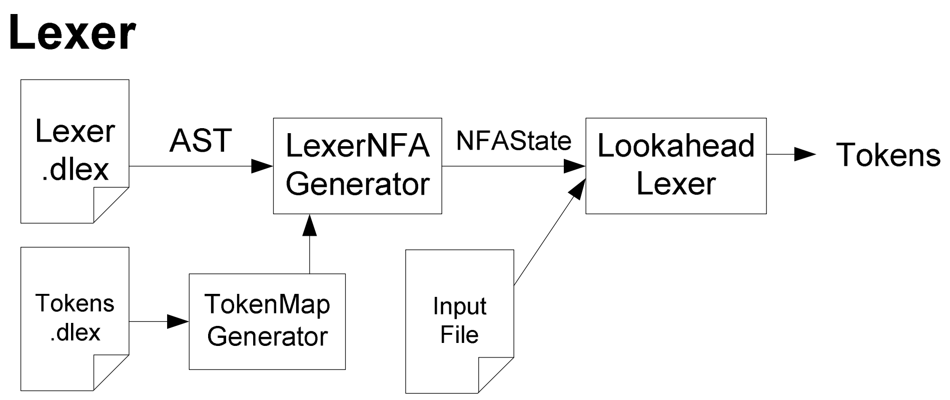
The Lexer

For PA2, you will start by writing a scanner that breaks a file into tokens based on a set of regular expressions provided in an input file. The main structure is illustrated by the picture.



Your lexer will have three inputs, a lexer description file containing the regular expressions you want to scan from your input. The second file will contain a listing of all the tokens that you will be looking for in the input, and the third file is the actual input file, to be scanned in search of tokens, and the tokens will be the output of the lexer.

The most important components will be the LexerNFA Generator, and the Lookahead Lexer.

Regular Expression to AST to NFA

We learned how to convert regular expressions to ASTs yesterday in lecture. We will go through an example here. Assume we define the following AST nodes for regular expressions:

| → OR

• → CONCAT

* → STAR

+ → PLUS

Assume this language only has characters. Note that the or and concat operators may have extended operands. Convert the regular expression (a|b|cd)* to an AST.

Given the AST, the NFA can be generated by simply traversing the tree. You can do this because there is a simple way to define an NFA for an expression involving any of the above operations in terms of the NFAs for its operands.

In PA2, we would like to use the Java parser from Eclipse to build the AST for us. So we introduce the following mapping:

| → |

• → +

* → *0

+ → *1

This makes the syntax for the regular expression be legal java syntax (but with radically different semantics!!). Rewrite the regular expression (a|b|cd)* using the new operators. How does this change the AST? |, +, and * are infix expressions.

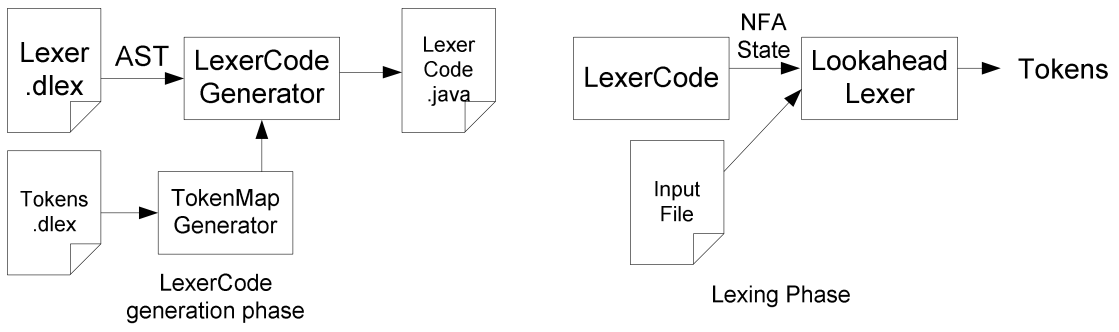
Lexing the File

With the NFA, we can give it to the lexer engine, and have it simulate it and lex the file. The simulation is a bit tricky for two reasons. First, you know that at each state you can be in a set of states, not just one state. Second, you want to return the maximum munch lexeme. The way to do this, is that when you reach a final state, you don't do the action. Instead you remember the current input position and the action and keep running. Once you get stuck in the NFA, only then do you run the last action that matched, which would be the maximal munch lexeme. You also need to back up in the input to the last remembered position. This requires some kind of buffering. In Java, you can use a PushbackReader, as in the PA2 starter kit.

Compiling a lexer

In practice it's much more efficient when you have the NFA hardcoded in your lexer, instead of having to build it every time from a specification file. Thus, for the second part of the project, instead of lexing the file with your NFA, you will build a Java program that will have the NFA hardcoded into it, and will simply get the source text as an input and scan it.

LexerCodeGenerator



In a way, original Lexer is an interpreter for lexer specifications, and LexerCodeGenerator is a compiler.