YAFSM Language Specification

Compilers Project

Group 6

Authors

Vishal Vijay Devadiga (CS21BTECH11061)

Satpute Aniket Tukaram (CS21BTECH11056)

Mahin Bansal (CS21BTECH11034)

Harshit Pant (CS21BTECH11021)

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Table of Contents

Introduction

What is YAFSM?

YAFSM is a domain specific language that simplifies working with Finite State Machines(FSMs). Finite State Machines include Deterministic Finite Automata(DFAs), Non-Deterministic Finite Automata(NFAs), Pushdown Automata(PDAs).

It supports the following features:

- Defining Finite State Machines
- Regular Expressions
- Context Free Grammars

Why YAFSM?

Finite State Machines are used in many applications, such as:

- Regular Expressions
- Lexical Analysis
- Compilers
- Network Protocols
- Digital Logic
- Artificial Intelligence
- Natural Language Processing
- etc.

Finite State Machines are used in many applications, but the syntax for defining a Finite State Machine is not very intuitive. YAFSM aims to simplify the syntax for defining a Finite State Machine, making it easier for programmers to work with Finite State Machines.

Language Specifications

YAFSM follows, making it easier for programmers to pick up YAFSM easily and keep their focus on the logic rather than YAFSM.

- YAFSM is a **statically typed** language
- YAFSM is a **strongly typed** language
- YAFSM is a **procedural** language
- YAFSM is case sensitive.

YAFSM does not support Object Oriented Programming(**OOPs**).

Data Types

YAFSM uses common data types found in most programming languages.

Primitive Data Types

Integer: Signed Integers are represented by the `int_x` keyword, where `x` is the number of bits used to represent the integer. YAFSM supports 8, 16, 32 and 64 bit integers.

```
int_8 a;
int_8 b = 10;
int_16 c = 20;
int_32 d = 30;
int_64 e = 40;
```

Unsigned Integers are represented by the `uint_x` keyword, where `x` is the number of bits used to represent the integer. YAFSM supports 8, 16, 32 and 64 bit integers.

```
unit_8 a;
uint_8 b = 10;
uint_16 c = 20;
uint_32 d = 30;
uint_64 e = 40;
```

Character: Characters are represented by the `char` keyword. YAFSM supports 8 bit characters.

```
char a;
char b = 'a';
```

Float: Floats are represented by the `float_x` keyword, where `x` is the number of bits used to represent the float. YAFSM supports 32 and 64 bit floats.

```
float_32 a;
float_32 b = 10.5;
float_64 c = 20.5;
```

Boolean: Booleans are represented by the `bool` keyword, which is similar to the `bool` keyword in C, C++, Java and Python.

```
bool a;
bool b = true;
bool c = false;
```

Composite Data Types

Strings: Strings are represented by the `string` keyword. Strings are immutable, and can be indexed using the `[]` operator.

```
string temp;
string a = "Hello World";
char b = a[0];
```

Finite-Sets: Sets are collections of elements of the same data type. YAFSM supports two types of sets: Ordered Sets and Unordered Sets.

- Ordered Sets are represented by the `o_set` keyword.
- Unordered Sets are represented by the `u_set` keyword.

```
o_set<int_8> a;
o_set<int_8> b = {1, 2, 3};
```

Structs: Structs are represented by the `struct` keyword. Structs can contain any data type supported by YAFSM.

```
struct Point {
    int_8 x;
    int_8 y;
    float_32 z;
    string name;
};
```

Additionally structs can contain other data types which have already been defined as structs

```
Point a,b;

struct set_of_coords {
  o_set<Point> coords = {a,b};
};
```

Regular Expressions: Regular Expressions are represented by the `regex` keyword. Regular Expression can contain definitions of other Regular Expressions, and can be used to define Finite State Machines.

```
regex alphabet = r'[a-z]';
regex Letter = r'{alphabet}';
regex Digit = r'[0-9]';
regex a = r'[ab]{2}';
regex b = r'{a}*';
regex c = r'{a}+';
regex d = r'{a}?';
regex e = r'^[ab]';
regex f = r'[ab]$';
regex g = r'{a}|{b}';
```

Context Free Grammars: Context Free Grammars are represented by the `cfg` keyword. Context Free Grammars are defined by a 4-tuple:

$$(N, \Sigma, P, S)$$

where:

- N is a set of non-terminal symbols
- Σ is a set of terminal symbols
- P is a set of production rules
- S is the start symbol

A production rule is represented as:

- A -> α where α is a string of terminal and non-terminal symbols
- A -> { α_1, α_2, ... } where α_1, α_2, ... are strings of terminal and non-terminal symbols.

```
cfg a;
a.T = {a:"a", demo:"b"};
a.N = {A, B};
a.S = A;
a.P = {
    A -> ${a}B,
    B -> ${demo}A,
    A -> {${a}A, \e}
};
```

DFAs: DFAs are represented by the `dfa` keyword.

A DFA is defined by a 5-tuple:

$$(Q, \Sigma, \delta, q_0, F)$$

where:

- Q is a `o_set` of states
- Σ is a **set** of input symbols
- δ is the transition function, which maps $Q \times \Sigma$ to Q
- q_0 is the initial state
- F is a set of final states

A transition can be represented as:

state1, input_symbol -> state2

In case of multiple transitions from the same state on different input symbols to the same state, the transitions can be represented as:

state1, {input_symbol1, input_symbol2, . . . } -> state2

This can also be done as:

state1, <regex> -> state2

state1 , <set> -> state2

δ is either a set of such transitions or it can be represented as a matrix of size $|Q| \times |\Sigma|$, where each element of the matrix is a state.

```
dfa a;
a.Q = {q0, q1, q2};
a.Sigma = {a:"0",b:"1",c:"2"};
```

```
a.delta = {
    q0, ${a} -> q1,
    a.Q[0] , ${b} -> a.Q[1],
    q1 , r'[${a}${c}]' -> q2,
    q2, { ${a} , ${c} } -> q1
};
a.q0 = q0;
a.F = {q1,q2};
```

NFAs: NFAs are represented by the `nfa` keyword.

A NFA is defined by a 5-tuple:

$$(Q, \Sigma, \delta, q_0, F)$$

where:

- Q is a `o_set` of states
- Σ is a **set** of input symbols
- δ is the transition function, which maps $Q \times \Sigma$ to 2^Q
- q_0 is the initial state
- F is a set of final states

Here 2^Q represent the power set of Q.

A transition can be represented as:

- state1, input_symbol -> {state2, state3, . . . }
- state1, {input_symbol1, input_symbol2, . . . } -> {state2, state3, . . . }
- state1, <regex> -> {state2, state3, . . . }
- state1, <set> -> {state2, state3, . . . }

Here input_symbols can include ϵ which is represented by '\e'.

```
nfa a;
a.Q = {q0, q1, q2};
a.Sigma = {a:"0",b:"1",c:"2"};
a.delta = {
    q0, ${a} -> {q1, q2},
    a.Q[0] , ${b} -> a.Q[1],
    q1 , r$'[${a}${b}]' -> q2,
    q2, {${a},${c}} -> q1,
    q2, \e -> q0
```

```
};
a.q0 = q0;
a.F = {q1,q2};
```

PDAs: PDAs are represented by the `pda` keyword.

A PDA is defined by a 6-tuple:

$$(Q, \Sigma, \Gamma, \delta, q_0, F)$$

where:

- Q is a `o_set` of states
- Σ is a **set** of input symbols
- Γ is a **set** of stack symbols
- δ is the transition function, which maps $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$ to $2^{Q \times \Gamma_\epsilon}$
- q_0 is the initial state
- F is a set of final states

Here $2^{Q \times \Gamma_\epsilon}$ represent the power set of $Q \times \Gamma_\epsilon$.

A transition can be represented as:

- state1, input_symbol, stack_symbol -> state2, stack_symbol
- state1, {(input_symbol1, stack_symbol1), (input_symbol2, stack_symbol2), ...} -> state2, stack_symbol
- state1, input_symbol, stack_symbol -> {(state2, stack_symbol2), (state3,stack_symbol3), ...}

```
pda a;
a.Q = {A, B, C};
a.S = {a:"a", b:"b", c:"c"};
a.G = {a:"a", b:"b", d:"d"};
a.delta = {
    A, ${a}, ${a} -> A, ${a},
    A, {(${b}, ${a}),(${b}, ${c})} -> {(B, ${a}), (A, ${d})},
    B, {(${b}, ${a}), (${b}, ${c})} -> B, ${a},
    B, ${c}, ${a} -> C, \e,
    C, ${c}, ${a} -> {(C, \e), (C, ${a})}
};
a.q0 = A;
a.F = {C};
```

Comments

YAFSM has only one type of comment, that can act as both single line and multi line comments. The comment starts with `<!--` and ends with `--!>`. Below is an example of a comment:

```
<!-- This is a comment --!>

<!-- This is a
multi line comment --!>
```

Operators

Operators supports by YAFSM are similar to the operators supported by C.

Arithmetic Operators

Operator	Description	Associativity
+	Addition	Left to Right
-	Subtraction	Left to Right
*	Multiplication	Left to Right
/	Division	Left to Right
%	Modulo	Left to Right

Logical Operators

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>&&</td><td>Logical AND</td><td>Left to Right</td></tr>
<tr><td>||</td><td>Logical OR</td><td>Left to Right</td></tr>
<tr><td>!</td><td>Logical NOT</td><td>Right to Left</td></tr>
</table>

Comparison Operators

Operator	Description	Associativity
==	Equal to	Left to Right
!=	Not equal to	Left to Right
>	Greater than	Left to Right
<	Less than	Left to Right
>=	Greater than or equal to	Left to Right

Operator	Description	Associativity
<=	Less than or equal to	Left to Right

Assignment Operators

Operator	Description	Associativity
=	Assignment	Right to Left
+=	Addition Assignment	Right to Left
-=	Subtraction Assignment	Right to Left
*=	Multiplication Assignment	Right to Left
/=	Division Assignment	Right to Left
%=	Modulo Assignment	Right to Left
&=	Logical AND Assignment	Right to Left
|=	Logical OR Assignment	Right to Left

Set Operators

Operator	Description	Associativity
+	Union	Left to Right
-	Difference	Left to Right
*	Intersection	Left to Right
^2	Power Set	Left to Right

```
o_set<int_8> a = {1, 2, 3};
o_set<int_8> b = {2, 3, 4};
o_set<int_8> c = a + b; <!-- c = {1, 2, 3, 4} --!>
o_set<int_8> d = a - b; <!-- d = {1} --!>
o_set<int_8> e = a * b; <!-- e = {2, 3} --!>
o_set<o_set<int_8>> f = a^2; <!-- f = {{}, {1}, {2}, {3}, {1, 2},
{1, 3}, {2, 3}, {1, 2, 3}} --!>
```

Automaton Operators

Operator	Description	Associativity
*	Kleene	Left to Right

Operator	Description	Associativity
@	Concatenation	Left to Right
+	Union	Left to Right
!	Negation	Right to Left

```
dfa a;
dfa b;

dfa c = a*; <!-- Kleene Star --!>
dfa d = a@b; <!-- Concatenation --!>
dfa e = a+b; <!-- Union --!>
dfa f = !a; <!-- Negation --!>
```

Misc Operators

Operator	Description	Associativity
.	Access Struct Member	Left to Right
[]	Access Set Element	Left to Right
()	Function Call	Left to Right

```
struct Point {
    int_8 x;
    int_8 y;
}

Point p;
p.x = 10;

o_set<int_8> a = {1, 2, 3};
int_8 b = a[0];

int_8 func(int_8 a, int_8 b) {
    return a + b;
}

int_8 a = func(10, 20);
```

Operator Precedence

Operator	Description
()	Parentheses
!	Logical NOT
*, /, %	Multiplication, Division, Modulo
+, -	Addition, Subtraction
>, <, >=, <=	Comparison
==, !=	Equality
&&	Logical AND
||	Logical OR
=	Assignment
+=, -=, *=, /=, %=, &=, |=	Assignment

Set Operators Precedence

Operator	Description
()	Parentheses
^2	Power Set
*,+,-	Intersection, Union, Set Difference

Automaton Operators Precedence:

Operator	Description
()	Parentheses
*,!	Kleene Star, Negation
@,+	Concatenation, Union

Control Flow

YAFSM enforce the use of curly braces for all control flow statements. YAFSM does not support the use of indentation for control flow statements. YAFSM supports the following control flow statements:

If-Else

Below is the syntax for the if-else statement:

```
if (condition) {
    statement;
}
elif (condition) {
    statement;
}
else {
    statement;
}
```

Loops

YAFSM only supports the `while` loop. Below is the syntax for the `while` loop:

```
while (condition) {
    statements;
}
```

Constants

Constants are represented by the `const` keyword. Constants can be of any data type supported by YAFSM.

Keywords

Keyword	Description
`int_x`	Integer
`uint_x`	Unsigned Integer
`char`	Character
`float_x`	Float
`bool`	Boolean
`const`	Constant
`struct`	Struct
`o_set`	Ordered Set
`u_set`	Unordered Set
`string`	String
`regex`	Regular Expression
`dfa`	DFA

Keyword	Description
`nfa`	NFA
`pda`	PDA
`cfg`	CFG
`if`	If
`elif`	Else If
`else`	Else
`while`	While
`break`	Break
`continue`	Continue
`return`	Return
`true`	True
`false`	False
`<!--`	Start of comment
`--!>`	End of comment

Identifiers

YAFSM uses the following rules for identifiers:

- Identifiers can only contain alphanumeric characters and underscores.
- Identifiers cannot start with a number.
- Identifiers cannot be a keyword.
- Identifiers cannot contain spaces.
- Identifiers cannot contain special characters.

Regular Expressions for Identifiers:

`[a-zA-Z_][a-zA-Z0-9_]*`

Statements

YAFSM supports the following statements:

Declaration Statement

Declaration statements are used to declare variables. Below is the syntax for declaration statements:

```
data_type identifier;
```

Multiple variables of the same data type can be declared in a single statement:

```
data_type identifier1, identifier2, ...;
```

Assignment Statement

Assignment statements are used to assign values to variables. Below is the syntax for assignment statements:

```
identifier = expression;
```

Function Declaration Statement

Function declaration statements are used to declare functions. Below is the syntax for function declaration statements:

```
data_type function_name(data_type1 arg1, data_type2 arg2, ...) {
    statements;
}
```

Function Call Statement

Function call statements are used to call functions. Below is the syntax for function call statements:

```
function_name(arg1, arg2, ...);
```

In case the function returns a value, the function call statement can be used as an expression:

```
data_type variable = function_name(arg1, arg2, ...);
```

IO Statements

Print statements are used to print values to the console. Below is the syntax for print statements:

```
out(expression);
```

Input statements are used to take input from the console. Below is the syntax for input statements:

```
inp(identifier);
```

In case multiple variables need to be inputted, the input statement can be used as:

```
inp(identifier1, identifier2, ...);
```

In-built Functions

YAFSM supports multiple in-built functions to help with working with Finite State Machines.

Set Functions

Function	Description	Return Type
`size(o_set<T> S)`	returns the size of the `o_set`	`int_8`
`size(u_set<T> S)`	returns the size of the `u_set`	`int_8`
`empty(o_set<T> S)`	returns **true** if S is empty otherwise returns **false**	`bool`
`empty(u_set<T> S)`	returns **true** if S is empty otherwise returns **false**	`bool`
`find(o_set<T> S, T x)`	finds x in S and returns **true** if $x \in S$, otherwise returns **false**	`bool`
`find(u_set<T> S, T x)`	finds x in S and returns **true** if $x \in S$, otherwise returns **false**	`bool`
`insert(o_set<T> S, T x, ...)`	inserts x into S (no duplicates)	-
`insert(u_set<T> S, T x, ...)`	- inserts x into S (no duplicates)	-
`remove(o_set<T> S, T x, ...)`	- remove x from S if $x \in S$	-
`remove(o_set<T> S, T x, ...)`	- remove x from S if $x \in S$	-
`delete(o_set<T> S)`	- deletes all elements from the `o_set`	-
`delete(u_set<T> S)`	deletes all elements from the `u_set`	-
`out(o_set<T> S)`	prints all elements of the `o_set`	-
`out(u_set<T> S)`	prints all elements of the `u_set`	-

```
o_set<int_8> a = {1, 2, 3};
u_set<int_8> b = {};

int_8 size_a = size(a); <!-- size_a = 3 --!>
int_8 size_b = size(b); <!-- size_b = 0 --!>

bool empty_a = empty(a); <!-- empty_a = false --!>
bool empty_b = empty(b); <!-- empty_b = true --!>

bool find_a = find(a, 1); <!-- find_a = true --!>
bool find_b = find(b, 1); <!-- find_b = false --!>
```

```
insert(a, 4); <!-- a = {1, 2, 3, 4} --!>
insert(b, 4); <!-- b = {4} --!>
insert(a,4); <!-- a = {1, 2, 3, 4} --!>

remove(a, 4); <!-- a = {1, 2, 3} --!>
remove(a, 4); <!-- a = {1, 2, 3} --!>

out(a); <!-- prints {1, 2, 3} --!>
out(b); <!-- prints {} --!>

delete(a); <!-- a = {} --!>
delete(b); <!-- b = {} --!>
```

CFG Functions

Function	Description	Return Type
add_T(cfg a, X:"val")	adds a terminal X with value "val" to the CFG	-
add_NT(cfg a, X)	adds a non-terminal X to the CFG	-
add_P(cfg a, X)	adds a production to the CFG where the production is in one of the allowed forms	-
remove_T(cfg a, X)	removes a terminal from CFG along with all the rules that use X	-
remove_NT(cfg a, X)	removes a non-terminal with name X from CFG along with all the rules that use X	-
remove_P(cfg a, X)	removes a production from CFG	-
change_start(cfg a, X)	changes the start symbol of CFG to X where X must be a terminal	-
test_membership(cfg a, X)	returns true if X is a member of the language generated by CFG	`bool`
cfg_to_PDA(cfg a)	converts the CFG a to a PDA and returns the PDA	`pda`
delete(cfg a)	deletes the CFG a	-
out(cfg a)	prints the CFG a	-

```
cfg a;
a.T = {a:"a", demo:"b"};
a.N = {A, B};
a.S = A;
a.P = {
    A -> ${a}B,
    B -> ${demo}A,
    A -> {${a}A, \e}
};

add_T(a, c:"c", d:"d"); <!-- a.T = {a:"a", demo:"b", c:"c", d:"d"}--!>
add_NT(a, E); <!-- a.N = {A, B, E} --!>
add_T(a, e); <!-- error: e has no value --!>

add_P(a, E -> ${c}B); <!-- a.P = {A -> ${a}B, B -> ${demo}A, A -> ${a}A,
A-> \e, E -> ${c}B} --!>

add_P(a, erroneous-production) <!-- error: production is not in
the correct form --!>

remove_T(a, c); <!-- a.T = {a:"a", demo:"b", d:"d"}, a.P = {A -> ${a}B,
B -> ${demo}A, A -> ${a}A, A-> \e} --!>
remove_NT(a, B); <!-- a.N = {A, B}, a.P = {A -> ${a}A, A-> \e} --!>

<!-- if the deleted non-terminal is a start variable then
entire CFG is deleted --!>

remove_P(a, A -> ${a}A); <!-- a.P = {A-> \e} --!>
<!-- if production is not present then no change --!>
remove_P(a, A -> ${a}A); <!-- a.P = {A-> \e} --!>

change_start(a, E); <!-- a.S = E --!>
change_start(a, c); <!-- error: c is not a non-terminal --!>

change_start(a,A) <!-- a.S = A --!>
bool flag = test_membership(a, ""); <!-- flag = true --!>
```

```
pda p = cfg_to_pda(a); <!-- p is the PDA equivalent of the CFG a --!>

out(a); <!-- prints the CFG a --!>

delete(a); <!-- deletes the CFG a --!>
```

DFA Functions

Function	Description	Return Type
insert_states(dfa a, X, ...)	adds states X, ... to the DFA along with transitions from/to X, ...	-
remove_states(dfa a, X)	removes states X, ... from the DFA along wit transitions from/to X, ...	-
nsert_letters(dfa a, X, ...)	adds letters X, ... to Σ of the DFA a	-
remove_letters(dfa a, X)	removes letters X, ... from Σ of a	-
change_start(dfa a, X)	changes the start state of the DFA to X where X	-
insert_final(dfa a, X, ...)	adds states X, ... to the set of final states	-
remove_final(dfa a, X, ...)	removes states X, ... from the set of accepting states	-
add_transition(dfa a, X, ...)	adds a transition X, ... where X is in one of the forms of the transitions mentioned in the DFA section	-
remove_transition(dfa a, X)	removes the transition X from the DFA	-
simulate(dfa a, X)	returns true if X is accepted by the DFA a, otherwise false	`bool`
delete(dfa a)	deletes the DFA a	-
out(dfa a)	prints the DFA a	-

```
dfa a;

a.Q = {q0, q1, q2};
a.q0 = q0;
a.Sigma = {a:"0",b:"1",c:"2"};
```

```
a.F = {q1, q2};
a.delta = {
  q0, ${a} -> q1,
  a.Q[0] , ${b} -> a.Q[1],
  q1 , r'[${a}${c}]' -> q2,
  q2, { ${a}, ${c} } -> q1
};

insert_states(a, q3, q4); <!-- a.Q = {q0, q1, q2, q3, q4} --!>

insert_letters(a, d:"3", e:"4"); <!-- a.Sigma = {a:"0",b:"1",c:"2",d:"3",e:"4"} --!>

change_start(a, q3); <!-- a.q0 = q3 --!>

add_transition(a, q3, ${d} -> q4); <!-- a.delta = {q0, ${a} -> q1, q0, ${e} -> q1,
q1 , ${b} -> q2, q1, ${c} -> q2, q2, ${a} -> q1, q2, ${c} -> q1, q3, ${d} -> q4} --!>

insert_final(a, q3); <!-- a.F = {q1, q2, q3} --!>

remove_states(a, q4); <!-- a.Q = {q0, q1, q2}, a.delta = {q0, ${a} -> q1,
q0, ${b} -> q1, q1 , ${b} -> q2, q1, ${c} -> q2, q2, ${a} -> q1, q2, ${c} -> q1} --!>

remove_letters(a, c); <!-- a.Sigma = {a:"0",b:"1",d:"3",e:"4"}, a.delta = {q0, ${a} -> q1,
q0, ${b} -> q1, q1 , ${b} -> q2, q2, ${a} -> q1} --!>

remove_transition(a, q2, ${a} -> q1); <!-- a.delta = {q0, ${a} -> q1,
 q0, ${b} -> q1,  q1 , ${b} -> q2} --!>

remove_transition(a, q2, ${a} -> q1); <!-- a.delta = {q0, ${a} -> q1,
 q0, ${b} -> q1,  q1 , ${b} -> q2} --!>

simulate(a, 101); <!-- error: DFA a is not in stable state(does not have
transitions for all letters from every state) --!>

out(a) <!-- prints the DFA a --!>
```

```
delete(a); <!-- deletes the DFA a --!>
```

NFA Functions

Functions that work with DFAs also work with NFAs. Some additional functions are

Function	Description	Return Type
nfa_to_dfa(nfa a)	converts the NFA a to a DFA	`dfa`

PDA Functions

Function	Description	Return Type
insert_states(pda a, X, . . .)	inserts states X, . . . into the PDA	-
remove_states(pda a, X, . . .)	removes states X, . . . from the PDA	-
insert_input_letters(pda a, X, . . .)	inserts letters X, . . . into the PDA	-
remove_input_letters(pda a, X, . . .)	removes letters X, . . . from the PDA	-
insert_stack_letters(pda a, X, . . .)	inserts letters X, . . . into the PDA	-
remove_stack_letters(pda a, X, . . .)	removes letters X, . . . from the PDA	-
change_start(pda a, X)	changes the start state of the PDA	-
insert_final(pda a, X, . . .)	inserts states X, . . . into the PDA	-
remove_final(pda a, X, . . .)	removes states X, . . . from the accepting states of PDA	-
insert_transition(pda a, X, . . .)	inserts a transition X, . . . into the PDA	-
remove_transition(pda a, X, . . .)	removes a transition X, . . . from the PDA	-
simulate(pda a, X, . . .)	returns true if X is accepted by the PDA a, otherwise	`bool`
delete(pda a)	deletes the PDA	-
out(pda a)	prints the PDA a	-

Regular Expression Functions

Function	Description	Return Type
simulate(regex a, string s)	returns true if s is accepted by the regex a, otherwise false	`bool`
delete(regex a)	deletes the regex a	`-`
regex_to_dfa(regex a)	converts the regex a to a DFA and returns the DFA	`dfa`
regex_to_nfa(regex a)	converts the regex a to a NFA and returns the NFA	`nfa`

References

- Wikipedia: FSMs
- Wikipedia: DFAs
- Wikipedia: NFAs
- Wikipedia: PDAs
- Wikipedia: CFGs
- Wikipedia: Regular Expressions
- Michael Sipser: Introduction to the Theory of Computation
- C_Programming Language by Kernighan and Ritchie