Name: Sean Rettig
Date: 2015-01-30

Milestone 2 Report

Handwritten Answers to Milestone Questions:

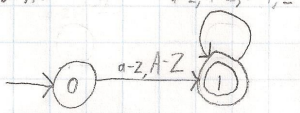
The types of tokens that are required include operators, ints, reals, strings, parentheses, and identifiers/keywords. The data structure I used for my tokens was a Token object that simply stored one of the above mentioned token types, and a value (such as the identifier itself in string form for an “id” token).

To create the lexer, I first designed NFAs/DFAs for each type of token that needed to be lexed. Then I combined them all into one big DFA, which could be translated to my lexer fairly easily, as the lexer simply stores state objects for each state of the DFA and traverses them. DFA diagrams:

Name: Sean Rettig
Date: 2015-01-30

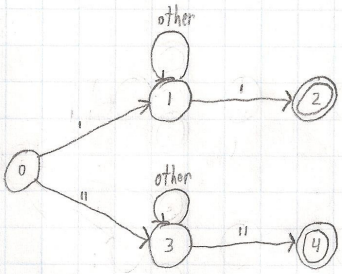
Identifier/keyword:

NFA/DFA:



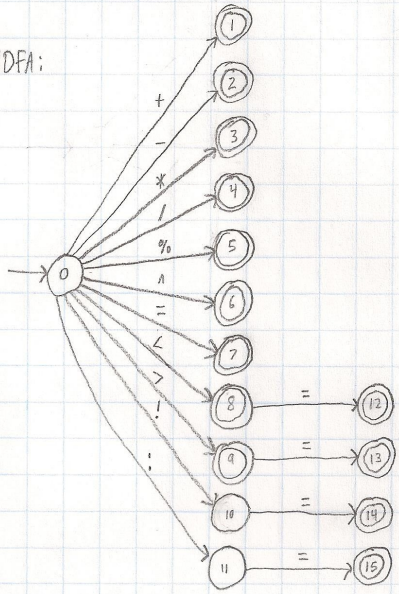
String:

NFA/DFA:



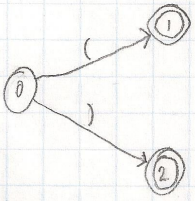
Operator:

NFA/DFA:



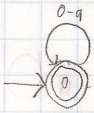
Parentheses:

NFA/DFA:



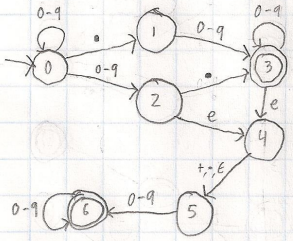
Integer:

NFA/DFA:

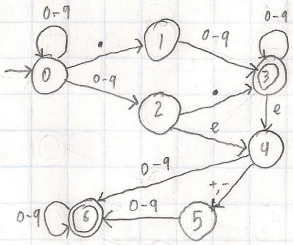


Real:

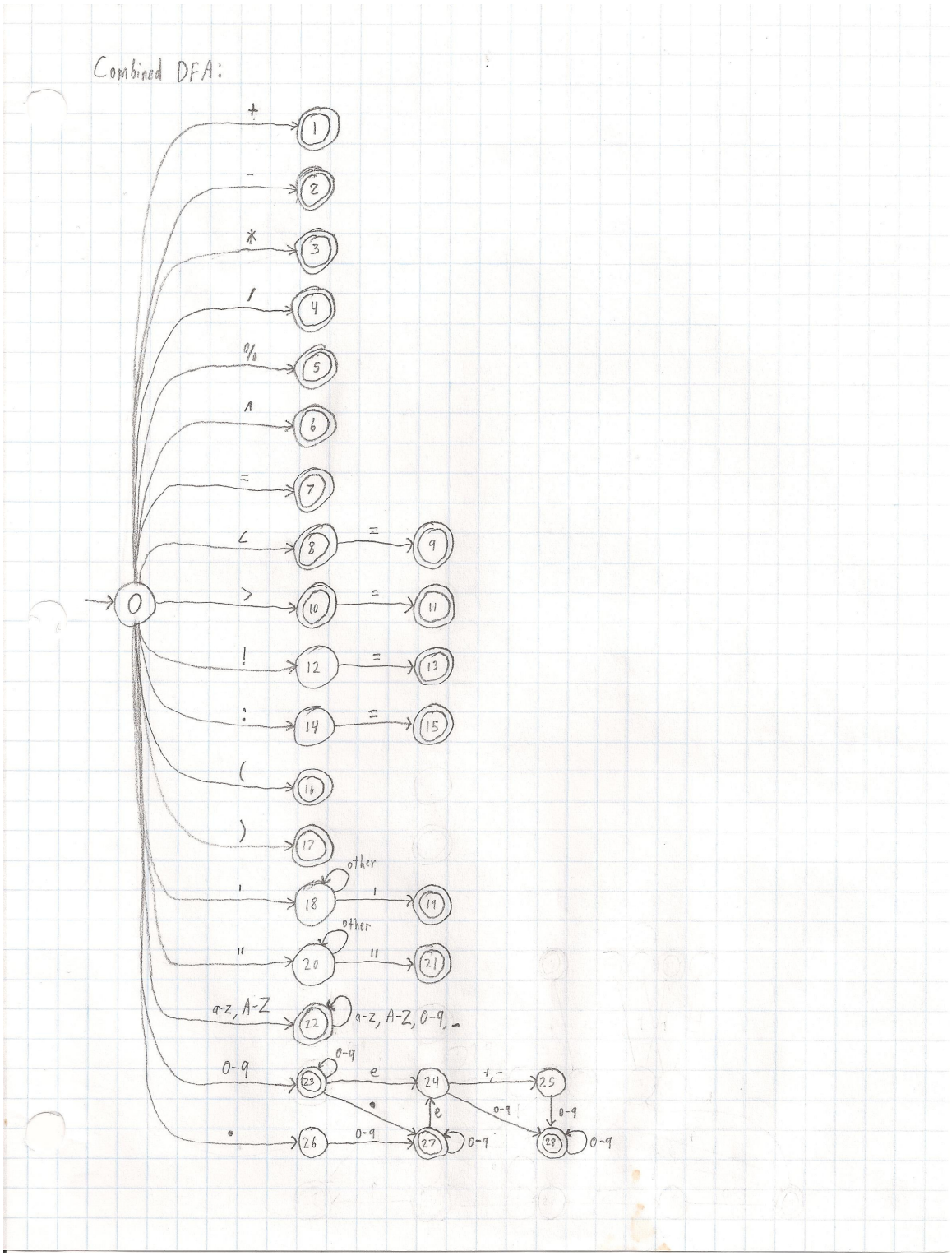
NFA:



DFA:



Name: Sean Rettig
Date: 2015-01-30



Name: Sean Rettig
Date: 2015-01-30

Specification (what do <u>you</u> think the purpose of this milestone is)

The purpose of this milestone was to learn the basic concepts associated with the lexing process and learn how to implement basic lexing functionality using finite state automata. We then converted these finite state automata into working programs that could not only accept or deny input, but tokenize it as well so that we can use the tokens as input to our parser in later milestones.

Processing (how did <u>you</u> go about solving the problem)

I went about solving the problem by first reading through the Naive Semantics page and attempting to write a few basic programs in the Itty Bitty Teaching Language (IBTL) (these programs later became the basis of my tests once I felt I understood the IBTL well enough). Then I identified all the different types of lexemes I would need to recognize and started drawing NFAs/DFAs to accept them. Once I had DFAs for all of the types of lexemes, I combined the DFAs into one big DFA and then started looking into actually implementing my lexer. I originally started by looping over each character, switching on each state, and then traversing an if-else block to determine the next state (as was shown in an example in the textbook), but I quickly found this cumbersome and instead implemented State objects that stored their number, what kind of lexeme they returned (if they were an accept/final state), and a map of char sets to other states that described all possible transitions from that state. Then rather than having a huge switch and if-else block, I simply performed the same operations on each state using the transition information and lexeme types stored inside them. This simplified my code greatly and heavily reduced code duplication. I also found it easier to debug. As I debugged, I kept

Name: Sean Rettig
Date: 2015-01-30

improving my tests to cover more and more cases until I felt that the lexer was lexing the IBTL correctly.

Testing Requirement (how did <u>you</u> test for correctness)

I have 12 test files, 6 of which are intended to pass, and 6 of which are intended to fail. The passing ones all hold different language constructs of the IBTL (such as various forms of reals, expressions, operators, functions, keywords, identifiers, strings, etc.). Between the 6 passing tests, all operators and keywords are used at least once; they are often used multiple times and in various situations, such as with other types of lexemes adjacent to them and differing amounts/types of whitespace. Various real formats are tested, as well as several different types of strings (such as strings containing non-language characters, keywords, spaces, and even newlines).

The six failing tests all cause the lexer to correctly return an error due to situations such as non-language characters being used outside strings, incomplete operators that require multiple characters (like := and !=), and real formatting errors (such as lone decimal points or missing numbers after an “e”).

Retrospective (what did <u>you</u> learn in this milestone)

While I was already quite familiar with the use of NFAs and DFAs from CS 321, learning how to use them to implement lexers and the idea of pushing back extra characters were new concepts to me. Additionally, I feel like a got a little bit better at programming and software design, having come up with what I feel is a much superior DFA implementation to the huge switch and if-else block.