10. Strings

Learning objectives:

- searching for patterns in a string
- finite-state machine

Most programming languages support simple operations on strings (e.g. comparison, concatenation, extraction, searching). Searching for a specified pattern in a string (text) is the computational kernel of most string processing operations. Several efficient algorithms have been developed for this potentially time-consuming operation. The approach presented here is very general; it allows searching for a pattern that consists not only of a single string, but a set of strings. The cardinality of this set influences the storage space needed, but not the time. It leads us to the concept of a *finite-state machine* (fsm).

Recognizing a pattern consisting of a single string

Problem: Given a (long) string $z = z_1 z_2 \dots z_n$ of n characters and a (usually much shorter) string $p = p_1\, p_2 \dots p_m$ of m characters (the pattern), find all (nonoverlapping) occurrences of p in z. By sliding a window of length m from left to right along z and examining most characters z_i m times we solve the problem using $m \cdot n$ comparisons. By constructing a finite-state machine from the pattern p it suffices to examine each character z_i exactly once, as shown in Exhibit 10.1. Each state corresponds to a prefix of the pattern, starting with the empty prefix and ending with the complete pattern. The input symbols are the input characters $z_1, z_2, \dots, z_n$ of z. In the j-th step the input character z_j leads from a state corresponding to the prefix $p_1\, p_2 \dots p_i$ to

- the state with prefix $p_1\, p_2 \dots p_i\, p_{i+1}$ if $z_j = p_i{+}1$
- a different state (often the empty prefix, λ) if $z_j \neq p_{i+1}$

Example

p = barbara (Exhibit 10.1).

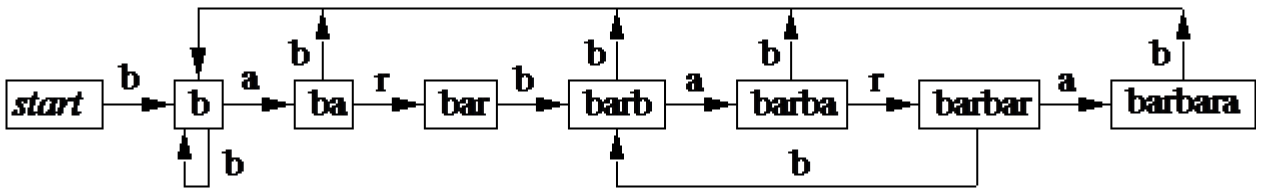

Exhibit 10.1: State diagram showing some of the transitions. All other state transitions lead back to the initial state.

Notice that the pattern 'barbara', although it sounds repetitive, cannot overlap with any part of itself. Constructing a finite-state machine for such a pattern is straightforward. But consider a self-overlapping pattern such as 'barbar', or 'abracadabra', or 'xx', where the first k > 0 characters are identical with the last: The text 'barbarbar' contains two overlapping occurrences of the pattern 'barbar', and 'xxxx' contains three occurrences of 'xx'. A finite-state machine constructed in an analogous fashion as the one used for 'barbara' always finds the first of several overlapping occurrences but might miss some of the later ones. As an exercise, construct finite-state machines that detect all occurrences of self-overlapping patterns.

Recognizing a set of strings: a finite-state-machine interpreter

Finite-state machines (fsm, also called "finite automata") are typically used to recognize patterns that consist of a *set* of strings. An adequate treatment of this more general problem requires introducing some concepts and terminology widely used in computer science.

Given a finite set A of input symbols, the *alphabet*, A* denotes the (infinite) set of all (finite) strings over A, including the nullstring λ. Any subset $L \subseteq A^*$, finite or infinite, is called a set of strings, or a *language,* over A. *Recognizing a language* L refers to the ability to examine any string $z \in A^*$, one symbol at a time from left to right, and deciding whether or not $z \in L$.

A *deterministic* finite-state machine M is essentially given by a finite set S of *states*, a finite alphabet A of *input symbols*, and a *transition function* f: S x A $\rightarrow$ S. The state diagram depicts the states and the inputs, which lead from one state to another; thus a finite-state machine maps strings over A into sequences of states.

When treating any specific problem, it is typically useful to expand this minimal definition by specifying one or more of the following additional concepts. An *initial state* s_0 S, a subset $F \subseteq S$ of *final* or *accepting states*, a finite alphabet B of *output symbols* and an *output function* g: S $\rightarrow$ B, which can be used to assign certain actions to the states in S. We use the concepts of initial state s_0 and of accepting states F to define the notion "recognizing a set of strings":

A set $L \subseteq A^*$ of strings is *recognized* or *accepted* by the finite-state machine M = (S, A, f, s_0, F) iff all the strings in L, and no others, lead M from s_0 to some state $s \in F$.

Example

Exhibit 10.3 shows the state diagram of a finite-state machine that recognizes parameter lists as defined by the syntax diagrams in Exhibit 10.2. L (letter) stands for a character a .. z, D (digit) for a digit 0 .. 9.

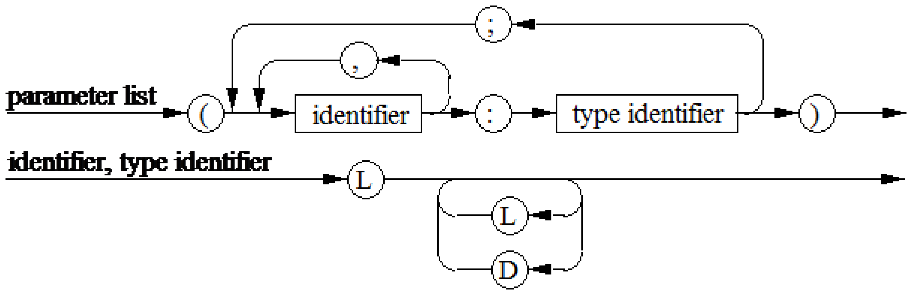

Exhibit 10.2: Syntax diagram of simple parameter lists.

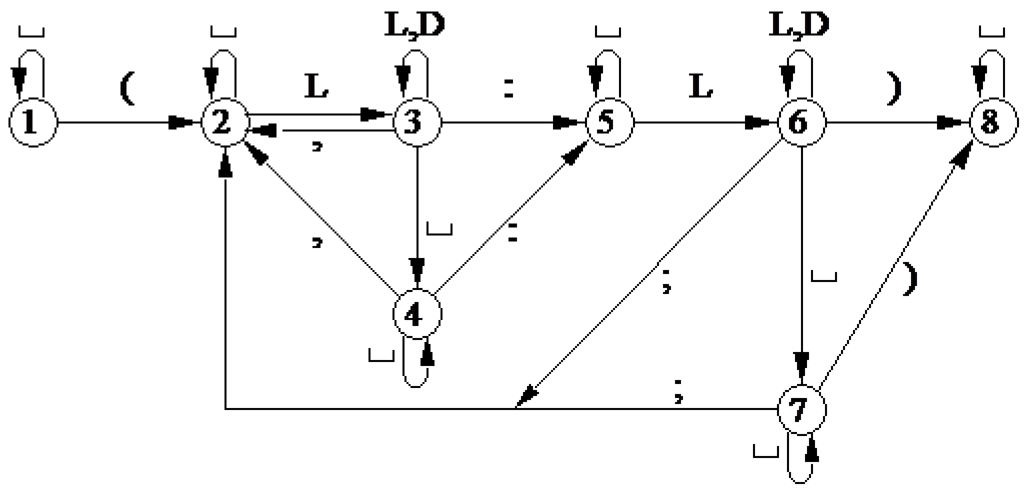

Exhibit 10.3: State diagram of finite-state machine to accept parameter lists. The starting state is '1', the single accepting state is '8'.

A straightforward implementation of a finite-state machine interpreter uses a transition matrix T to represent the state diagram. From the current state s the input symbol c leads to the next state T[s, c]. It is convenient to introduce an error state that captures all illegal transitions. The transition matrix T corresponding to Exhibit 10.3 looks as follows:

```
L represents a character a .. z.
D represents a digit 0 .. 9.
! represents all characters that are not explicitly mentioned.
```

	␣	(	)	:	,	;	L	D	!	
0	0	0	0	0	0	0	0	0	0	error state
1	1	2	0	0	0	0	0	0	0	skip blank
2	2	0	0	0	0	0	3	0	0	left parenthesis read
3	4	0	0	5	2	0	3	3	0	reading variable identifier
4	4	0	0	5	2	0	0	0	0	skip blank
5	5	0	0	0	0	0	6	0	0	colon read
6	7	0	8	0	0	2	6	6	0	reading type identifier
7	7	0	8	0	0	2	0	0	0	skip blank
8	8	0	0	0	0	0	0	0	0	right parenthesis read

The following is a suitable environment for programming a finite-state-machine interpreter:

```
const  nstate = 8; { number of states, without error state }
type state = 0 .. nstate; { 0 = error state, 1 = initial state }
   inchar = ' ' .. '¨'; { 64 consecutive ASCII characters }
   tmatrix = array[state, inchar] of state;
var  T: tmatrix;
```

After initializing the transition matrix T, the procedure 'silentfsm' interprets the finite-state machine defined by T. It processes the sequence of input characters and jumps around in the state space, but it produces no output.

```
procedure silentfsm(var T: tmatrix);
var  s: state;  c: inchar;
begin
   s := 1;  { initial state }
   while  s ≠ 0  do  { read(c);  s := T[s, c] }
```

```
        end;
```

The simple structure of 'silentfsm' can be employed for a useful finite-state-machine interpreter in which initialization, error condition, input processing and transitions in the state space are handled by procedures or functions 'initfsm', 'alive', 'processinput', and 'transition' which have to be implemented according to the desired behavior. The terminating procedure 'terminate' should print a message on the screen that confirms the correct termination of the input or shows an error condition.

```
        procedure fsmsim(var T: tmatrix);
        var  … ;
        begin
          initfsm;
          while  alive  do  { processinput;  transition };
          terminate
        end;
```

Exercise: finite-state recognizer for multiples of 3

Consider the set of strings over the alphabet {0, 1} that represent multiples of 3 when interpreted as binary numbers, such as: 0, 00, 11, 00011, 110. Design two finite-state machines for recognizing this set:

- *Left to right:* Mlr reads the strings from most significant bit to least significant.
- *Right to left:* Mrl reads the strings from least significant bit to most significant.

Solution

Left to right: Let r_k be the number represented by the k leftmost bits, and let b be the (k + 1)-st bit, interpreted as an integer. Then $r_{k+1} = 2 \cdot r_k + b$. The states correspond to r_k mod 3 (Exhibit 10.4). Starting state and accepting state: 0'.

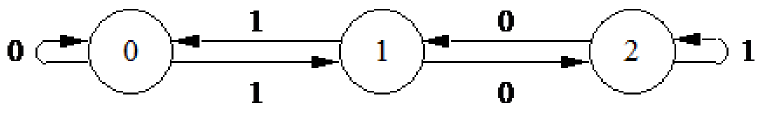

Exhibit 10.4: Finite-state machine computes remainder modulo 3 left to right.

Right to left: $r_{k+1} = b \cdot 2^k + r_k$. Show by induction that the powers of 2 are alternatingly congruent to 1 and 2 modulo 3 (i.e. 2^k mod 3 = 1 for k even, 2^k mod 3 = 2 for k odd). Thus we need a modulo 2 counter, which appears in Exhibit 10.5 as two rows of three states each. Starting state: 0. Accepting states: 0 and 0'.

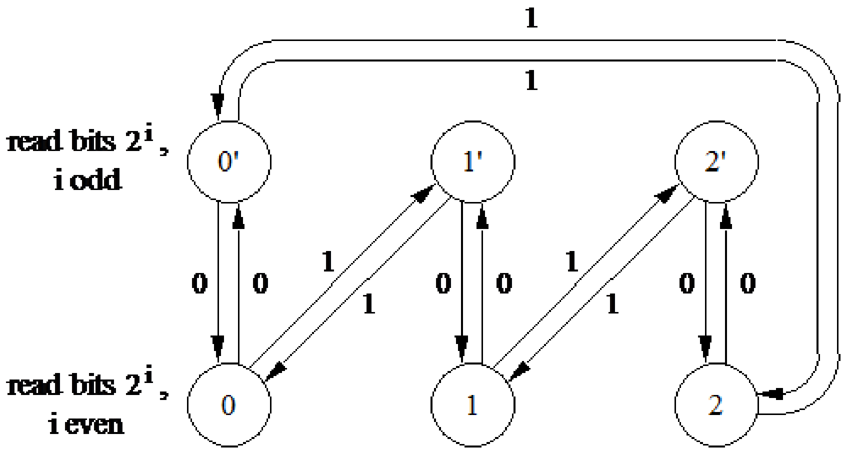

Exhibit 10.5: Finite-state machine computes remainder modulo 3 right to left.

Exercises and programming projects

1. Draw the state diagram of several finite-state machines, each of which searches a string z for all occurrences of an interesting pattern with repetitive parts, such as 'abaca' or 'Caracas'.
2. Draw the state diagram of finite-state machines that detect all occurrences of a self-overlapping pattern such as 'abracadabra', 'barbar', or 'xx'.
3. Finite-state recognizer for various days:
   Design a finite-state machine for automatic recognition of the set of nine words:

   ```
   'monday','tuesday','wednesday','thursday',
   'friday', 'saturday', 'sunday', 'day', 'daytime'
   ```

   in a text. The underlying alphabet consists of the lowercase letters 'a' .. 'z' and the blank. Draw the state diagram of the finite-state machine; identify the initial state and indicate accepting states by a double circle. It suffices to recognize membership in the set without recognizing each word individually.
4. Implementation of a pattern recognizer:
   Some useful procedures and functions require no parameters, hence most programming languages incorporate the concept of an empty parameter list. There are two reasonable syntax conventions about how to write the headers of parameterless procedures and functions:

   ```
   (1)  procedure p;   function f: T;
   (2)  procedure p();function f(): T;
   ```

   Examples: Pascal uses convention (1); Modula-2 allows both (1) and (2) for procedures, but only (2) for function procedures.
   For each convention (1) and (2), modify the syntax diagram in Exhibit 10.2 to allow empty parameter lists, and draw the state diagrams of the corresponding finite-state machines.
5. Standard Pascal defines parameter lists by means of the syntax diagram shown in Exhibit 10.6.

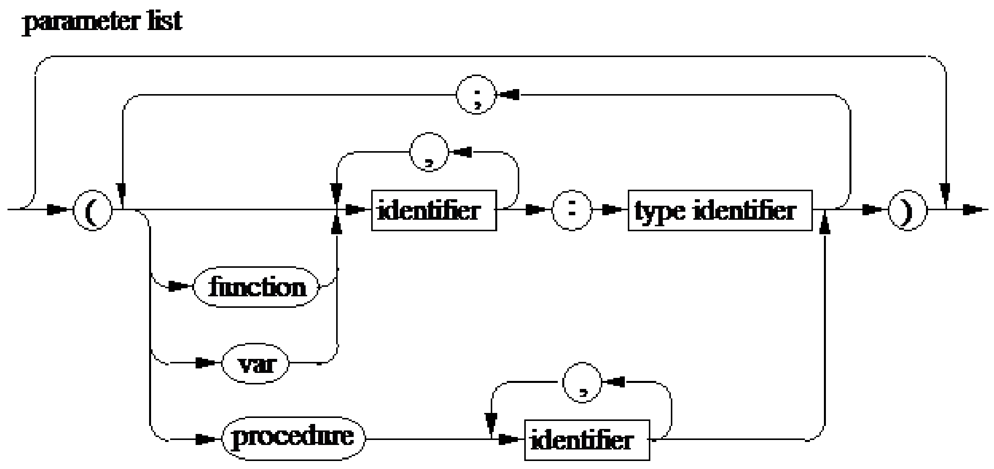

Exhibit 10.6: Syntax diagram for standard Pascal parameter lists.

Draw a state diagram for the corresponding finite-state machine. For brevity's sake, consider the reserved words 'function', 'var' and 'procedure' to be atomic symbols rather than strings of characters.