Writing A Scanner

A Scanner should create a token stream from the source code. Here are 4 ways to do this:

A. Hack something together till it works. (Blech!)
B. Use the language specs to produce a Deterministic Finite Automaton (DFA) that identifies tokens, then write a program that models this DFA.
C. Write table-driven Scannner (more on this later).
D. Use LEX (a standard Unix program that generates a Scanner.

It is not hard to guess which approach we'll take. We first need to do a little background work.

A *regular expression* is an expression made from the following rules. We start with an alphabet Σ.

a) If letter a is in Σ, then a is a regular expression.
b) If s and t are regular expressions, so are st, and s|t.
c) If s is a regular expression, so are (s), s+ and s*.

Regular expressions are patterns that describe strings:

a) Any letter of the alphabet describes itself.
b) st describes the strings that can be made from something described by s, followed by (concatenated with) something described by t. s|t describes anything that is described by either s or t.
c) (s) describes the same strings as s. s+ describes all the strings that can be made by concatenating together one or more strings described by s. s* is the same, only concatenating 0 or more strings described by s.

For example, let dig represent the decimal digits and pos the non-zero digits:

    pos ::= 1|2|...|9
    dig ::= 0|pos

Then we could represent the positive integers as

    PosInt ::= pos dig*

We could represent the nonnegative integers as

    NonNeg ::= 0 | pos dig*

We could represent all integers with

    Int ::= 0 | pos dig* | - pos dig*

If we let Letter be the alphabet of letters a-z, then

letter* bob letter*

is the set of strings that contain "bob" as a substring.

Here is a regular expression describing the strings of 0's and 1's that contain an even number of 1's:

$$0^*(10^*10^*)^*0^*$$

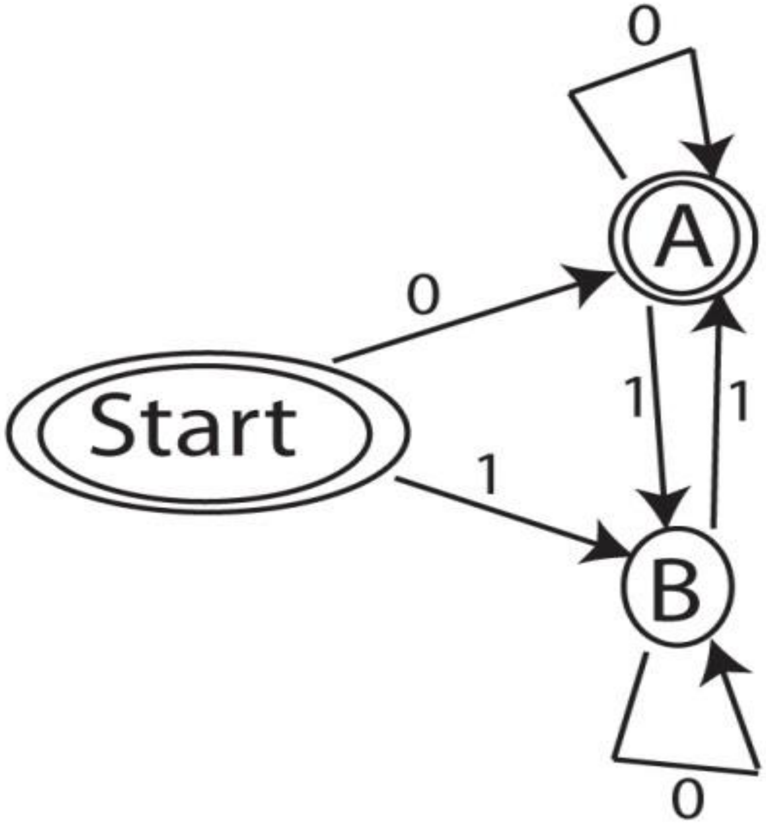
In language theory a language is any set of strings. We say a language is *regular* if it is described by some regular expression.

Here is why we care: in almost all programming languages, the set of tokens forms a regular language.

Sadly, programming languages almost never specify the regular expressions that describe their tokens.

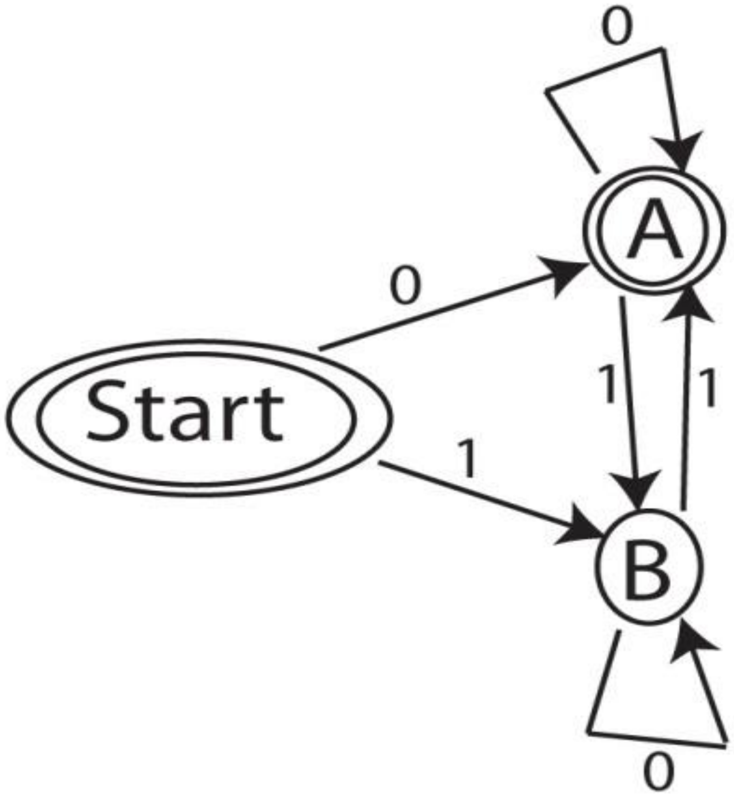
Fortunately, there is another way to describe regular languages, and it leads directly to programs.

A *deterministic finite automaton* or DFA consists of a set of states and transitions between those states. Some people call this a "finite state" automaton. Rather than give a formal definition, here is a picture of one:



The states are represented by circles, the transitions by labeled edges. One state is labeled "Start". One or more states are represented by double circles; these are the "terminal" states.

We say this automaton "accepts" or "rejects" a string according to the following rules. We begin in the Start state at the start of the string. We move through the transitions of the automaton using the letters of the string to determine which transition to take. If we get to a state where there is no transition out corresponding to the next letter, we reject the string. At the end of the string, we look to see what state we are in. If it is a terminal (double circle) state we accept the string; it it is not a terminal state we reject the string.



For example, consider our automaton with the string 10010. We begin in the Start state. With the first 1 we go to state B. On the next two 0's we stay in state B. On the 1 we go to state A; on the final 0 we stay in state A. We are at the end of the string in a terminal state, so this automaton accepts string 10010.

It is easy to turn such an automaton into code. Here is a Java function that takes a string and returns true or false according to whether this automaton accepts or rejects the string. For simplicity I have numbered the states: Start is state 0, A is state 1, and B is state 2:

```
public static boolean Accept(String s){
            int state = 0;
            for (int i =0; i < s.length(); i++ ) {
                        char ch = s.charAt(i);
                        if (state == 0) {
                                    if (ch == '0')
                                                state = 1;
                                    else
                                                state = 2;
                        }
                        else if (state == 1) {
                                    if (ch == '1')
                                                state = 2;
                        }
                        else if (state == 2) {
                                    if (ch == '1')
                                                state = 1;
                        }
            }
            if (state == 2)
                        return false;
            else
                        return true;
}
```

In the Theory course (CSCI 383) we show that a language is regular if and only if it is the language accepted by some DFA.

The tokens of most programming languages form regular languages. It is usually an easy matter to draw out a DFA that accepts these tokens and then convert that DFA to code. In principle, that is your Scanner.

For example, suppose our language has 4 kinds of tokens: Identifiers, non-negative Number, + and *. Here are some definitions:

alpha ::= a|b|...|z

digit ::= 0|1|2|...|9

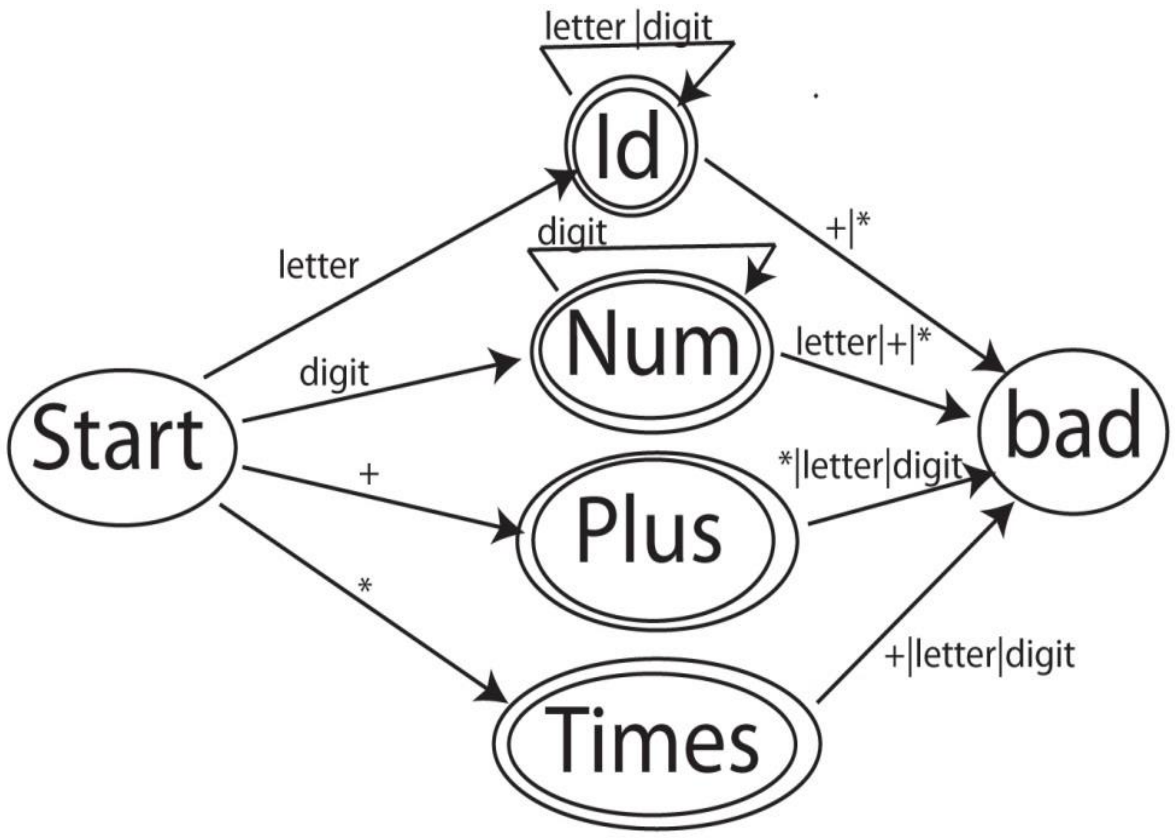
Id ::= alpha (alpha | digit)

Int ::= digit$^+$ (let's not worry about leading 0's)

Plus ::= +

Times ::= *

Here is the DFA for this:



It is easy enough to turn this DFA into code.

There is an issue here that is inherent to thinking of scanners as DFA's. We scan through the entire program; no one is giving us individual strings and asking if these strings are tokens. One of the tasks of our scanner is to find the boundaries where one token ends and another one starts.

Think of the expression 23*4. If we are scanning through the start of this we see the digit 2, and know we are looking at a number. We see the digit 3 and we are still looking at a number. We see the *character and we know we are done with the number and have found a token, representing 23. However, we have already read the start of the next token!

We will generally consider a token to be the longest string that will result in a valid token. So "23skidoo" results in two tokens: the Num "23" and the Id "skidoo".

We still need to deal with the fact that we don't recognize that most tokens are finished until we have read the first character of the next token.

The solution to this problem is to keep a *lookahead* character -- the next character that has not yet been used.

This can be handled in various ways. In C, where there is an ungetc( ) function that sticks the most recently read character back on the input buffer, it is easy to use the input buffer for the lookahead; when you realize you have read past the end of the current token just unget the extra character.

Java, and many other languages, don't have an unget operation. I find that the easiest way to handle this issue in such languages is to read the source code one line at a time and keep an index to the current location on this line. When you use the current character from the line, increment this index. TO "unget" a character just don't increment the index. The end of the line terminates any token. When you are at the end of the line and need the next character, read the next line from the file and set the character index to 0.

We will structure our Scanner to have a getNextToken() method. This method reads through the source file until it has found the next token, which it assigns to a Token variable.

The next two slides have such a function for the Token language consisting of Nums, Ids, and + (with room for 4 more lines we could add a case for * as well).

```
public void getNextToken() throws ScannerException {
        int i = 0;   // i points at the next character to handle
        int j=0;
        while (i < currentLine.length() &IsWhitespace(currentLine.charAt(i)))
                i += 1;
        if (i == currentLine.length()) {
                if (input.hasNextLine()) {
                        currentLine = input.nextLine();
                        lineNumber += 1;
                        getNextToken();
                }
                else {
                        nextToken = new Token(Token.T_EOF, "", lineNumber);
                        currentLine = "";
                }
        }
```

```
else  { //  i < currentLine.length()
            char ch = currentLine.charAt(i);
            if (isDigit(ch)) {
                        j = i+1;
                        while (j < currentLine.length() && isDigit(currentLine.charAt(j)) )
                                    j += 1;
                        String tokenString = currentLine.substring(i, j);
                        nextToken = new NumberToken(Token.T_NUM, tokenString, lineNumber);
                        currentLine = currentLine.substring(j);
            }
            else if (isLetter(ch)) {
                        j = i+1;
                        while (j < currentLine.length() && isAlphaNum(currentLine.charAt(j)))
                                    j += 1;
                        String tokenString = currentLine.substring(i, j);
                        nextToken = new Token(Token.T_ID, tokenString, lineNumber);
                        currentLine = currentLine.substring(j);
            }
            else if (ch == '+') {
                        nextToken = new Token( Token.T_PLUS, "+". lineNumber);
                        currentLine  = currentLine.substring(i+1);
            }
}
```

Most languages, including BPL, have some tokens that are keywords, such as "if", "while", "void", etc. You could make a complicated DFA that recognizes these, but an easier way to handle them is to make them part of the code that finds Identifiers. Before you decide that a string represents an Id token,   run it through a filter that searches for each for each of the keywords. If you find that your token string is "if", for example, you want to make it a IF-token rather than and ID-token.