PastaNFA – Final Project

Ryland Edwards, Riley Meyerkorth, Colin Treanor

University of Kansas (KU)
EECS 510

Table of Contents

Definitions

Some of the terms used in this document and elsewhere translate to vocabulary used in Automata Theory. These terms and their corresponding Automata Theory meanings are as follows:

- **Dish**: “valid string”
- **Ingredient**: “symbol”
- **Available Ingredients**: “alphabet”
- **Pasta**: “symbols A through C” (see Alphabet)
- **Sauce**: “symbols D through F” (see Alphabet)
- **Addon**: “symbols G through J” (see Alphabet)

Part 1 – Design a Formal Language

Purpose

The purpose of our language is to validate strings that represent different types of pasta preparations while ensuring adherence to defined constraints. This allows for easy parsing, validation, and computations.

Intent

The intent of this language is to give a framework for understanding a given string representing ingredients for a pasta dish and then in return, generating the steps to put together the dish. By defining a specific alphabet and clear rules, this language can validate whether an input string corresponds to a reasonable pasta dish or not.

Structure

Alphabet

The language's alphabet is composed of symbols that are used to represent the ingredients that would be used in a pasta dish. Each letter will correspond to an element commonly found in recipes. Our alphabet is as follows:

- A-C: Represents pastas (spaghetti, fettuccini, and macaroni)
- D-F: Represents sauces (marina, alfredo, and pesto)
- G-J: Represent additional ingredients/addons (ground meat, chicken, olive oil, garlic, onions, broccoli)
- ~: Used as an input-safe null/lambda (λ) symbol.

Rules and Patterns

The following rules and patterns are to be adhered to when creating a dish:

1. The input string must always include a singular type of pasta. This is the central focus of the dish, and it won't be possible to make a “pasta” dish without it.
2. Only a single type of pasta is allowed in any dish.
3. The pasta must come first in any pasta dish, with sauces and addons being added on *top*.

4. Sauces and addons are optional in any given dish.
5. Any number of sauces and/or addons may be added.
6. All letters must be uppercase to keep consistency and clarity in the interpretation.

Example:
- **ADJI** (spaghetti noodles with marinara, garlic, and onion)
- **CF** (macaroni noodles with Pesto sauce)
- **BEH (**fettuccine noodles with alfredo sauce and olive oil)
- **A** (plain spaghetti noodles)

Part 2 – Grammar

We decided to create our machine using a **non-deterministic finite automaton** (or **NFA**).

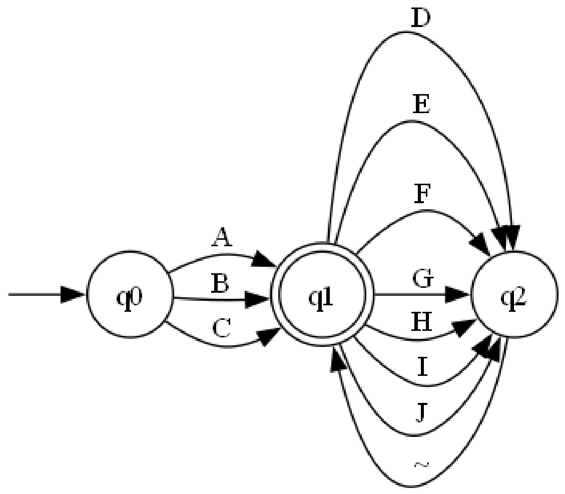
Production Rules

The formal grammar for our language is as follows:

$$\begin{aligned} S &\rightarrow PX \\ P &\rightarrow A \mid B \mid C \\ X &\rightarrow D \mid E \mid F \mid G \mid H \mid I \mid J \mid \sim \end{aligned}$$

where S is the starting state, P is the pasta, and X is an optional ingredient (sauce/addon).

Part 3 – Automaton



States

$$q_0, q_1, q_2$$

Start State: q_0

Accepting States: $\{q_1\}$

Transitions

$$q_0\ A\ q_1$$
$$q_0\ B\ q_1$$
$$q_0\ C\ q_1$$
$$q_1\ D\ q_2$$
$$q_1\ E\ q_2$$
$$q_1\ F\ q_2$$
$$q_1\ G\ q_2$$
$$q_1\ H\ q_2$$
$$q_1\ I\ q_2$$
$$q_1\ J\ q_2$$
$$q_2 \sim q_2$$

Part 4 – Data Structure

For installation instructions, programming language, and platform information, refer to the Installation Instructions section of this document.

Schema Description

NFA

The root/parent data structure that represents our automata in memory is the NFA class. In our project files, the source code for this data structure is found at /src/automata/nfa.py.

Properties

- states: set[State]
  - The states of the NFA represented in set notation as State objects.
- symbols: set[str]
  - The symbols/alphabet of the NFA represented in set notation.
- start_state: State
  - The start state of the NFA represented as a State object.
- accept_states: set[State]
  - The accepting/end state(s) of the NFA represented in set-notation as State objects.
- transitions: dict[State, dict[str, set[State]]
  - The transitions of the entire NFA represented as a dictionary/map.
  - Key: A state of the NFA represented as a State object.
  - Value: A dictionary/map for that specific state that represents each symbol's transitions from that state.
    - Key: A symbol that is used to transition to the next state
    - Value: A set of State objects representing the to/destination states from the symbol's transition.

- debug: bool
    - A flag that enables/disables debug printing.

Public Methods

- find_state(name: str) -> State
    - Finds and returns the state of the given name.
    - name: the name of the state
- accept(w: str) -> AcceptResult
    - Tests the acceptance of w and returns an AcceptResult object.
    - w: an input string
- print() -> None
    - Prints information about the NFA. This includes states, symbols, the start state, the accepting states, and the transitions.
- generate_diagram(output_dir: str, output_name: str) -> None
    - Generates a diagram of the NFA using Graphviz and then displays it.
    - output_dir: the directory that the Graphviz image should be saved to
    - output_name: the name of the files produced by Graphviz with no extension.

Private Methods

- _lambda_closure(states: set[State]) -> set[State]
    - Computes the lambda closure of the given set of states and returns the updated set of states.
    - states: the set of current states to collapse
- _lambda_closure_with_path(states: set[State], path: list[Transition]) -> tuple[set[State], list[Transition]]
    - Computes the lambda closure of the given set of states **while keeping track of lambda transitions**. Returns a tuple of the updated set of states and the updated path.
    - states: the set of current states to collapse
    - path: the current list of Transition objects to update
- _get_states_as_strings() -> list[str]
    - Gets the list of State objects as a list of strings instead (mainly for Graphviz)
- _get_accepting_states_as_strings() -> list[str]
    - Gets the list of accepting State objects as a list of strings instead (mainly for Graphviz)
- _get_transitions_as_string() -> dict[str, dict[str, set[str]]]
    - Gets a copy of the transitions dictionary where State objects are swapped for strings (mainly for Graphviz)
- _parse_input_file(filename: str) -> bool:
    - Parses the passed filename as an NFA.
    - filename: the filename (**with extension**) to parse as an NFA.

Implemented Dunder/Magic Methods

- __init__(filename: str, debug: bool)

Constants

- LAMBDA_SYMBOL: str
  - The lambda symbol that is to be used. As previously mentioned, our lambda symbol is represented as the tilde (~).

PastaNFA

The sub/child data structure of the NFA class that helps to format the steps of pasta dish creation is the PastaNFA class. It inherits all of the previous properties and methods of the NFA class. In our project files, the source code for this data structure is found at /src/automata/pasta_nfa.py.

Properties

- ingredients: dict[str, str]
  - A dictionary representing the mapping between the symbols and their ingredient counterpart.
  - Key: a symbol represented as a string
  - Value: an ingredient represented as a string

Public Methods

- print_recipe(result: AcceptResult) -> None
  - Prints the steps of a given AcceptResult as a formatted recipe

Implemented Dunder/Magic Methods

- __init__(filename: str, debug: bool, ingredients: list[str])

Constants

NOTE: Changes/additions to these constants will change the mapping of ingredients to symbols. All of the following constants need to be **in the same order** as the alphabet/symbols.

- PASTA: list[str]
  - The formatted names of each of the pasta/noodle types **in-order**.
- SAUCES: list[str]
  - The formatted names of each of the sauces **in-order**.
- ADDONS: list[str]
  - The formatted names of each of the addons **in-order**.
- INGREDIENTS: list[str]
  - The combination of PASTA, SAUCES, and ADDONS. *Should be the same length as the alphabet (minus the lambda (~) symbol)*.

State

The dataclass that represents automata states in an OOP design is the State class. In our project files, the source code for this class is found at /src/automata/models/state.py.

Properties

- name: str
  - The name of the state as a string.

Implemented Dunder/Magic Methods

- __init__(name: str)
- __str__()
- __repr__()

Transition

The dataclass that represents automata transitions in an OOP design is the Transition class. In our project files, the source code for this class is found at /src/automata/models/transition.py.

Properties

- from_state: State
    - The “left” or “outbound” state of a transition
- symbol: str
    - The symbol to transition on
- to_state: State
    - The “right” or “inbound” state of a transition

Implemented Dunder/Magic Methods

- __repr__()
- __str__()
- __len__()

AcceptResult

The dataclass that encapsulates the return values from the `accept` method is the AcceptResult class. In our project files, the source code for this class is found at /src/automata/models/acceptresult.py.

Properties

- accepted: bool
    - “True” if the input string was accepted, and “False” otherwise.
- path: list[Transition]
    - Represents the path taken by the input string as a list of Transition objects.

Implemented Dunder/Magic Methods

- __repr__()
- __str__()
- __bool__()
- __len__()

Automata – Schema Form

Our data structure parses an NFA into memory through reading an input file. The parsing rules are as follows:

Line 1: a space-separated list of states

Line 2: a space-separated list of symbols

Line 3: a singular start state

Line 4: a space-separated list of accept states

Line 5+: a space-separated list of 3 components:

- From State: the outbound/current state of the transition
- Symbol: the symbol to transition on
- To State: the inbound state of the transition

For examples or to learn more, visit the /src/machine folder and corresponding documentation.

nfa.txt

Our specific NFA has the following input file:

```
q0 q1 q2
A B C D E F G H I J ~
q0
q1
q0 A q1
q0 B q1
q0 C q1
q1 D q2
q1 E q2
q1 F q2
q1 G q2
q1 H q2
q1 I q2
q1 J q2
q2 ~ q1
```

Part 5 – Testing

This section will detail how to run/test our program, as well as outlining how the machine checks the acceptance of a string.

Installation Instructions

Overall, our code was written in Python 3.12.1 and tested on Windows 11 environments. We used GitHub as our code collaboration platform, and the repository can be found at this link.

Dependencies

- pillow (GitHub)
    - Version: 10.3.0
- automathon (GitHub)
    - Version: 0.0.15

In general, to install these packages, you can run the following commands:

```
pip install pillow

pip install automathon
```

However, using a virtual env would be recommended for installation and usage. A venv env or Conda env would be best to use.

Conda:

```
conda create –name myenv python=3.12.1

conda activate myenv
```

To install dependencies, change pip into conda:

```
conda install pillow

conda install automathon
```

Venv env:

```
python3.12 –m venv myvenv
```

Activate on Mac/Linux:

```
source myvenv/bin/activate
```

Activate on Windows

```
myvenv/Scripts/activate.bat
```

Optional Dependencies

To generate a visual graph of our NFA, the program Graphviz is required to be installed and added to your system PATH. This is optional, however, as it does not affect string parsing nor acceptance.

Usage and Options

When in the root directory of the project folder, the usage of the program is as follows:

```
python src/main.py [-h] [-m] [-i] [-d] [-p] [-f]
```

- -h, --help
    - Show usage and options
    - Optional
- -m, --machine_path
    - Path to the NFA/machine file to load.
    - Default: “machines/nfa.txt”
    - Optional
- -i, --input_string
    - Input string to test on the machine.
    - Default: None
    - Optional
- -d, --debug
    - Enable/disable debug output.
    - Default: false
    - Optional
- -p, --print_path
    - Enable/disable printing the path/trace of the input.
    - Default: true
    - Optional
- -f, --formatted_steps
    - Enable/disable printing the formatted steps of the input.
    - Default: true
    - Optional

Function Definition

We have defined and implemented a function “accept(A, w)” in the /src/main.py file for assessing our NFA. It is an abstraction of the class-defined “accept(w)” method, thus the main functionality is located there instead.

Here is that function/method implementation from the NFA data structure/class:

```python
def accept(self, w: str) -> AcceptResult:
    """
    Checks if the given string is accepted by the NFA.
    Returns an AcceptResult object holding the acceptance results.
    """
    # Initialize return value
    result: AcceptResult = AcceptResult(False, [])

    # Start with lambda closure of the start state
    current_states, result.path = self._lambda_closure_with_path({self.start_state}, result.path)

    # For each symbol in the input string
    for symbol in w:
    next_states = set()

        # For each state currently reachable, add transitions on 'symbol' if they exist
        for state in current_states:
            if state.name in self.transitions and symbol in self.transitions[state.name]:
                # Record transition
                result.path.append(Transition(state, symbol, list(self.transitions[state.name][symbol])[0]))

                # Update next states set
                next_states.update(self.transitions[state.name][symbol])

        # After consuming 'symbol', take lambda closure of the set of reachable states
        current_states, result.path = self._lambda_closure_with_path(next_states, result.path)

    # Check if any current state is an accept state
    for s in current_states:
        if s in self.accept_states:
            result.accepted = True
            break

    # Return the result
    return result
```