Utilisation of Finite Automata to a Basic Calculator

14247922

41080

Finite Automata

Finite Automaton is a classical computing model to recognize patterns of the input pulled to the set of states. The input to a Finite Automation is the words from the alphabet Σ and takes its alphabet to the sequential states. To move from certain state to the other, it needs to meet the condition given by the transition function. Once the status of the input ends up reaching to the final state F, it will then be assumed as the automaton **Accepted** the input of the alphabet. If the input cannot reach to the final state, it will be then assumed as **Rejected**.

As the assignment program requires the deterministic finite automaton (DFA), this report will not cover any theory nor information of non-deterministic finite automaton.

Deterministic Finite Automaton

Deterministic Finite Automaton, or DFA is one of the most well-known automata that is commonly used in Lexical Analysis. DFA inherits the same concept of finite automata, as it accepts the input if the process reaches and ends in the accept state. As well as the finite automata, DFA also rejects all other cases.

The design of the DFA is often clarified as 2 steps. It is required to initially design the state sets including start and final states, and transition design to finalise the step.

Within these features, the basic calculator could be successfully applied with DFA.

Five Tuples

Formally, DFA is consisting of 5 tuples, which are the following:

1. Q, is a finite state of internal states
2. $\sum$, is a finite set of symbols called alphabet
3. $\delta: Q \times \sum \rightarrow Q$, is a total function called transition function
4. $q_0 \in Q$, is a start state
5. $F \subseteq Q$, is a final state

All of these are mandatory to be used to construct a DFA.

Applying DFA into a Basic Calculator

To start constructing a basic calculator using DFA, a hand drawn DFA was the first step that need to be taken to move onto the actual coding. This is shown in Figure 1 below:

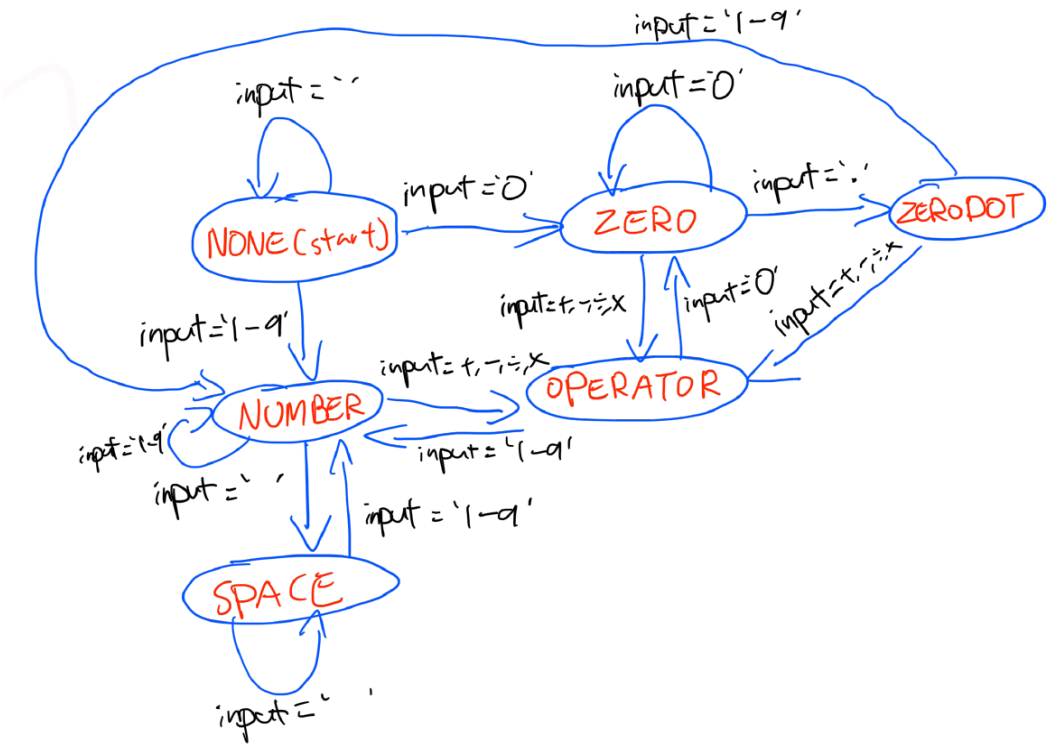

Figure 1: Hand-drawn Deterministic Finite Automaton sketch

Five Tuples are utilised above, as the $q_0 \in Q$ is shown as starting point from the State of 'NONE', and has two paths of states Q, 'ZERO' and 'NUMBER'. This is due to the limitation of calculator as it can only be accepted when the first character of $\sum$ is either a natural number or a zero. Unless it will come across as NumberException and ExpressionException (ex, starting with dot, or any operators). These accepting conditions are the transition functions; $\delta: Q \times \sum \rightarrow Q$. As zero can generate a decimal, it is excluded from the state of number. The state 'ZERODOT' is only for the case of decimal places and can only be reached through the state of 'ZERO'. The state 'SPACE' is for the case when there's a gap between input numbers such as "11 332". Through this state, all the incorrect format of numbers are filtered and not registered as the exception. The state 'OPERATOR' can be used from all the other states besides the special cases like 'NONE', which is the starting state $q_0 \in Q$, and 'SPACE', which is the prevention for the number input errors. All Q, including 'NONE' can be a final state, $F \subseteq Q$ as this is just a verification section to check whether the input is validated or not.

The drawn DFA must be initially executed from the state, 'NONE' and can be sent to either 'ZERO' or 'NUMBER' state. Other cases such as having operator is not specified above as all exception states are not included into this DFA. This is for the visibility of the DFA, and for the easier understanding of the concept that need to be applied into the code later. Thus, any cases that are not specified, are going to either 'NumberException', or 'ExpressionException'.

The situated 'NumberException' cases are when it does not meet the string of digit format, decimal point errors and decimal that doesn't consist of '0.[something]' (such as 1.3 or 2.0).

The situated 'ExpressionException' cases are when the input doesn't have a valid expression, or the operator follows up with another operator in a row (such as ++-).

Transferring Hand-Drawn DFA to the Code

Since all basic concept of the DFA is already formed above, it is left with how to apply this logic into the code. To do this, Enum type with switch statement were employed to create the DFA in java code.

```
private enum State {
    NONE, ZERO, ZERODOT, NUMBER, OPERATION, SPACE
}
```

```
switch (state) {
case NONE:
    state = State.ZERO;
    break;
```

Figure 2: Enum types / switch-case statement

As same as the hand drawn DFA, the states were only consisting of the following 6, 'NONE', 'ZERO', 'ZERODOT', 'NUMBER', OPERATION, and 'SPACE'. These states are used to determine the input's status; when the input contains a number, the state 'NUMBER' will be granted on that time.

To separately check each character of the input to check the validity, for loop is employed by comparing the size of the new variable, 'i,' which increments every loop, with the length size of the input. Through this, the function charAt(i) can be used to check each character of the input at the time.

```
for (int i = 0; i < input.length(); i++)
```

Figure 3: For loop

The IF statement is utilised to differentiate and grant the states on the specific time:

```
if (input.charAt(i) == '0') {
    switch (state) {
    case NONE:
        state = State.ZERO;
        break;
    }
}
```

Figure 4: IF statement for the case '0'

```
else if (tkn.typeOf(input.charAt(i)) == tkn.typeOf('+'))
    switch (state) {
    case NONE:
        throw new ExpressionException();
    case NUMBER:
        state = State.OPERATION;
        break;
    case OPERATION:
        throw new ExpressionException();
    case SPACE:
        state = State.OPERATION;
        break;
```

Figure 5: IF statement for the case '+'

Within the help of for loop in Figure 3, if statement in Figure 4 can use the function, charAt(i), to separately check whether the character contains certain case. If it does, it will grant the state of the certain case, and this can further use for the following character.

For example, in Figure 4, if the input's first character contains '0', it will grant the current state as 'ZERO' and will stop the loop for this case; increments the 'i' by 1 and repeats the step.

As shown above on Figure 5, exceptions can be thrown at any time when the input contains the incorrect format. When the input starts with any operators; in this case, plus, it will throw the 'ExpressionException' as it will get detected by the IF statement, go through the case of 'NONE' (the starting state), then reach to the 'ExpressionException' code.

This is not specified in hand drawn DFA but used in code. This is to simplify the code to make it more efficient by not having extra states, such as 'EXPRESSION' or 'NUMBER'.

Applying token to DFA conditions

Tokenising the characters to the certain types are used to improve the automaton on the java code. As the class 'Token' is already given, by creating the object of the class and using the function, typeOf() can make the step of checking the condition more straightforwardly. It also simplifies the step of specifying every condition that need to be met. For example, when checking whether the character is a number or not, rather than comparing with every single number like '1', '2', … '9', it's more efficient to check its type and compare with the type of 'NUMBER'. This is shown on below:

```
else if (input.charAt(i) != '0' && tkn.typeOf(input.charAt(i)) == tkn.typeOf('1')){
```

Figure 6: Utilising Token class

Dot state

As dot is not registered as a type in 'Token' class, it was necessary to use the if statement with " == '.' " to check whether the character is a dot or not. Further, when granting the state of 'ZERODOT', another IF statement is used to check the last digit of the input is a number and grants the state. This is to verify that the space is not included at the end of the input, which should cause the 'ExpressionException'.

Challenges

Initially, the few of the previous Enum were used to represent the unnecessary states. These include, "ZEROSPACE", "OPERATIONDOT", and "NUMBERDOT". This is due to the misunderstanding of DFA concept, where the exceptions do not need to be specified as states. This is resolved by removing all of them and is finalised on Figure 2.

Additionally, differentiating between space and numbers encountered as a problem once mots of the concept were finalised. The case '66 6' was not detecting as Expression Exception, thus the case 'SPACE' was added to resolve the problem. The concept is that when the If statement detects the space as a character, it will grant the case of 'SPACE', and will throw the ExpressionException when it goes through the case of 'NUMBER' afterwards. These are shown below:

```
if (input.charAt(i) == ' ') {
    switch (state) {
    case NUMBER:
        state = State.SPACE;
    }
}
```

```
case SPACE:
    throw new ExpressionException();
}
```

Figure 7: 'SPACE' state

However, when the space is between any operators, it will grant the state of 'OPERATOR' as it can have spaces between number and operator.

Also, the case 'SPACE' will only be triggered when the case 'NUMBER' was granted previously, meaning that starting from the space does not affect the other DFA cases.