CS 164 Section Notes on Lexing (2/1/2010)

Bill McCloskey

Consider the following input to `flex`.

```
%option noyywrap

%{
enum { EOF_TOK, IN, INTO, ID };
char *names[] = { "EOF", "IN", "INTO", "ID" };
%}

%%

in { return IN; }
into { return INTO; }
[a-z]+ { return ID; }
[ ] {}

%%

int main(int argc, char *argv[])
{
  while (1) {
    int t = yylex();
    if (t == EOF_TOK) break;
    printf("* Got token %s (%s)\n", names[t], yytext);
  }
  return 0;
}
```

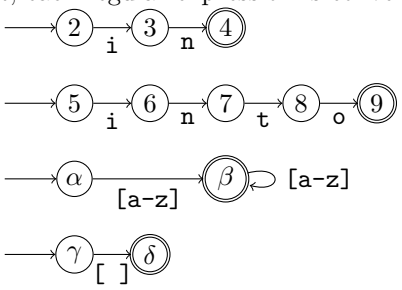
The four lines after the first `%%` line are the *rules*. Each one is a regular expression followed by an action. When the given regular expression is matched, the code from the action is run. The first three actions return an enumeration identifying the token type. Whitespace is ignored.

If you have the `flex` tool installed, you can save this code to a file (say `tokenize.l`) and then run `flex tokenize.l` on the command line. This will generate `lex.yy.c`. Then compile this via `gcc lex.yy.c` and run `a.out`. Here is a sample run.

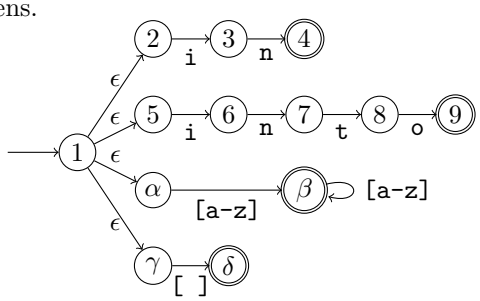
```
hello a in into x
* Got token ID (hello)
* Got token ID (a)
* Got token IN (in)
* Got token INTO (into)
* Got token ID (x)
```

1 What `flex` does (approximately)

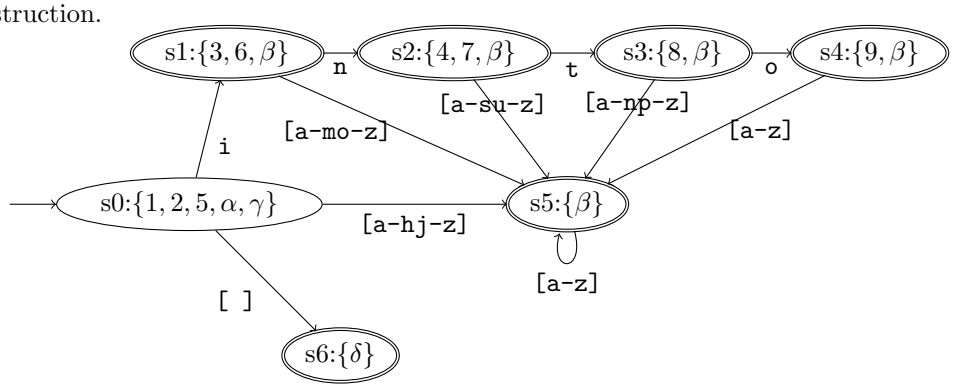
First, each regular expression is converted to an NFA.



Then the NFAs are combined into a single NFA that accepts any of the tokens.



Next, this NFA is converted to a DFA via the subset construction. For more details on this algorithm, look up the Wikipedia article on the subset construction.



This diagram omits the state ∅, which we will call s7. It is a dead state. Every other state except the start state goes to it on the space character. Also, s6 goes to the dead state on every input character.

From this DFA `flex` constructs a table. To find where to go from state s on input c, find the row containing s under the "State" column and scan across to the column labeled c.

State	`[a-hj-mp-su-z]`	`i`	`n`	`t`	`o`	`[ ]`
s0	s5	s1	s5	s5	s5	s6
s1	s5	s5	s2	s5	s5	s7
s2	s5	s5	s5	s3	s5	s7
s3	s5	s5	s5	s5	s4	s7
s4	s5	s5	s5	s5	s5	s7
s5	s5	s5	s5	s5	s5	s7
s6	s7	s7	s7	s7	s7	s7

Notice how we used the same column for every character in the class `[a-hj-mp-su-z]`. To save memory, `flex` also does this. It uses an array to map each input character to an *equivalence class*. It stores the transition function for the DFA in terms of equivalence classes rather than characters.

Let's look at some sample inputs. First consider the input "`inside␣`" (notice the space at the end). `flex` will run it through the DFA as follows.

$$\text{s0} \xrightarrow{\texttt{i}} \underline{\text{s1}} \xrightarrow{\texttt{n}} \underline{\text{s2}} \xrightarrow{\texttt{s}} \underline{\text{s5}} \xrightarrow{\texttt{i}} \underline{\text{s5}} \xrightarrow{\texttt{d}} \underline{\text{s5}} \xrightarrow{\texttt{e}} \underline{\text{s5}} \xrightarrow{\texttt{␣}} \text{s7}$$

The final states are underlined. `flex` will stop processing when it reaches a dead state (i.e., a state that cannot possibly reach a final state). At this point, it looks back to the last time it was in a final state. Here, it is upon reaching s5 after reading `e`. Thus, "`inside`" forms the token.

To choose the action to perform, `flex` looks at the original NFA states that generated s5—in this case, $\{\beta\}$. The state β was from the third rule in `tokenize.l`, the `ID` rule. `flex` executes the action associated with this rule.

Now consider the input "`in␣`". Running the DFA gives the following.

$$\text{s0} \xrightarrow{\texttt{i}} \underline{\text{s1}} \xrightarrow{\texttt{n}} \underline{\text{s2}} \xrightarrow{\texttt{␣}} \text{s7}$$

This time s2 was the last final state reached. It corresponds to the NFA states $\{4, 7, \beta\}$. Of these, only 4 and β are final states. They correspond to different rules in the `tokenize.l` file. To disambiguate, `flex` chooses the first rule, `IN`, to execute because the programmer listed it first.

Conclusion

- When multiple rules can match the input, `flex` chooses the one that produces the longest match. It does so by running the DFA until it gets to a dead state. Then it finds the last input position in which it was in a final state and selects a rule based on state.
- When there are two rules that both produce a match of the same length, `flex` chooses the one that was listed first in the `.l` file.