DFA Assignment

Description: Design a DFA for a possible application.

A Finite State machine also-known-as Finite State Automata is a machine which consists of several states (which include initial, transition and final states)

A deterministic finite state machine contains five tuples

$(\Sigma, S, s_0, \delta, F)$,

where:

- **Σ** is the input alphabet (a finite, non-empty set of symbols).
- **S** is a finite, non-empty set of states.
- **s_0** is an initial state, an element of S**.**
- **δ** is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
- (in a nondeterministic finite state machine it would be $\delta : S \times \Sigma \rightarrow \wp(S)$, i.e., δ would return a set of states). ($\wp(S)$ is the Power set of S)
- **F** is the set of final states, a (possibly empty) subset of **S.**

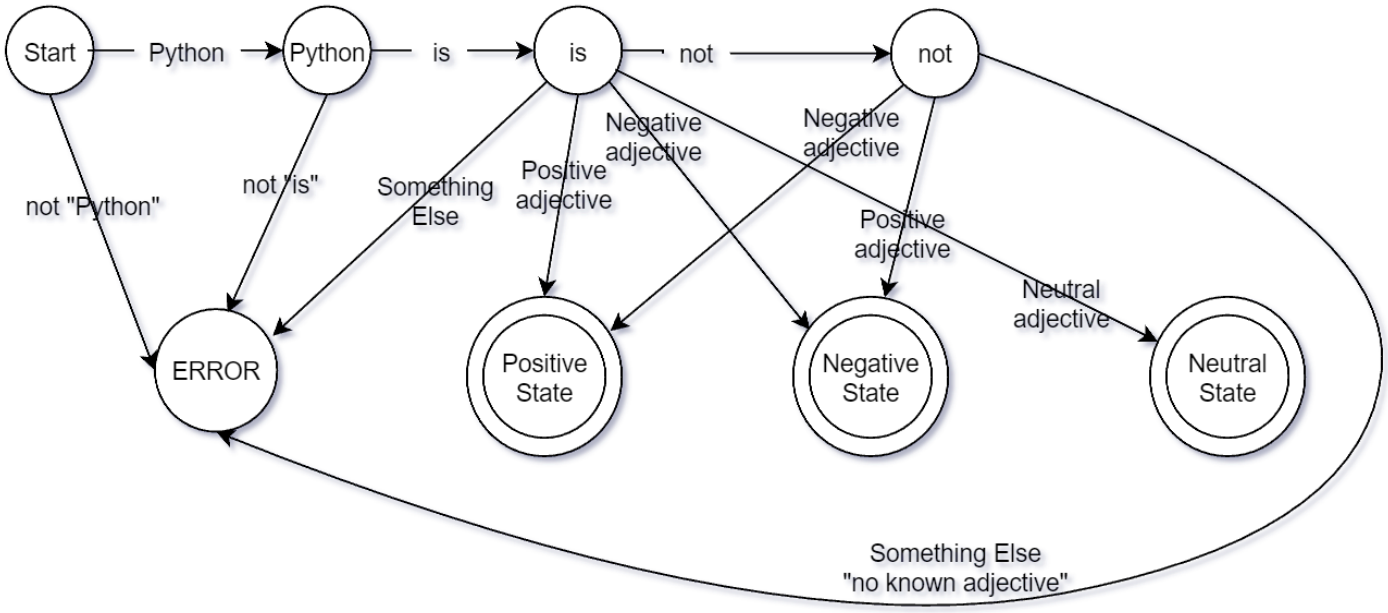
Simple Application of DFA is to know the meaning of a sentence whether it has a positive, negative or neutral meaning.

We want to recognize the meaning of very small sentences with a limited vocabulary and syntax.

These sentences should start with "Python is" followed by:

- an adjective or
- the word "not" followed by an adjective. e.g.
  "Python is great" => positive meaning
  "Python is stupid" => negative meaning
  "Python is not ugly" => positive meaning

DFA Diagram for the application



Python Implementation

```
# Creating the General Finite State Machine
class StateMachine:
      def __init__(self):
        self.handlers = {}
        self.startState = None
        self.endStates = []

    def add_state(self, name, handler, end_state=0):
        name = name.upper()
        self.handlers[name] = handler
        if end_state:
            self.endStates.append(name)

    def set_start(self, name):
        self.startState = name.upper()

    def run(self, cargo):
        try:
            handler = self.handlers[self.startState]
        except:
```

```
            raise InitializationError("must call
.set_start() before .run()")

        if not self.endStates:

            raise  InitializationError("at least one state
must be an end_state")



        while True:

            (newState, cargo) = handler(cargo)

            if newState.upper() in self.endStates:

                print("reached ", newState)

                break

            else:

                handler = self.handlers[newState.upper()]
```

```
# For our application
positive_adjectives = ["great","super", "fun",
"entertaining", "easy"]
negative_adjectives = ["boring", "difficult", "ugly", "bad"]
neutral_adjectives = ["ok","fair","unbaised","simple"]

def start_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else
(txt,"")
    if word == "Python":
        newState = "Python_state"
    else:
        newState = "error_state"
```

```
    return (newState, txt)

def python_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else
(txt,"")
    if word == "is":
        newState = "is_state"
    else:
        newState = "error_state"
    return (newState, txt)

def is_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else
(txt,"")
    if word == "not":
        newState = "not_state"
    elif word in positive_adjectives:
        newState = "pos_state"
    elif word in negative_adjectives:
        newState = "neg_state"
    elif word in neutral_adjectives:
        newState = "neutral_state"
    else:
        newState = "error_state"
    return (newState, txt)

def not_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else
(txt,"")
    if word in positive_adjectives:
        newState = "neg_state"
```

```
    elif word in negative_adjectives:
        newState = "pos_state"
    elif word in neutral_adjectives:
        new_state = "neg_state"
    else:
        newState = "error_state"
    return ( newState,txt)


def neg_state(txt):
    print("Hallo")
    return ("neg_state", "")


m = StateMachine()
m.add_state("Start", start_transitions)
m.add_state("Python_state", python_state_transitions)
m.add_state("is_state", is_state_transitions)
m.add_state("not_state", not_state_transitions)
m.add_state("neg_state", None, end_state=1)
m.add_state("pos_state", None, end_state=1)
m.add_state("neutral_state",None, end_state=1)
m.add_state("error_state", None, end_state=1)
m.set_start("Start")
m.run("Python is boring")
m.run("Python is not difficult")
m.run("Python is fair")
```

OUTPUT

```
reached  neg_state
reached  pos_state
reached  neutral_state
```

Coding Images

```
class StateMachine:

    def __init__(self):
        self.handlers = {}
        self.startState = None
        self.endStates = []

    def add_state(self, name, handler, end_state=0):
        name = name.upper()
        self.handlers[name] = handler
        if end_state:
            self.endStates.append(name)

    def set_start(self, name):
        self.startState = name.upper()

    def run(self, cargo):
        try:
            handler = self.handlers[self.startState]
        except:
            raise InitializationError("must call .set_start() before .run()")
        if not self.endStates:
            raise  InitializationError("at least one state must be an end_state")

        while True:
            (newState, cargo) = handler(cargo)
            if newState.upper() in self.endStates:
                print("reached ", newState)
                break
            else:
                handler = self.handlers[newState.upper()]
```

```
positive_adjectives = ["great","super", "fun", "entertaining", "easy"]
negative_adjectives = ["boring", "difficult", "ugly", "bad"]
neutral_adjectives = ["ok","fair","unbaised","simple"]

def start_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "Python":
        newState = "Python_state"
    else:
        newState = "error_state"
    return (newState, txt)

def python_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "is":
        newState = "is_state"
    else:
        newState = "error_state"
    return (newState, txt)

def is_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "not":
        newState = "not_state"
    elif word in positive_adjectives:
        newState = "pos_state"
    elif word in negative_adjectives:
        newState = "neg_state"
    elif word in neutral_adjectives:
        newState = "neutral_state"
    else:
        newState = "error_state"
    return (newState, txt)
```

```
def not_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word in positive_adjectives:
        newState = "neg_state"
    elif word in negative_adjectives:
        newState = "pos_state"
    elif word in neutral_adjectives:
        new_state = "neg_state"
    else:
        newState = "error_state"
    return ( newState,txt)


def neg_state(txt):
    print("Hallo")
    return ("neg_state", "")


m = StateMachine()
m.add_state("Start", start_transitions)
m.add_state("Python_state", python_state_transitions)
m.add_state("is_state", is_state_transitions)
m.add_state("not_state", not_state_transitions)

m.add_state("neg_state", None, end_state=1)
m.add_state("pos_state", None, end_state=1)
m.add_state("neutral_state",None, end_state=1)
m.add_state("error_state", None, end_state=1)
m.set_start("Start")
strr = input("Enter the String with rules: ")
m.run(strr)
# m.run("Python is not difficult")
# m.run("Python is fair")
```

Output Image(Execution)

```
reached  neg_state
reached  pos_state
reached  neutral_state
```

Transition Diagrams for each input

Python is boring

```
Enter the String with rules: Python is boring
Python_state-> is_state-> neg_state -> reached  neg_state
```

Python is not difficult

```
Enter the String with rules: Python is not difficult
Python_state-> is_state-> not_state -> reached  pos_state
```

Python is fair

```
Enter the String with rules: Python is fair
Python_state-> is_state-> neutral_state -> reached  neutral_state
```

Perl is fair

```
Enter the String with rules: Perl is fair
error_state-> reached  error_state
```
