Jim Lambers
CS143 Compilers
Summer Quarter 2000-01
Introduction to Flex

1 Using Flex

`flex` is a *lexical analyzer generator*. To use it, you create a file that defines the following for each token that you want to recognize:

- A *pattern* that describes all valid lexemes for the token, and
- A corresponding *action* that you wish to be performed upon recognizing the token.

The file you create is processed by `flex`, and a file named `lex.yy.c` is generated. This file contains C code that will recognize your tokens from text read from standard input, and perform the actions you have prescribed for each token.

The output file `lex.yy.c` contains a function `yylex`, whose job is to recognize the next token in the input. You may call `yylex` from your own code simply by compiling `lex.yy.c` with the `-c` option to produce an object file `lex.yy.o`, and then linking that object file with your own code. You must also link with the `lex` library, using the `-ll` option, to include other functions called by `yylex`.

2 The `flex` input file

Your input file to flex is organized as follows:

```
%{
Declarations
%}
Regular definitions
%%
Patterns and actions
%%
Additional code
```

The declarations section should include any declarations you need for code that you will include in this file. If you are not using `flex` in conjunction with a parser generator such as `yacc`, then you will need to supply a declaration for the global attribute variable `yylval` (discussed further below), which may be of any type that you wish.

Next, you may describe any regular definitions that you will need to help you specify patterns in the next section. You may use any of the notational shorthands discussed in class for writing these regular definitions. It is strongly recommended that you use parentheses throughout your regular expressions to eliminate any possible ambiguity. You should make sure to enclose all of your regular expressions in parentheses since they will be included in other regular expressions.

After the first `%%`, you must list the patterns that describe your tokens, and the actions that must be performed when each token is recognized. Patterns and actions are discussed in greater detail in the next section.

Finally, after the second `%%`, you may add any additional code that you need. Often this section includes helper functions for the code that is included in the actions.

3 Specifying Patterns and Actions

Patterns can be specified in two ways: using the names of regular definitions defined in the previous section of the input file, or using strings representing actual lexemes for tokens. For tokens with very few lexemes, such as keywords, punctuation marks, or operators, the latter option is often simpler.

When adding patterns to your input file, keep in mind that the order may be significant in some cases. The reason for this is this one key rule that `flex` follows in order to resolve ambiguity: whenever a string fits multiple patterns, the first pattern listed in the file is chosen. Therefore, when recognizing tokens for a typical programming language that includes keywords and identifiers, the patterns for keywords *must* be listed before the pattern for identifiers, or all keywords will be treated as identifiers.

The code that you include in the actions depends on what information you need about each token. For a compiler that uses the lexical analyzer generated by `flex`, the following tasks should be performed:

- If the token has any attributes, they should be stored in the global variable `yylval`.

- An integer value representing the token that has just been recognized should be returned.

If any patterns are to be ignored, such as white space or comments, then no action should be provided.

After all of your patterns, it is common practice to add one final pattern that matches anything that does not match one of the previously defined patterns. Such a pattern can be specified using a single . (dot). The action to perform depends on the particular application, but it is most commonly used to report a lexical error, since this pattern is only matched when no patterns corresponding to valid tokens can be matched. This pattern is sometimes referred to as a *default pattern*.

A note about the syntax of the input file: while the code for actions may extend over multiple lines, the first line of code for an action must be on the same line as the pattern associated with that action.

4 Flex Global Variables

Flex uses the following global variables to help you obtain information about recognized tokens. These variables should only be used within the actions prescribed for each of your patterns, or by functions called within those actions. When executing these actions, it is your responsibility to store any data held in these variables, as their values will change once the next token is recognized.

- `yylval` is of type `YYSTYPE`. You must supply the definition of this type and declare `yylval` to be of this type. This is best done in an include file that is included by your Flex input file. If you are using `flex` in conjunction with a parser generator that is compatible with `yacc`, then `yylval` is declared for you, and `YYSTYPE` is defined in the input file to the parser generator.

  `yylval` is used to hold any application-dependent implementation about the token that has just been recognized, so that it may be used by other portions of the application.

- `yylloc` is of type `yyltype`. Like `yylval`, you must define `yyltype` and declare `yylloc` to be of type `yyltype`, unless you are using a `yacc`-compatible parser generator, which handles these details for you. This variable holds application-dependent information about the location of the token that has just been recognized.

- `yytext` is a pointer to a null-terminated string containing the text of the lexeme that has just been recognized as a token. When using `yytext`, it is strongly recommended that you use the standard C library function `strdup` function whenever it is necessary to store the lexeme elsewhere. `yytext` is declared for you in `lex.yy.c`.
- `yyleng` is an `int` whose value is the length of the lexeme stored in `yytext`. It is declared for you in `lex.yy.c`.

5 The Lookahead Operator

The *lookahead operator* `/` may be used in patterns. A pattern of the form p_1/p_2 matches if the input matches p_1p_2, and the lexeme is the portion of the input that matches p_1. In other words, `flex` matches the input against p_1, but then "looks ahead" and ensures that the subsequent input matches p_2 before declaring a match.

6 A Sample Flex File

A `flex` file that recognizes the tokens from the simple arithmetic expression language presented in the first lecture is available in the course directory, at

```
/usr/class/cs143/sample/compiler.l
```

To compile the lexical analyzer defined in this file, execute the following shell commands:

```
> flex compiler.l
> g++ -DYY_MAIN lex.yy.c -ll
```

The first command generates the implementation the lexical analyzer defined in `compiler.l`, in C code, and writes it into the file `lex.yy.c`. The second command compiles `lex.yy.c`, indicating that the default `main` function supplied in that file is used. Note the inclusion of `libl`, the `lex` library. This contains externals needed by `lex.yy.c`.

This program, when executed, will recognize one token. If it is a valid token, the program will exit quietly. Otherwise, it will report that there is a lexical error and then exit.

7 Flex Resources

The manual page on `flex` is very thorough, and should answer any questions you have on the nuances of `flex`. You can also learn more about `flex` on the GNU web site at `http://www.gnu.org/software/flex/flex.html`