Experimental Report of lexical analyzer

1. Motivation/Aim

The purpose of the experiment is to establish a lexical analyzer according to some simple programming language laws, like java, python, c++ or other programming language. During the lab, we can learn more about how to setup a NFA or a DFA according to some given regular equations.

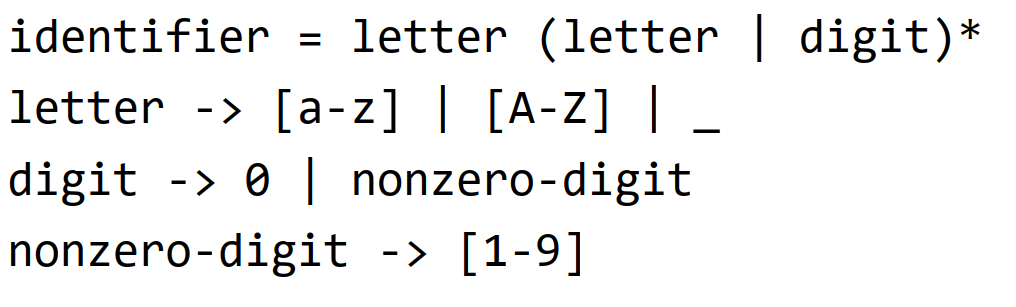
1. Content description

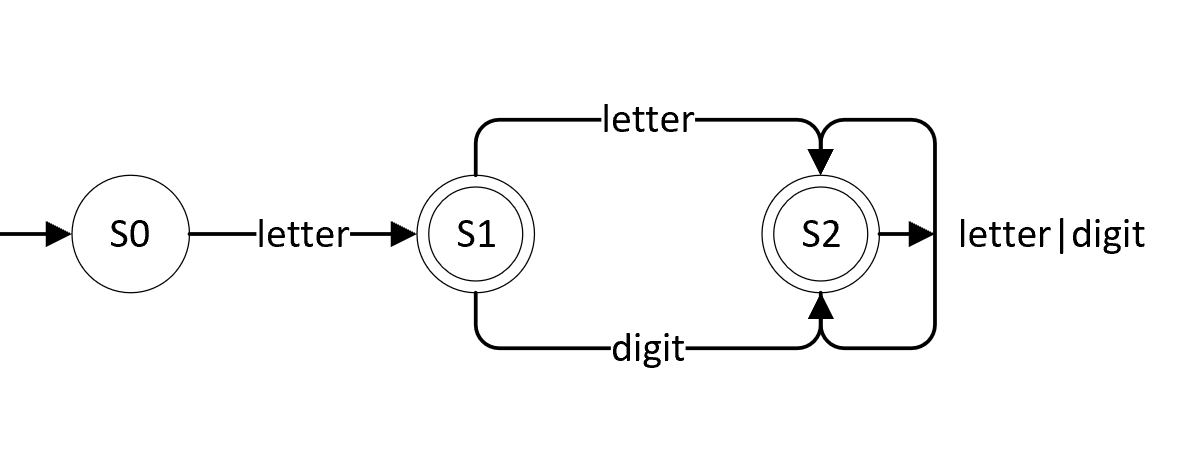
In this experiment, we realize the conversion of the most basic and simple c++ program. The lexemes we can recognize include identity, reserved word, number, operator and so on. After running the program, the source code is converted to a lex file., which will be used in the syntax analysis.

1. Ideas/Methods

Before the experiment, we have to build the NFA about recognizing token and transform it into a DFA. To different token, we need different NFA.

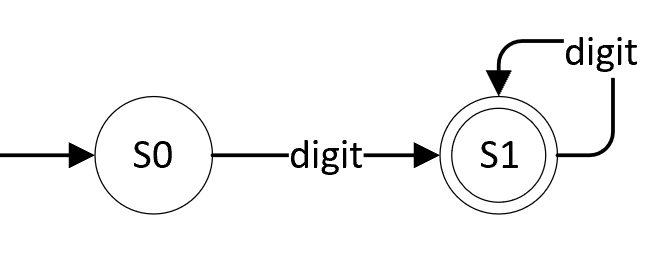
To identifier, we establish the RE as following. We also get the DFA according to this RE.





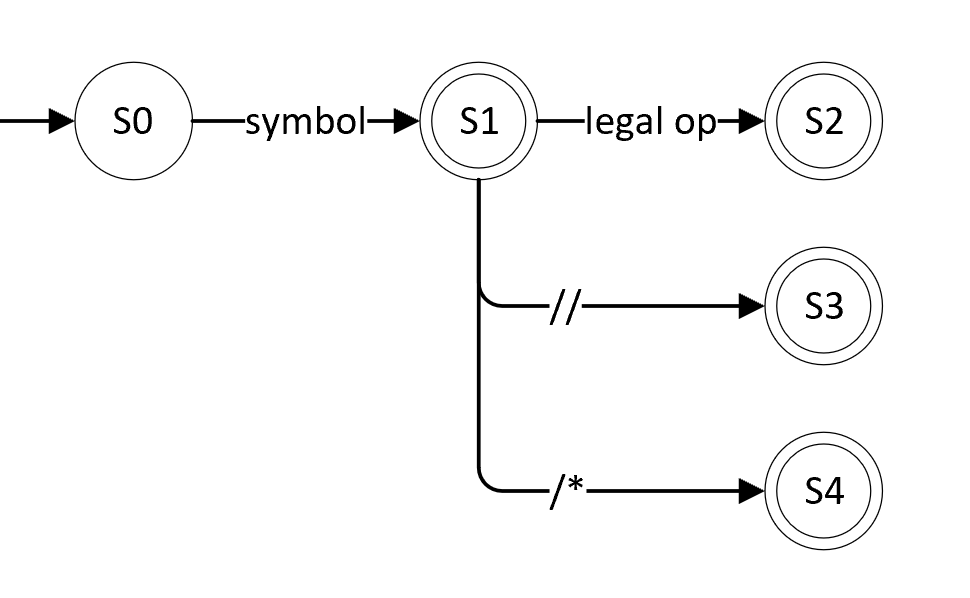
With this FA we can achieve id reading and recognition.

For numbers, we specify that it is just a collection of numbers, so it is easy to recognize. The corresponding FA has only one state.



For operators, since we take into account the presence of binomial operators and annotations, we use checking when the second operator is encountered. When the operator we have in existence is legal, then it goes on the stack to form a new binomial operator, otherwise it is used only as a unipersonal operator. Legal op contains <=, >=, ==, != and so on.

There are two annotation situations considered separately here. The first is a single line comment, where the next line can be read directly. The second is a multi-line comment. After reading here, we have to search for the end of the comment symbol in the future code, and then continue reading after the comment.



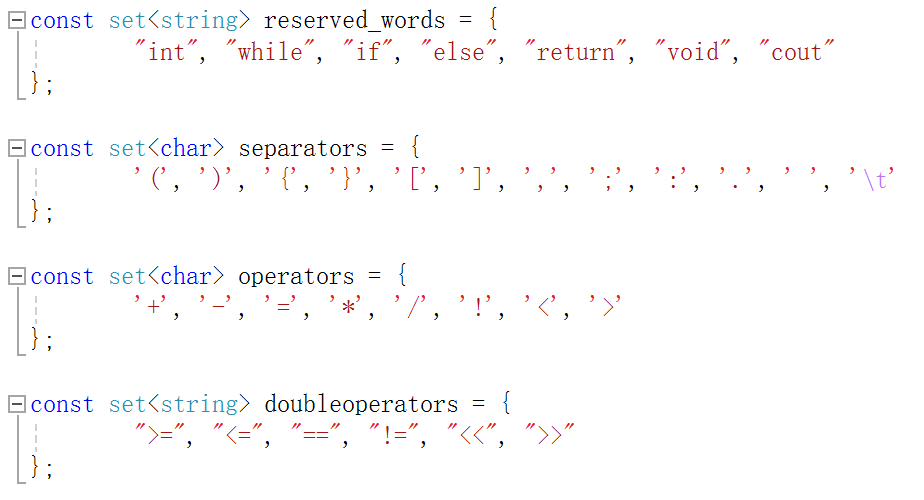
To reserved words, since keywords starting with # were not considered, we used a simplified approach. Read all the keywords as ids. After reading the ids, determine if they are keywords. If it is, then transform to keyword state.



The next section is about the generation of error. For the lexical parser, it can't recognize syntactic issues such as whether there are two operators concatenated together. So the error here would normally only occur with a misnamed id. When our input does not match the naming convention of the id, an error is reported.

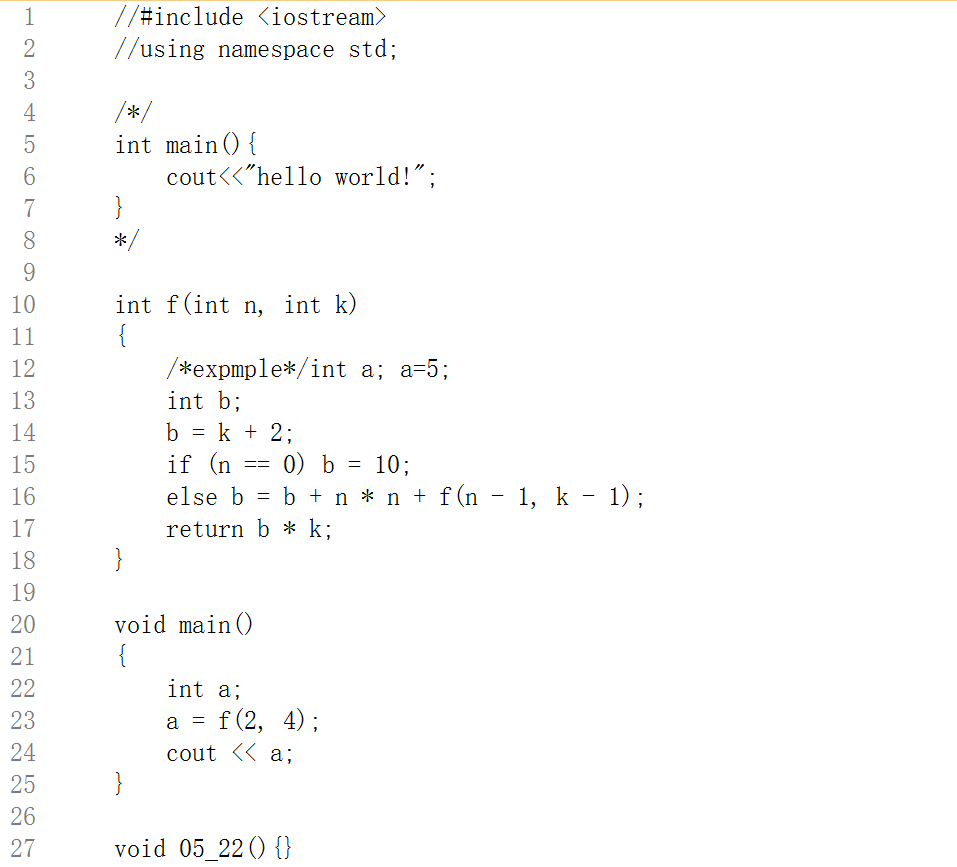
1. Description of important Data Structures

The different string types that need to be recognized are stored inside the set's collection. It is easy to read the characters and determine the token.



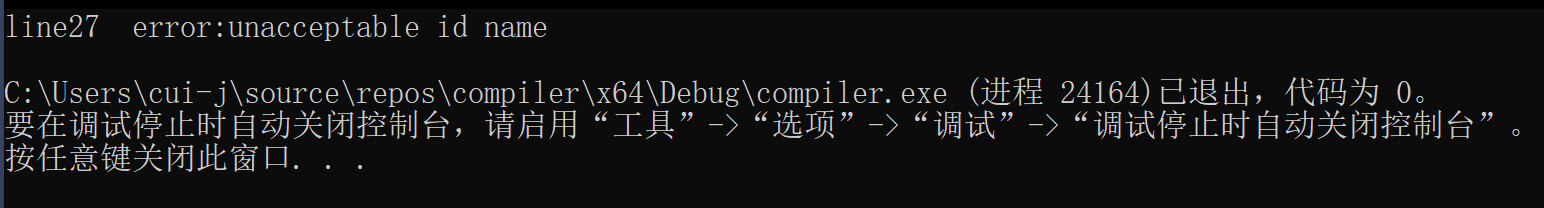
1. Use cases on running

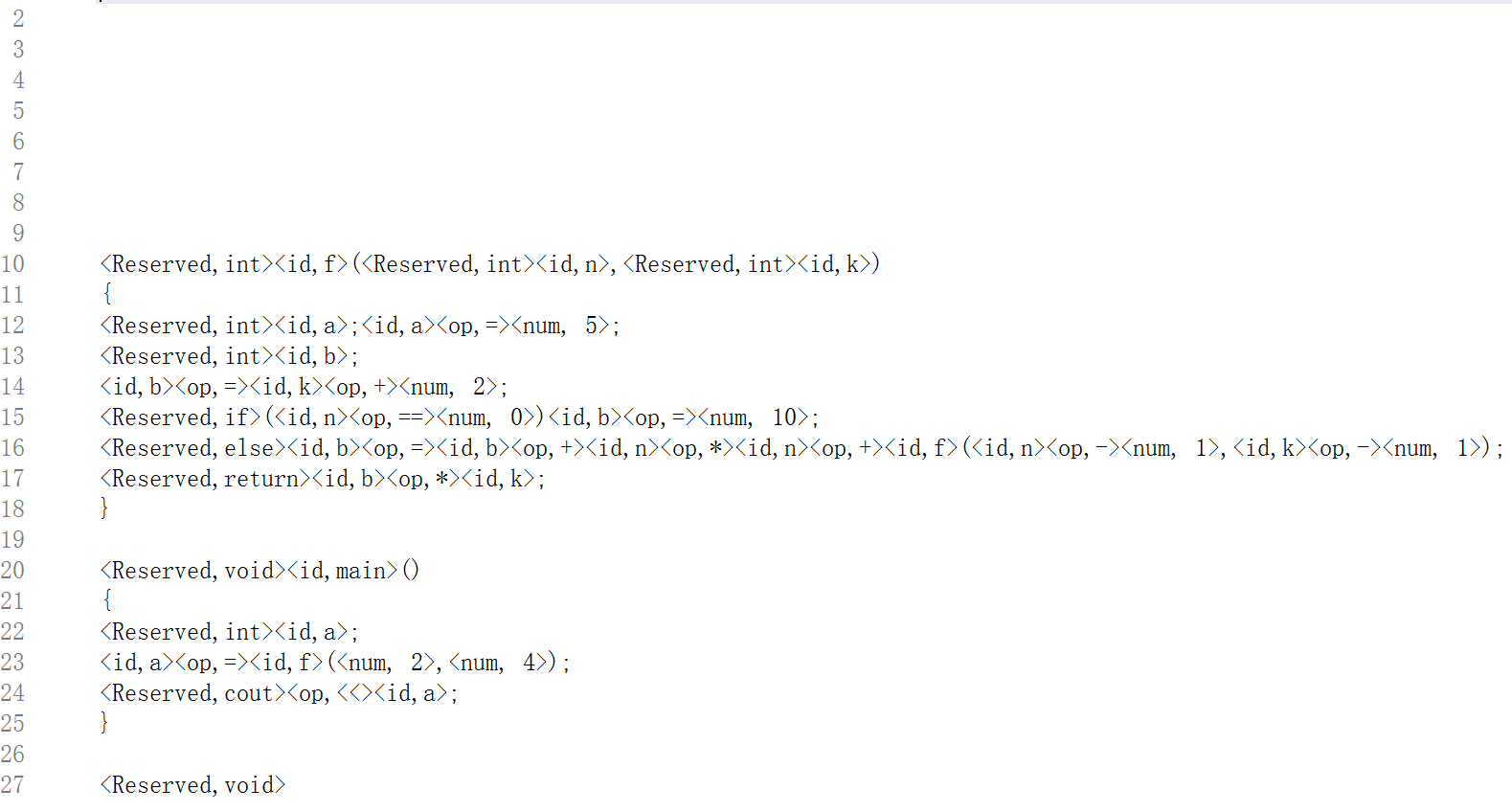
To verify the correctness of the code, a sample code was used for testing. The sample code is shown below.



This code contains two annotation structures and basic token types. A typical naming error is also added at the end of the code. The results of the run are shown below. The type of error contained in the code of the error and the number of corresponding lines.

At the same time we can get the relative lex file, which saves each token with the corresponding one.





This output does not remove the blank lines. It is to demonstrate the effect of comment removal. For single-line comments, all the lines after // are deleted directly. For multi-line comments, only the content between the two comment symbols will be deleted.

1. Problems occurred and related solutions

The code for lexical analyzer is imperfect. There are still some problems.

The first is that the program is not intelligent enough. When the program generates an error, the program terminates. There is no way to recognize multiple errors at the same time. This is because when the program generates an error, it will start reading from the character that generated more error, and will not be able to effectively recognize following code.

Secondly, there is no way to implement a hint for wrong ids. This is because when an error is generated, there is no way to confirm when the id will terminate.

The third problem is the lack of error messages for multi-line comments without \*/ symbols. This is because in the experiments, the treatment of multiline comments was simply deletion. When no \*/ is encountered, the code is constantly deleted and thus no error message is generated.

In order to address these issues, the error handling mechanism needs to be refined by adding a new state for error management. Errors are further refined and handled within this state.

1. Your feelings and comments

The process of writing the lexical analyzer gave me a deeper appreciation for regular expressions and DFA. I have been able to flexibly use the tools to analyze some simple tokens.

However, there are still imperfections in the handling of error messages. Processing error messages is a lot more complicated than I thought it would be. It's hard to recognize tokens very accurately and give appropriate hints with error messages. This aspect still needs to be improved.