Whitney Henderson
Nathaniel Fan
Matthew Compton
Ben Bariteau

Scanner Generator

Assumptions:

We made several base assumptions in order to build our scanner generator. Most of these assumptions revolve around the grammar that will be parsed. First we assume the line between tokens and grammar is empty string or a comment, and no combination of the two. We assume there will only be comments at the beginning of the character classes, between the characters and tokens, and perhaps at the end of the tokens. Our code does not allow for inline comments or comments within the characters section or the tokens section. We assume that all tokens have a $ at their beginning, and that they only contain Capital letters and or a hyphen. We also assume that the rules we outlined for Regular expressions in "Created Regex Rules" are the only rules we will encounter.

Reading the Spec

In the main file, our program starts by pre-parsing the input file: first for characters, then for tokens. The pre-parsers for Tokens and Characters are essentially the same. Both start by checking that the string is acceptable using the regex pattern "\\$[A-Z-]+". We then take the initial character as the Character/Token and then remove the white space and take the rest as its definition. Our initial parser breaks the spec up in this manner so we can proceed with the real work.

We then parse the Tokens and Characters for real using an LL(1) parser. First we create a parsing table for the regex rules we developed. Then for every key in the token and character sets, we parse the grammar segment from that key using the regex table. The parser matches productions to rules and symbols and creates a parse tree for the each of the characters to tokens and both trees are retuned to the main functions. Both are then submitted to create NFAs.

How We Built NFAs

The system to create NFAs for the tokens and the system to create NFAs for the characters are the same. Using the parse trees we constructed previously, we parse through each character/token and recursively iterate through the parse trees, adding first the start state and proceeding from there to add each state and accept state in proper progression order. We parse through each key in the parse tree and if the key is not already included in the NFA, we add it. We then parse for its children using the parse trees and check if any of the children are accept states so we can mark them as such. It then concatenates the partial NFAs into full NFAs. Then we combine all the NFAs we have constructed into a single massive NFA.

Building DFAs

From this massive NFA we construct our DFA. We start by creating a new hash map to store the DFA in. We then parse for all states that can be reached from the start state via epsilon transitions. These make our set of states for our DFA start states. For each state set we extract the set of DFA states and epsilon states and place them into the new DFA states list, along with putting the transitions into a new Transitions list. We iterate over the resulting new states, checking for epsilon transitions to construct our DFA states, and checking the transitions for each state set. Once we've put all states and transitions into their respective tables we check for accept states, and add them to the map. We continue to parse through the other states adding them and their transitions to the map. We continue to iterate through this process until we have processed all NFAs and created a completed derivative DFA.

Testing

We run fairly basic tests. First our code must pass a regimen of test inputs, making sure that none of our constructed grammars and inputs stump our system. The test cases are included in the code, but the basic functions are listed below:

- set0 is a set with a simple grammar and a simple input
- set1 is a set with a simple grammar but a complex input
- set2 is a set with a complex grammar but a simple input
- set3 is a set with a complex grammar and a complex input
- set4 contains a grammar with a lot of comments
- set5 contains a grammar with a comment line instead of a blank line separator
- set6 is a set with no unnecessary whitespace characters
- set7 is a set with a lot of unnecessary whitespace

The test cases are parameterized so that issues encountered in one test case will not spread to the others. This allows for easier debugging. We also added several "invalid" test cases. Test cases that have grammar our generator shouldn't be able to process.

All testing cases are provided in the code base.

Problems We Encountered

Most of our issues were with understanding the parameters for the project and what tools we could and couldn't use, and how we were expected to solve each problem. There was a combination of seemingly vague construction parameters paired with strict grading parameters. As it is, we addressed all the problems assigned for the project and solved them to the best of our ability.