Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 — Handout 9 — JLex — Thursday, September 9

JLex is a scanner-generator for java. It's similar to the traditional `lex` command. The input of JLex is a scanner description file, and the output is java code. To use JLex, you write the description file, run JLex to produce a java file, and then compile the java file using javac to get a scanner program. You'll also have to write some support java files to read the commandline, load in the input program, and so on.

JLex is also documented on the web. The link is at the bottom of the 6.035 web page, at `http://web.mit.edu/6.035/`. JLex is a tool developed for use with Appel (the Tiger book), so you may find this a useful reference for this part of the course.

Here's a simple example scanner.lex file:

```
%%

%function nextToken
%type java_cup.runtime.Symbol
%char

%{
private java_cup.runtime.Symbol tok(int kind, Object value) {
    return new java_cup.runtime.Symbol(kind, yychar, yychar+yylength(), value);
}
%}


alpha=[A-Za-z_]
digit=[0-9]
char=(\\\"|\\\'|\\\\|\\n|[^\"\'\n])


%%
[" "\t]                     {}
{alpha}({alpha}|{digit})*   { return tok(sym.ID, yytext()); }
null                        { return tok(symbol.NULL, yytext()); }
.                           {any code to handle non-matching symbol}
```

There are three sections, separated by `%%`. The first contains code that you want JLex to add to the top of its generated file. In this example, that section is empty.

The second section is for JLex directives. There's a lot of different things that can go here, and you should consult the JLex manual for complete information. In this example, we've used the

`%function` and `%type` directives to specify what we want JLex to call the tokenizing function, and what we want it to call the return type. The `%char` command enables character counting using the `yychar` variable. The `%{` and `%}` enclose code that we want copied into the class definition for the lexical analyzer. Here, we're defining the method `tok` which we use to build a symbol.

The other thing we can put in the second section are macros, which simplify writing the descriptions of the tokens. Here we define `alpha`, `digit`, and `char` as regular expressions. Note that JLex supports a richer regexp syntax than what was described in lecture. The notation `[A-H]`, for example, is equivalent to `(A|B|C|D|E|F|G|H)`. The definition of `char` is weird because we want to support things like `\n` for the literal newline. In the description file, `\n` will match a *literal* newline character in the input stream, which isn't what we want. So, we use `\\n`, or `\\\"`.

The third section of the lex file is where the actual tokens are defined. For each token, we specify a regular expression that matches it, and an *action* to perform when that token is found. The first "token" we list is a regexp to match whitespace, and it has an empty action because we just want to skip them. We define variables with a regexp that uses the macros defined above, and the action returns a new `Symbol`. Inside JLex, the `yytext()` function returns the actual scanned text, and presumably the `java_cup.runtime.Symbol` constructor saves that value away for later use.

The last regexp is "`.`", which matches anything. It is akin to a default case in a switch statement. Since the entire program should be tokenizable, anything that doesn't match any token matches this one, and is an error. The action should do something to produce an error message, including the offending line number and text.

To use the lexer, you'll need to write some wrapper code. Your code will need to do something like:

```
lex lexer = new lex(input);
```

where `input` is some `FileInputStream` that you've created. Then you can call `lexer.Yytoken()` (or `nextToken()` if you've renamed it as in the above example) and it'll return the next token. The return type is `Yytoken`, or `java_cup.runtime.Symbol` if using the `%type` directive as above.