**State Machines**

**Introduction:**

consider a system that accepts or rejects strings of input. Assume the system has the following rules:

Acceptable "sentences" consist of one or more A elements, Or, they can consist of one or more A elements, followed by a B element, followed by one or more A elements again. Here are examples of valid sentences:

A

AAAA

AAA..AAAA

AAABAA

ABAAA

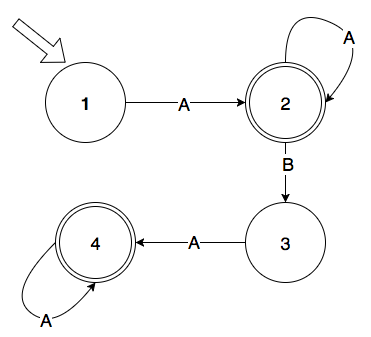
The following are examples of invalid (rejected) sentences:

[EMPTY INPUT]

BAAAA

AAB (AA can be considered a valid sentence, but not AAB)

One can express these rules using a GRAPH (also called a **state machine**):



In this graph, the arcs represent the input (see arc labels) and the circles represent "states." Circles with double line are called "success states."

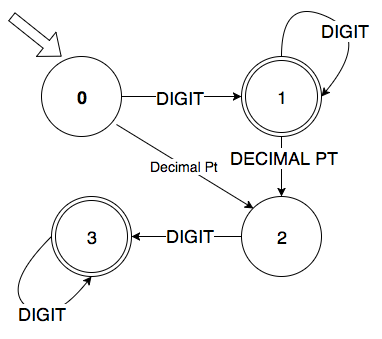
As we process input one item at a time and follow the arcs from one state to the next, at some point (may be at the end of the input,) we arrive at a state for whom there is no arc to the next state. At this point, the input we received (and the data we assembled using that input) at the last success state is a valid input.

consider the following input:

AABA.

We begin at state 1, we take input A, we follow the arc to state 2 [a success state]. we take another input A (the second A in the input) and we follow the loop arc back to state 2 [a success state] we take the B as input, we follow the arc from state 2 to state 3  and the third A will take us to state 4 [success state]. But the period is not in this grammar and that causes us to "fail" But all the input we gathered at the last success state (state 4: AABA) is acceptable input. All the input up to the period is considered processed.

**A state machine for a valid double number:**

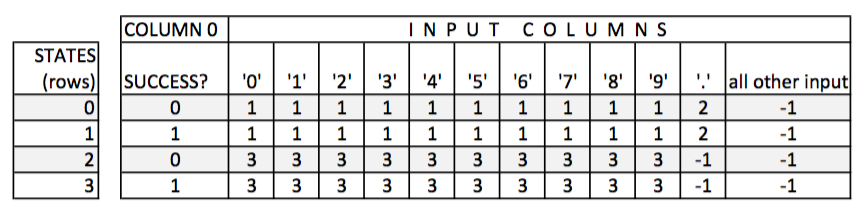


Bases on this state machine, [3.14], [23], [42.0], [.5] are all valid sentences. but [54.] is not

**Using an "Adjacency Matrix" to represent a graph:**

We will create a table (a two dimensional array) where the rows represent the states (0, 1, 2...) and the columns represent the possible inputs. The cells of the table where row R and column C intersect indicate the state to which R is connected to via input C. For example, if you are in state 1 and you see a decimal point, you will end up in state 2. If you are in state 2 and you see a decimal point, you will go to state -1 which does not exist, so your process will stop.

We will use column zero to store 1 to mark success states and fail states will be marked with zero.



**Formalizing the adjacency matrix:**

Declare a two dimensional int array with 30 rows and 256 columns. Each column of the array represents the ASCII value of all possible input characters. So, for input character '0', we will look at column 48 (ASCII value for '0')

Obviously, it will not be easy to mannually fill and maintain this array accurately. So, we need some functions to do that for us:

//Fill all cells of the array with -1

void init\_table(int \_table[][MAX\_COLUMNS]);

//Mark this state (row) with a 1 (success)

void mark\_success(int \_table[][MAX\_COLUMNS], int state);

//Mark this state (row) with a 0 (fail)

void mark\_fail(int \_table[][MAX\_COLUMNS], int state);

//true if state is a success state

bool is\_success(int \_table[][MAX\_COLUMNS], int state);

//Mark a range of cells in the array.

void mark\_cells(int row, int \_table[][MAX\_COLUMNS], int from, int to, int state);

//Mark columns represented by the string columns[] for this row

void mark\_cells(int row, int \_table[][MAX\_COLUMNS], const char columns[], int state);

//Mark this row and column

void mark\_cell(int row, int table[][MAX\_COLUMNS], int column, int state);

//This can realistically be used on a small table

void print\_table(int \_table[][MAX\_COLUMNS]);

//show string s and mark this position on the string:

//hello world pos: 7

// ^

void show\_string(char s[], int \_pos);

**Preparing the table for the token DOUBLE:**

//doubles:

mark\_fail(table, 0); //Mark states 0 and 2 as fail states

mark\_success(table, 1); //Mark states 1 and 3 as success states

mark\_fail(table, 2);

mark\_success(table, 3);

mark\_cells(0, table, DIGITS, 1); //state [0] --- DIGITS ---> [1]

mark\_cells(0, table, '.', '.', 2); //state [0] --- '.' ------> [2]

mark\_cells(1, table, DIGITS, 1); //state [1] --- DIGITS ---> [1]

mark\_cells(1, table, '.', '.', 2); //state [1] --- '.' ------> [2]

mark\_cells(2, table, DIGITS, 3); //state [2] --- DIGITS ---> [3]

mark\_cells(3, table, DIGITS, 3); //state [3] --- DIGITS ---> [3]

**A Function To Extract A Valid "Token" From A String Input:**

The function will take as input the table, the input string, the current position in the string, and the start state. It tries to find the longest possible string that will end in a success state. If the function is successful, it will return true after resetting the position to the character following the last acceptable character in the string and storing the extracted string in object token.

