Summary of Lexical Analysis: The most important materials

1. How to compose regular expressions

For practical purposes, we emphasize the ability to write expressions to define tokens that are commonly used in programming languages, e.g. identifiers, numbers, and operators. It is also important to know how to write regular expressions for comments and white spaces.

In class, we discussed the following examples.

Identifiers: **l (l | d) *, where l = [a–z, A-Z] and d = [0-9]**

<int>: **d d***

<real>: **<int> (ε | "." <int>)**

[NOTE: an equivalent form for <real> is **<int> | <int> "." <int>**

<number>: **<real> (ε | (E | e) (+ | - | ε) <int>)**

We see that the symbol ε is very useful for indicating substrings that are optional.

2. The equivalence between regular expressions and finite automata

If we know a set of strings can be represented by a regular expression, then the set must also be representable by a finite automaton, and vice versa.

Students must be able to apply algorithms in the textbook to convert any given regular expression to an equivalent NFA. We also must be able to convert any given NFA to an equivalent DFA.

(We have mentioned that there exist algorithms to convert a given DFA to an equivalent regular expression, but we do not study or require such algorithms. There exist algorithms to reduce a DFA to an equivalent DFA that has the minimum number of states. This reduces the storage requirement, but the time to analyze the input is the same, i.e. linear time with respect to the input length. We do not study or require the minimization algorithm.)

2. Scanning the input based breath-first search of the NFA

Following the algorithms in the textbook, at each step, we first determine the set of all states that can be reached before scanning any character in the input. This set is the ε-closure of the original starting state. We then iteratively scan the next character in the input. In each iteration, we first determine the

set of all NFA states that can be reached by scanning the next character and then we compute the ε-closure of this set. We do this until exhausting the input or until we cannot reach any state given the next character. In the latter case we reject the input. In the former case, we check to see whether the current set of states include any accepting state. If not, we reject the input. Otherwise, we accept the input.

In class, we worked on the example of scanning the input "8.0" based on an NFA constructed for <real> = <int> | <int> "." <int>

3. Converting an NFA to an equivalent DFA

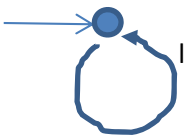
When we analyze the input based on an NFA, every time we scan the next input, we spend time computing the whole set of states that can be reached. To avoid this cost, we can pre-compute all possible sets of NFA states that might be reached given any arbitrary input. To do this we need to pre-compute the related ε-closures also. As the result, we obtain a DFA and reduce the scanning time to linear in terms of the input length. This, however, comes with the potential (storage) cost of introducing additional states. The total number of additional states in the worst case can be exponential.

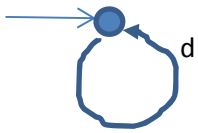
Scanning the input based on the original NFA avoids such storage cost because at each step, we overwrite the states that are reached in previous steps.

4. Applying Thompson's construction steps to convert any given regular expression to an equivalent NFA

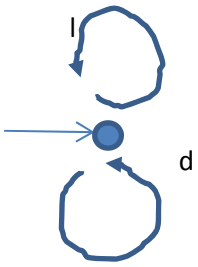
In class, we worked on the examples of <int> and <real>. Notice how ε-edges are used to construct the graph at each step. The use of such edges, in addition to the requirement that, at every construction step, we main a single (intermediate) accepting state and a unique incoming edge to the (intermediate) starting state, is to ensure a consistent construction procedure without causing unintended strings to be recognized.

Example: consider regular expression **l* d*,** where an NFA can be drawn for **l*** and d* can be drawn respective as the following:

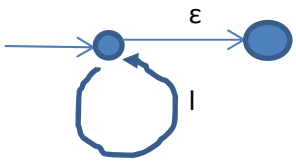


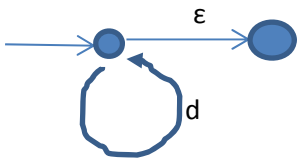


A simplistic way to build the graph as the result of concatenation of R1 R2 may give us the following incorrect graph:

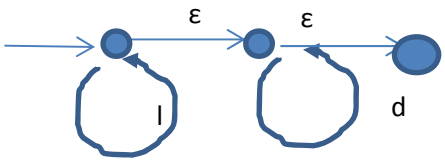


Adding ε-edges would avoid this incorrect result:





After concatenation, we have the following correct graph



Limitations of regular expressions

Although we do not require the details of the proofs, it is important to understand what kind of sets of strings cannot be specified by using regular expressions alone.

In class, we explained that a list of simple assignment statements such as "id = id + id * id - … + id;" can be easily specified by using regular expressions, as shown following:

(id = id ((+ | - | *) id)* ;)+

Unfortunately, as soon as we introduce parentheses, regular expressions are no longer powerful enough to specify all legal input. In class, we present an argument, by contradiction, to show that it is impossible to match the left parentheses with the correct number of right parentheses by using any DFA (and equivalently by using any regular expression).

Another important limitation of regular expression is that it does not contain any information about the structure of the program constructs. For example, it does not contain information about the precedence among operations (+, *, etc) and therefore cannot be used to build AST.