How I went from Specs to DFA to Code

Noah Ostle 24740589

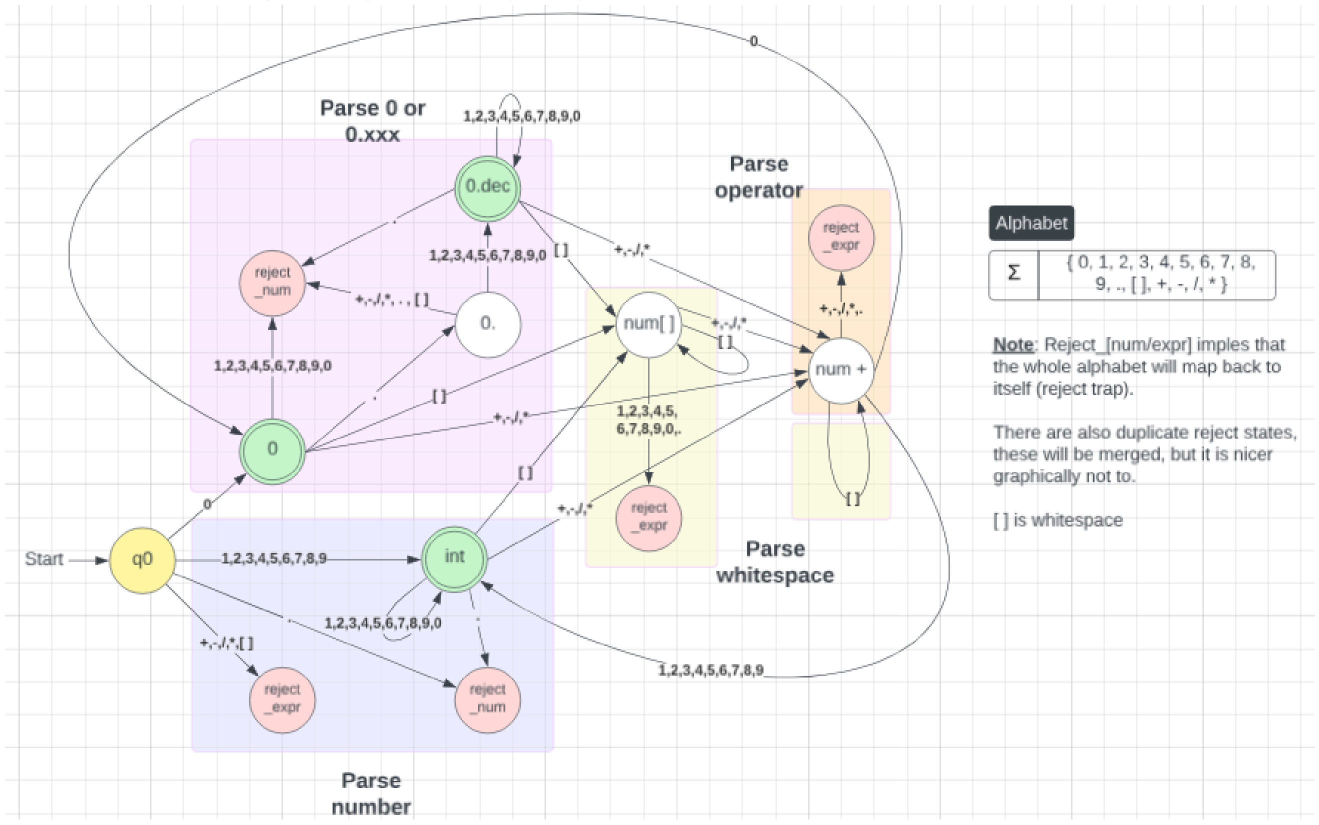
DFA Construction Process

The first step in my process was carefully reading the brief, and analysing the Java skeleton code to determine what goals I had to accomplish with both my code and my DFA. I opened LucidChart to begin drafting my DFA, and wrote down several key notes from my analysis, including the alphabet, some examples of valid and non valid expressions, as well as some of the types of tokens and errors I would have to parse.

I quickly realised that at a high level, I would essentially be parsing for a number, then parsing for an operator, then parsing for a number etc., accepting when the string ended in a valid number.

I began by just describing the process of how I would parse the example strings if I was a DFA in verbose English. This really helped to get the ball rolling, but led to some inefficiencies that had to be cleaned up, which we will see later.

After I had a general idea of how my DFA would work, I broke it down into different parts, such as recognizing a number, or a 0, or a decimal, or an operator, and I drew highlighted boxes around each logical segment of my program.



Note: Reject_[num/expr] imples that the whole alphabet will map back to itself (reject trap).

There are also duplicate reject states, these will be merged, but it is nicer graphically not to.

[ ] is whitespace

Fig. 1 - DFA

Doing this allowed me to abstract away some of the complexity, and get a much better idea of how each part of my DFA worked together without worrying about the specific details every time I analysed its flow, and even allowed me to really easily reuse parts of my DFA, almost like a function in traditional programming.

Working graphically actually really helped me to realise when things were inefficient. During the process of designing my DFA, I would describe things in my head in english, and just drew down the logical version of that in symbols. This did help me get started, but lots of times led to repeated sections, where I had two or more of the same 'subroutine' (eg. parsing whitespace after an Int, as well as parsing whitespace after a Decimal using a separate set of states).

Designing my DFA graphically, and highlighting each logical subroutine really helped me realise when there was a pattern between two different areas of my DFA, and a lot of the time, I could merge both these sections into one singular subroutine without loss of information, removing duplicate functionality, and decreasing the number of states, and overall complexity of my DFA.

For example, rather than parsing whitespace on Ints and Decimals separately, I could simply point both transition functions to a shared and more general ***number*** whitespace parsing subroutine (as seen in num[] above), meaning I could remove one of the duplicate whitespace parsing blocks.

This was an iterative process of enhancing, stepping through, and fixing my design, until I was confident it was fairly accurate, and ready for implementation.

How my DFA works

My DFA works by first either parsing for an Int, which is a string of numbers only, or a Zero/Decimal, which is either a zero or a zero, then a decimal point, then a string of numbers. Both of these 'subroutines', highlighted in blue and purple respectively, have parsing checks for invalid numbers.

Once you have either a valid 0, decimal, or int, you can either stop there, add whitespace, or add an operator. If you add a whitespace, all 3 options transition to one common whitespace checker 'subroutine', that ensures the expression is a number, followed by a whitespace, followed by an operator, by rejecting anything that isn't an operator or a digit.

Either from here, or from any of the 3 states that indicate a valid number, you can add an operator, followed by more whitespace.

After this point, any operator causing an invalid expression will be rejected, and any digit will point back around to the start, where upon entering another valid number, the string will be accepted again.

This means that any sequence of numbers, separated by operators (and optional whitespace) will be accepted, as per the brief.

Implementation techniques and challenges

After doing some thinking, and a little bit of research, I chose to Implement my DFA design using a nested hash map for two reasons. First, the switch/if conditional block style seemed very messy and confusing, especially for a DFA of this size, and secondly, if I did choose this method, I would be able to print the hashmap out like a transition function table.

My hash map design involved a master map (you can think of this as the transition function table). This master map would accept a state, and return another hashmap representing the transition function for that state.

For Example, you would put the “start” state into the master map, and it would return a ‘startTransitions’ map, which would have entries for every character of the alphabet, mapping char ‘0’ to State.ZERO, or char ‘1’ to State.INT.

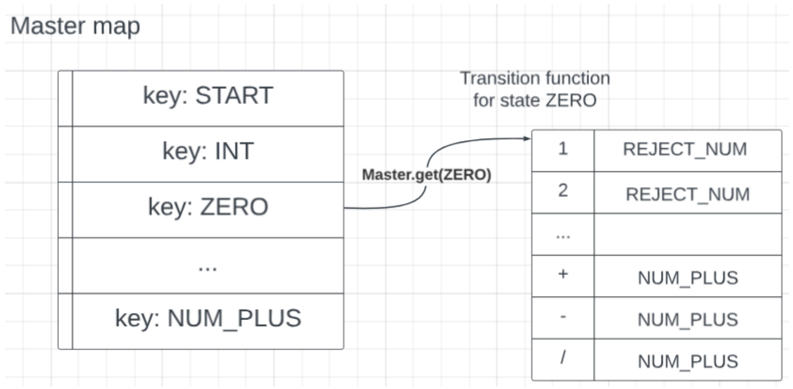
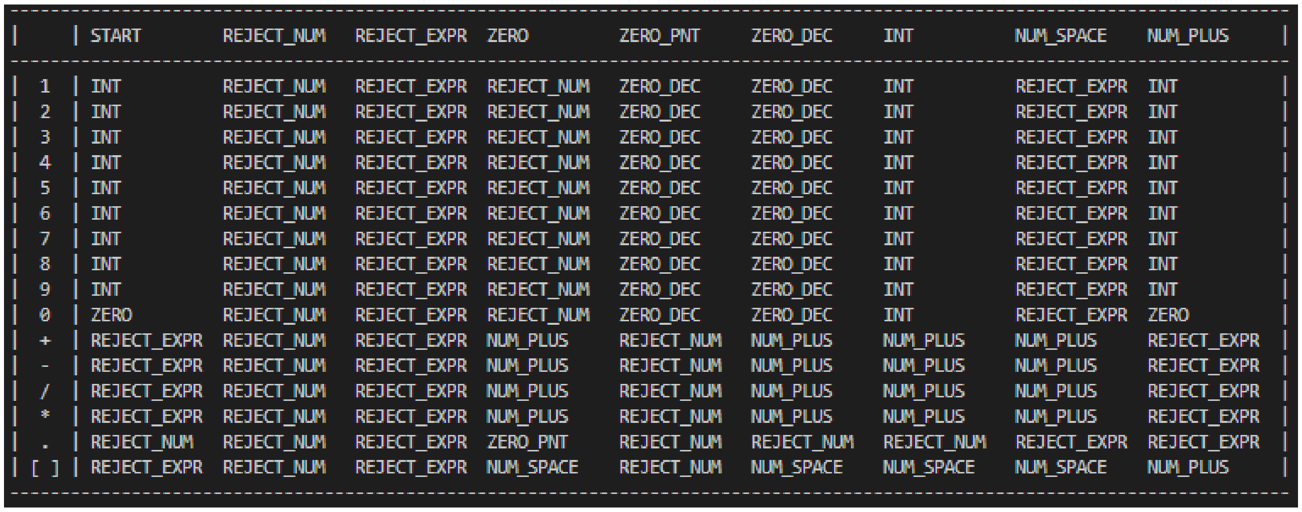


Fig. 2 - Hashmap

I encoded the full dfa logic into my hashmap manually, being very careful to not miss anything from my draft diagram. Once I was happy that all of the transition functions and states were set up correctly, I wrote a very simple function to print the transition table from the hashmap.



```
-------------------------------------------------------------------------------------------------------------------------------
|     | START        REJECT_NUM   REJECT_EXPR  ZERO         ZERO_PNT     ZERO_DEC     INT          NUM_SPACE    NUM_PLUS     |
-------------------------------------------------------------------------------------------------------------------------------
|  1  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  2  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  3  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  4  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  5  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  6  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  7  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  8  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  9  | INT          REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  INT          |
|  0  | ZERO         REJECT_NUM   REJECT_EXPR  REJECT_NUM   ZERO_DEC     ZERO_DEC     INT          REJECT_EXPR  ZERO         |
|  +  | REJECT_EXPR  REJECT_NUM   REJECT_EXPR  NUM_PLUS     REJECT_NUM   NUM_PLUS     NUM_PLUS     NUM_PLUS     REJECT_EXPR  |
|  -  | REJECT_EXPR  REJECT_NUM   REJECT_EXPR  NUM_PLUS     REJECT_NUM   NUM_PLUS     NUM_PLUS     NUM_PLUS     REJECT_EXPR  |
|  /  | REJECT_EXPR  REJECT_NUM   REJECT_EXPR  NUM_PLUS     REJECT_NUM   NUM_PLUS     NUM_PLUS     NUM_PLUS     REJECT_EXPR  |
|  *  | REJECT_EXPR  REJECT_NUM   REJECT_EXPR  NUM_PLUS     REJECT_NUM   NUM_PLUS     NUM_PLUS     NUM_PLUS     REJECT_EXPR  |
|  .  | REJECT_NUM   REJECT_NUM   REJECT_EXPR  ZERO_PNT     REJECT_NUM   REJECT_NUM   REJECT_NUM   REJECT_EXPR  REJECT_EXPR  |
| [ ] | REJECT_EXPR  REJECT_NUM   REJECT_EXPR  NUM_SPACE    REJECT_NUM   NUM_SPACE    NUM_SPACE    NUM_SPACE    NUM_PLUS     |
-------------------------------------------------------------------------------------------------------------------------------
```

Fig. 3 - Transition Table

This table was actually very valuable to me, as when I first printed it out, there were states missing from it, which were immediately obvious because they were completely blank. This indicated to me places where I had forgotten to assign a state to every character for a particular states transition function.

I fixed these, and made sure that there was exactly one state for every character, so that I could begin tokenizing and throwing exceptions.

Technical challenges - Hashmap vs Theoretical Concept

Throwing exceptions was easy enough, as I simply had to check after every character was consumed whether or not it ended up in my reject traps, then depending on which one it ended up in (REJECT_NUM or REJECT_EXPR), I could throw the appropriate exception.

Tokenizing was slightly harder, due to the conflict between the static nature of my code implementation, and the more dynamic nature of the theoretical concept of a DFA. This is because it relies not only on the state you are in, but also how you got there, in other words, whether you tokenize, how you tokenize, and what you tokenize, is all dependent on the transition function you used to get there.

This would be fine in a switch/if implementation, as the various branches nicely represent a transition*,* however I found out that there is a tradeoff for my hashmap implementation; it does not logically transition states.

This is because there is a key difference in the actual process of state transitions between my implementation and a DFA. Both the DFA, and the switch/if method can be thought of like a branching tree (this is what fig. 1 is), where a string will follow a directed path depending on its letters and the transition functions of each state. However, my hashmap is not a tree, it is a map, so there is no notion of direction as there would be in a directed graph.

This means that for my hashmap, there is no directed transition function, I can change the state by performing a lookup and setting the state variable, but unless I code one in, there is no notion of an intermediary ‘transition’, only the state before, and the state once a transition has occurred.

This did initially freak me out for a couple of minutes, as I realised I may have to start over and get rid of my beautiful O(1) lookup speed hashmap, but I realised that there was a pretty convenient solution.

I realised that I could make an analogue to branching/transitioning by calculating the next state based on a hashmap lookup, and then waiting before I set the state. This introduces a way for my implementation to be between two states at once, in other words, it allows for execution of code with knowledge of the current transition function.

Rather than before, where I have the initial state, perform a lookup, and overwrite that state with a new one, preventing me from determining where I have come from, i.e. the current transition function, my code now has the initial state, performs a lookup, and before switching to the new state, can do actions such as tokenizing with the required knowledge of what transition function it is in. After it is done, my program can simply set the state to the one it got from the lookup, and continue as normal.

After overcoming this issue, I had all of the main features of my program in place, and it was just down to some minor tweaks I had to perform to fix some logical mistakes I made in order to get my code to pass all of the unit tests.