An introduction to Flex

Computer Science 324
Theoretical Computer Science

3 February 2004

1 Introduction

1.1 What is Flex?

Flex is a program that tokenizes lexemes (a lexeme is a string in a source file that forms a token). It recognizes lexemes represented as regular expressions. [1]

1.2 A short history

Before we start out discussion of Flex, let us look a little bit at Lex, the predecessor of Flex. Lex was developed at Bell Laboratories in the 1970s. Lex was designed by Mike Lesk and Eric Schmidt to work with yacc (a program that we will discuss later in this course). BSD and GNU Project distribute a free version of lex, called Flex (Fast Lexical Analyzer Generator). Flex was written by Jef Poskanzer. It was considerably improved by Vern Paxson and Van Jacobson. [3]

Several other versions of Lex exist. There are at least two versions available for MS-DOS. Furthermore, implementations similar to Lex exist for the Java platform – JFlex and JLex are the most popular ones.

1.3 What does Flex do?

It takes as its input a text file containing regular expressions, together with the action to be taken when each expression is matched. It produces an output file that contains C source code defining a function `yylex` that is a table-driven implementation of a DFA corresponding to the regular expressions of the input file. The Flex output file is then compiled with a C compiler to get an executable. [4]

2 The format of a Flex source file

As shown below, a lexical specification file for Flex consists of three parts divided by a single line starting with %%:

```
Definitions
%%
Rules
%%
User Code
```

In all parts of the specification comments of the form **/* comment text */** are permitted.

2.1 Definitions

The definition section occurs before the first %%. It contains two things. First, any C code that must be inserted external to any function should appear in this section between the delimiters %{ and %}. Secondly, the definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, [4], [2]. (For a discussion of start conditions, see the Flex Manual ([2]), pages 13 - 18.) Name definitions have the form:

```
name definition
```

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to by using "name", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. [2]

2.2 Rules

The "lexical rules" section of a Flex specification contains a set of regular expressions and actions (C code) that are executed when the scanner matches the associated regular expression. It is of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line. [2]

2.3 Usercode/Auxiliary routines

The user code section is simply copied to ”lex.yy.c” (output file generated by Flex) verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second ’%%’ in the input file may be skipped, too.

2.4 Insertion of C Code

1. Any text written between %{ and %} in the definition section will be copied directly to the output program external to any procedure. [4]

2. Any text in the auxiliary procedures section will be copied directly to the output program at the end of the Flex code. [4]

3. Any code that follows a regular expression (by at least one space) in the action section (after the first %%) will be inserted at the appropriate place in the recognition procedure `yylex` and will be executed when a match of the corresponding regular expression occurs. [4]

4. In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors. [2]

2.5 A brief discussion of start condiditions

Start conditions is a mechanism of Flex that enables the use of conditional activation rules. In this section we will illustrate start conditions by discussing a small example. For further information, consult [2], pages 13-18.

Say, for example, we want a program that replaces a string in a file with the word `string`. In other words, as soon as we encountered a quotation mark, we want to remove all the text until we find the next quotation mark, and replaced the removed text with the word `string`. Here is a fragment of code that will accomplish that.

```
1
2  %s STRING
3
4  %%
5
6  /* Some other rules */
7  "\""        { BEGIN STRING; }
8  <STRING>[^"]        { printf( "string" ); BEGIN INITIAL; }
9  /* Some other rules */
```

On line 2, we define a start condition `STRING`. On line 7, we define a rule that is applicable if the lexer finds a quotation mark. The action of this rule will enable all rules that start with the prefix `STRING`. In our examples, it implies that the rule on line 8 is enabled. The rule on line 8 matches everyting until it find a quotation mark. Then it will print the word `string` and go into the `INITIAL` state, which is the default state of Flex. If the lexer is in the default state, the rules with the `STRING` prefixed, will not be active.

3 Flex operators and what they do

Table 1 gives a summary of the metacharacter conventions in Lex (taken from [4] and [3]).

Table 1: Metacharacter Conventions in Lex

Pattern	**Description**	**Meaning**
a		The character a
”a”		The character a, even if a is a metacharacter. For example, in the regular expression ”b+”, the sequence $b+$ should be matched exactly and not one or more repetitions of b (see the ”+” metacharacter explanation).
\a		The character a when a is a metacharacter
a*	Kleene Closure	Zero or more repetitions of a
a+	Iteration	One or more repetitions of a
a?	Option	An optional a
a|b	Union	a or b
ab	Concatenation	The character a followed by b
(ab)		ab itself. Parentheses group a series of regular expressions together into a new regular expression. For example (ab) represents the character sequence ab. Parentheses are useful when building up complex patterns with *, + and —.
[abc]		Any of the characters a, b or c
[a-d]		Any of the characters a, b, c or d
[^ab]		Any character except a or b
a{n}	Repeat	Is equivalent to n times the concatenation of a.
		Continued on next page

Table 1 – concluded from previous page

Pattern	Description	Meaning
a{n,m}		Is equivalent to at least n times and at most m times the concatenation of a.
.		Any character except a newline
{xxx}		The regular expression that the name xxx represents. For example, if we define the name "DIGIT" as `DIGIT [0-9]`, every time we write down {DIGIT} in our Flex specification, it will be replaced by [0-9].

4 Predefined/internal variables

This section summarises the most important predefined variables and functions available to the user.

char *yytext Holds the text of the current token

int yyleng The length of the current token

FILE *yyin The file which by default Flex reads from

void yyrestart(FILE *newFile) Change the value of `yyin`

For other predefined variables and functions, see [2].

5 An example

In this section we give a flex specification that generates a word count program (similar to the UNIX program *wc*. The example, as well as most of the descriptions, was adapted from [3].

The definition section for our word count example is:

```
%{
  int charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n
```

The section bracketed by "%{" and "%}" is C code which is copied verbatim into the lexer. The code block declares three variables used within the program to track the number of characters, words and lines encountered. [3]

Line 5 and line 6 are definitions. The first provides our description of a word: any non-empty combination of characters except space, tabs and newline. The second describes our end-of-line character, newline. We use these definitions in the second section of the Flex specification.

```
1  %%
2
3  {word}  { wordCount++; charCount += yyleng; }
4  {eol}   { charCount++; lineCount++; }
5  .       { charCount++; }
```

On line 3 we increment the value of `wordCount` if we recognized a word. Furthermore, our sample lexer uses the Flex internal variable `yyleng` to increment the value of `charCount`. `yyleng` contains the length of the string the lexer recognized. If it matched ”well-being”, `yyleng` would return 10.

On line 4, we specify the rule when a newline is encountered. The appropriate counters are incremented in such an event.

It is worth mentioning that Flex always tries to match the longest possible string. Thus, our sample lexer would recognize the string ”well-being” as a single word. If there are two possible rules that match the same length, the lexer uses the earlier rule in the Flex specification. Thus, the word, ”I” would be matched by the {`word`} rule, not by the . rule. [3]

Here is the full listing of our lexer (followed by a brief discussion of some outstanding issues):

```
1  %{
2   int charCount = 0, wordCount = 0, lineCount = 0;
3  %}
4
5  word [^ \t\n]+
6  eol  \n
7
8  %%
9
10 {word}  { wordCount++; charCount += yyleng; }
11 {eol}   { charCount++; lineCount++; }
12 .         charCount++;
13
14 %%
15 int main()
16 {
17 yylex();
```

```
18  printf( "\t%d %d %d\n", lineCount, wordCount, charCount );
19  }
```

On line 15 we declare the `main` function of our program. It calls the `yylex()` method that will be generated by Flex. After there is no more input available, it will print out the results as specified on line 18.

6 How to create our wordcount program

1. Save the Flex specification in a text file, say `wordcount.flex`.
2. Convert the Flex program file to a C program with the command `flex wordcount.flex`. If there is no error, a file named `lex.yy.c` will exist.
3. Compile this C file using by executing `gcc lex.yy.c -lfl -o wordcount`. The `lfl` option instructs the linker to use the routines in the Flex library if needed. This option is most of the time necessary in order for the linker to resolve all external references.
4. Run the wordcounter with the command `./wordcounter <filename>`, where filename is a file which words should be counted. To verify the output of WordCounter, use the UNIX program *wc*.

7 Conclusion

This was only a brief discussion of Flex. Consult the references if more information is required.

References

[1] DODDS, R., "Computer Science 354 Slides." 2002.

[2] Free Software Foundation, "Flex — a scanner generator." `http://www.gnu.org/software/flex/manual/`. February 2004.

[3] LEVINE, J. R., MASON, T., BROWN, D., *lex & yacc*. O'Reilly & Associates, 1992.

[4] LOUDEN, K. C., *Compiler Construction — Principles and Practice*. PWS Publishing Company, 1997.