        while(c == ' ')
            c = getchar();
        while(c != EOF && c != ' ' && c != '\n') {
            putchar(c);
            c = getchar();
        }
        putchar('\n');
        while(c != EOF && c != '\n')
            c = getchar();
    } while(c != EOF);
    return 0;
}
```

Automata-based style program

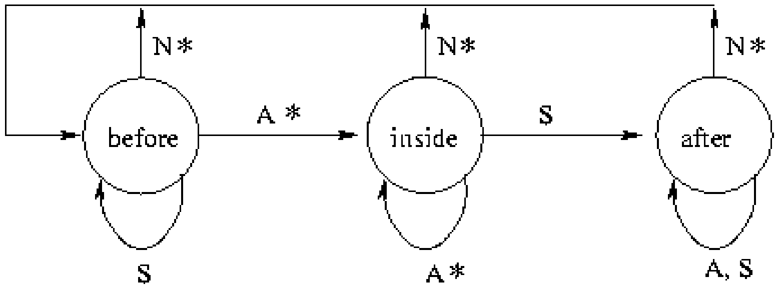
The same task can be solved thinking in terms of finite state machines. Please note that line parsing has three stages: skipping the leading spaces, printing the word and skipping the trailing characters. Let's call them **states** `before`, `inside` and `after`. The program may now look like this:

```
#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;
}
```

Although the code now looks longer, it has at least one significant advantage: there's only one *reading* (that is, call to the `getchar()` function) instruction in the program. Besides that, there's only one `while` loop instead of the four the previous version had.

In this program, the body of the `while` loop is the **automaton step**, and the loop itself is the *cycle of the automaton's work*.

The program implements (models) the work of a *finite state machine* shown on the picture. The **N** denotes the end of line character, the **S** denotes spaces, and the **A** stands for all the other characters. The automaton follows exactly one *arrow* on each step depending on the current state and the encountered character. Some state switches are accompanied with printing the character; such arrows are marked with asterisks.



It is not absolutely necessary to divide the code down to separate handlers for each unique state. Furthermore, in some cases the very notion of the *state* can be composed of several variables' values, so that it could be impossible to handle each possible state explicitly. In the discussed program it is possible to reduce the code length noticing that the actions taken in response to the end of line character are the same for all the possible states. The following program is equal to the previous one but is a bit shorter:

```
#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;
        } else
        switch(state) {
            case before:
                if(c != ' ') {
                    putchar(c);
                    state = inside;
                }
                break;
            case inside:
                if(c == ' ') {
                    state = after;
                } else {
                    putchar(c);
                }
                break;
            case after:
                break;
        }
    }
    return 0;
}
```

A separate function for the automaton step

The most important property of the previous program is that the automaton step code section is clearly localized. It is possible to demonstrate this property even better if we provide a separate function for it:

```c
#include <stdio.h>
enum states { before, inside, after };
void step(enum states *state, int c)
{
    if(c == '\n') {
        putchar('\n');
        *state = before;
    } else
    switch(*state) {
        case before:
            if(c != ' ') {
                putchar(c);
                *state = inside;
            }
            break;
        case inside:
            if(c == ' ') {
                *state = after;
            } else {
                putchar(c);
            }
            break;
        case after:
            break;
    }
}
int main()
{
    int c;
    enum states state = before;
    while((c = getchar()) != EOF) {
        step(&state, c);
    }
    return 0;
}
```

This example clearly demonstrates the basic properties of automata-based code:

1. time periods of automaton step executions may not overlap
2. the only information passed from the previous step to the next is the explicitly specified *automaton state*

Explicit state transition table

A finite automaton can be defined by an explicit state transition table. Generally speaking, an automata-based program code can naturally reflect this approach. In the program below there's an array named `the_table`, which defines the table. The rows of the table stand for three *states*, while columns reflect the input characters (first for spaces, second for the end of line character, and the last is for all the other characters).

For every possible combination, the table contains the new state number and the flag, which determines whether the automaton must print the symbol. In a real life task, this could be more complicated; e.g., the table could contain pointers to functions to be called on every possible combination of conditions.

```
#include <stdio.h>
enum states { before = 0, inside = 1, after = 2 };
struct branch {
    enum states new_state:2;
    int should_putchar:1;
};
struct 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;
    struct 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;
}
```

Using object-oriented capabilities

If the implementation language supports object-oriented programming, it is reasonable to encapsulate the automaton into an object, thus hiding implementation details from the outer program. for example, the same program in C++ can look like this:

```
#include <stdio.h>
class StateMachine {
    enum states { before = 0, inside = 1, after = 2 } state;
    struct branch {
        enum states new_state:2;
        int should_putchar:1;
    };
    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;
}
```

Please note we have intentionally used the input/output functions from the standard library of C to minimize changes not directly related to the subject of the article.

Applications

Automata-based programming is widely used in lexical and syntactic analyses.[1]

Besides that, thinking in terms of automata (that is, breaking the execution process down to *automaton steps* and passing information from step to step through the explicit *state*) is necessary for event-driven programming as the only alternative to using parallel processes or threads.

The notions of states and state machines are often used in the field of formal specification. For instance, UML-based software architecture development uses state diagrams to specify the behaviour of the program. Also various communication protocols are often specified using the explicit notion of *state* (see, e.g., RFC 793[2]).

Thinking in terms of automata (steps and states) can also be used to describe semantics of some programming languages. For example, the execution of a programm written in the Refal language is described as a sequence of *steps* of a so-called abstract Refal machine; the state of the machine is a *view* (an arbitrary Refal expression without variables).

Continuations in the Scheme
language require thinking in terms of steps and states, although Scheme itself is in no way automata-related (it is recursive). To make it possible the call/cc
feature to work, implementation needs to be able to catch a whole state of the executing program, which is only possible when there's no implicit part in the state. Such a *caught state* is the very thing called *continuation*, and it can be considered as the *state*
of a (relatively complicated) automaton. The step of the automaton is deducing the next continuation from the previous one, and the execution process is the cycle of such steps.

Alexander Ollongren in his book[3] explains the so-called *Vienna method* of programming languages semantics description which is fully based on formal automata.

The STAT system [1] (http://www.cs.ucsb.edu/~seclab/projects/stat/index.html) is a good example of using the automata-based approach; this system, besides other features, includes an embedded language called *STATL* which is purely automata-oriented.

History

Automata-based techniques were used widely in the domains where there are algorithms based on automata theory, such as formal language analyses.[1]

One of the early papers on this is by Johnson et al, 1968.[4]

One of the earliest mentions of automata-based programming as a general technique is found in the paper by Peter Naur, 1963.[5] The author calls the technique *Turing machine approach*, however no real Turing machine is given in the paper; instead, the technique based on states and steps is described.

Compared against imperative and procedural programming

The notion of state is not exclusive property of automata-based programming. Generally speaking, *state* (or program state) appears during execution of any computer program, as a combination of all information that can change during the execution. For instance, a *state* of a traditional imperative program consists of

1. values of all variables and the information stored within dynamic memory
2. values stored in registers
3. stack contents (including local variables' values and return addresses)
4. current value of the instruction pointer

These can be divided to the **explicit** part (such as values stored in variables) and the **implicit** part (return addresses and the instruction pointer).

Having said this, an automata-based program can be considered as a special case of an imperative program, in which implicit part of the state is minimized. The state of the whole program taken at the two distinct moments of entering the *step* code section can differ in the automaton state only. This simplifies the analysis of the program.

Object-oriented programming relationship

In the theory of object-oriented programming an **object** is said to have an internal *state* and is capable of *receiving messages*, *responding* to them, *sending*
messages to other objects and changing the internal state during message handling. In more practical terminology, *to call an object's method* is considered the same as *to send a message to the object*.

Thus, from one hand, objects from object-oriented programming can be considered as automata (or models of automata) whose *state* is the combination of internal fields, and one or more methods are considered to be the *step*. Such methods must not call each other nor theirselves, neither directly nor indirectly, otherwise the object can not be considered to be implemented on the automata-based manner.

From the other hand, it is obvious that *object*
is good for implementing a model of an automaton. When the automata-based approach is used within an object-oriented language, an automaton model is usually implemented by a class, the *state* is represented with internal (private) fields of the class, and the *step*
is implemented as a method; such a method is usually the only non-constant public method of the class (besides constructors and destructors). Other public methods could query the state but don't change it. All the secondary methods (such as particular state handlers) are usually hidden within the private part of the class.

References

1. ^ ***a b*** Aho, Alfred V.; Ullman, Jeffrey D. (1973). *The theory of parsing, translation and compiling* **1**. Englewood Cliffs, N. J.: Prentice-Hall.
2. **^** RFC 793
3. **^** Ollongren, Alexander (1974). *Definition of programming languages by interpreting automata*. London: Academic Press.
4. **^**
   Johnson, W. L.; Porter, J. H.; Ackley, S. I. & Ross, D. T. (1968), "Automatic generation of efficient lexical processors using finite state techniques", *Comm ACM* **11** (12): 805-813
5. **^** Naur, Peter (September 1963). "The design of the GIER ALGOL compiler Part II" (in English). *BIT Numerical Mathematics* **3**: 145-166. doi:10.1007/BF01939983. ISSN 0006-3835.

See also

- Erlang: Gen_Fsm Behaviour (http://erlang.org/doc/design_principles/fsm.html) FSM support in Erlang
- Esterel, an automata-based language
- J. V. Noble. «Finite State Machines in Forth» (http://dec.bournemouth.ac.uk/forth/jfar/vol7/paper1/paper.html) — automata-based programming in Forth
- Harell, David (1987). "Statecharts: A Visual Formalism for Complex Systems". *Sci. Comput. Programming* (8): 231—274.
- Harell, David; Drusinsky, D. (1989). "Using Statecharts for Hardware Description and Synthesis". *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems* (8): 798—807.
- Libero home page (http://legacy.imatix.com/html/libero/index.htm) Libero — automata-based code generator (exists since 1982)
- GNU AutoFSM (http://www.gnu.org/software/autogen/autofsm.html) FSM-bsed code generator

Retrieved from "http://en.wikipedia.org/wiki/Automata-based_programming"
Categories: Programming paradigms