Compiler Design

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language).

The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Lexical analysis

Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).

A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, though scanner is also a term for the first stage of a lexer.

lexical, from a Latinized form of Greek lexikos, means pertaining to words.

Flex and Bison

Flex and bison are tools designed for writers of compilers and interpreters, although they are also useful for many applications that will interest writers of other programs.

Any application that looks for patterns in its input or has an input or command language is a good candidate for flex and bison.

Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs.

In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the programming being done by Schmidt. They saw it both as a standalone tool and as a companion to Johnson's yacc. Lex also became quite popular, despite being relatively slow and buggy. (Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.)

In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written in ratfor (an extended Fortran popular at the time) and translated it into C, calling it flex, for "Fast Lexical Analyzer Generator." Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex. Flex is now a SourceForge project, still under the Berkeley license.

BSD licenses are a family of permissive free software licenses, imposing minimal restrictions on the use and redistribution of covered software.

lexical analysis and syntax analysis

The earliest compilers back in the 1950s used “utterly ad hoc” techniques to analyse the syntax of the source code of programs they were compiling. During the 1960s, the field got a lot of academic attention, and by the early 1970s, syntax analysis was a well understood field.

One of the key insights was to break the job into two parts: lexical analysis (also called lexing or scanning) and syntax analysis (or parsing).

Roughly speaking, scanning divides the input into meaningful chunks, called tokens, and parsing figures out how the tokens relate to each other. For example, consider this snippet of C code:

```
alpha = beta + gamma;
```

A scanner divides this into the tokens alpha, equal sign, beta, plus sign, gamma, and semicolon.

Then the parser determines that beta + gamma is an expression, and that the expression is assigned to alpha.

Scanners generally work by looking for patterns of characters in the input.

For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter followed by zero or more letters or digits, and the various operators are single characters or pairs of characters.

A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp.

These are the same kind of patterns that the editors ed and vi and the search program grep use to describe text to search for.

A flex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions.

Regular expressions

A regular expression, regex or regexp is a sequence of characters that define a search pattern.

Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis. Many programming languages provide regex capabilities.

Each character in a regular expression (that is, each character in the string describing its pattern) is either a metacharacter, having a special meaning, or a regular character that has a literal meaning.

For example, in the regex

a.

a is a literal character which matches just 'a', while '.'
is a meta character that matches every character except a newline.

Therefore, this regex matches, for example, 'a ', or 'ax', or 'a0'.

Together, metacharacters and literal characters can be used to identify text of a given pattern, or process a number of instances of it.

A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.

Flex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously, so it’s just as fast for 100 patterns as for one.

The internal form is known as a deterministic finite automaton (DFA). Fortunately, the only thing you really need to know about DFAs at this point is that they’re fast, and the speed is independent of the number or complexity of the patterns.

Much of this program should look familiar to C programmers, since most of it is C.

A flex program consists of three sections, separated by %% lines.

The first section contains declarations and option settings.

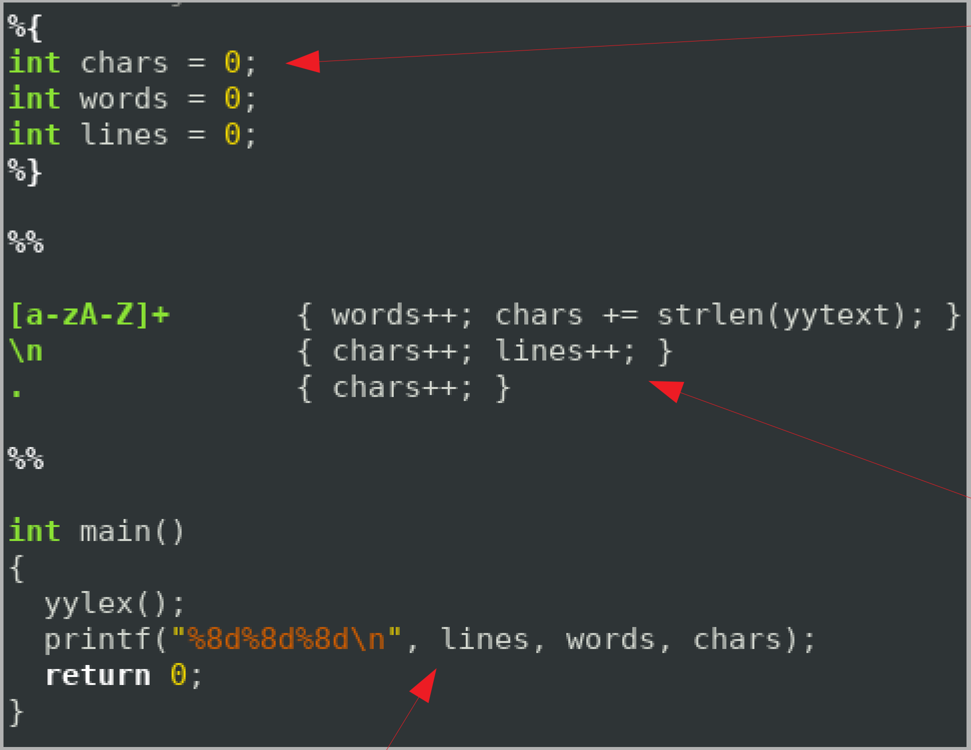
The second section is a list of patterns and actions, and the third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions.

setenv LANG C

The C locale is a special locale that is meant to be the simplest locale. You could also say that while the other locales are for humans, the C locale is for computers. In the C locale, characters are single bytes, the charset is ASCII (well, is not required to, but in practice will be in the systems most of us will ever get to use), the sorting order is based on the byte values, the language is usually US English (though for application messages (as opposed to things like month or day names or messages by system libraries), it's at the discretion of the application author) and things like currency symbols are not defined.

ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL



```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%

[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }

%%

int main()
{
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
  return 0;
}
```

The first section contains declarations and option settings.

The second section is a list of patterns and actions

The third section is C code that is copied to the generated scanner

In this program, there are only three patterns.

The first one, [a-zA-Z]+ , matches a word. The characters in brackets, known as a character class, match any single upper or lower case letter, and the + sign means to match one or more of the preceding thing, which here means a string of letters or a word.
The action code updates the number of words and characters seen.
In any flex action, the variable yytext is set to point to the input text that the pattern just matched.
In this case, all we care about is how many characters it was so we can update the character count appropriately.

The second pattern, \n , just matches a new line. The action updates the number of lines and characters

The final pattern is a dot, which is regex-ese for any character except \n.
The action updates the number of characters. And that's all the patterns we need.

In the declaration section, code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file. In this case it just sets up variables for lines, words, and characters.

In the second section, each pattern is at the beginning of a line, followed by the C code to execute when the pattern matches. The C code can be one statement or possibly a multiline block in braces, { } .

Each pattern must start at the beginning of the line, since flex considers any line that starts with whitespace to be code to be copied into the generated C program.

The four rules for matching tokens are:

Characters are only matched once. That is, each character is matched by only one pattern.

Longest matching string gets matched first. That is, if one pattern matches "zin" and a later pattern matches "zinjanthropus" the second pattern is the one that matches.

If same length of matching string then first rule matches.

If no pattern matches then the character is printed to standard output.

Most flex programs are quite ambiguous, with multiple patterns that can match the same input.

In most flex programs multiple patterns can match the same input.

Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in the vast majority of cases. Consider this snippet from a scanner for C source code:

```
"+"       { return ADD; }
"="       { return ASSIGN; }
"+="      { return ASSIGNADD; }
"if"      { return KEYWORDIF; }
"else"    { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
```

For the first three patterns, the string += is matched as one token, since += is longer than + . For the last three patterns, so long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly.

The default rule

```
.|\n    ECHO;
```

Remember that . in (f)lex does not match a newline.
To disable the default rule, use the declaration
%option nodefault

Once you do that, you will get a warning if your rules do not cover every eventuality.
If you ignore the warning and use the generated scanner, it will stop with a fatal error if the input does not match any pattern.

Since you hardly ever want the default rule, I recommend always using the above %option.

If a dot matches anything, won’t it also match the letters the first pattern is supposed to match?

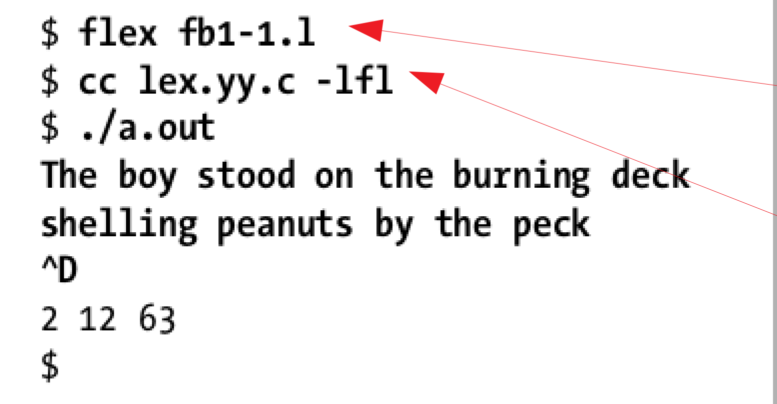
It does, but flex breaks a tie by preferring longer matches, and if two patterns match the same thing, it prefers the pattern that appears first in the flex program.

Lex's own default rule matches one character and prints it.

The C code at the end is a main program calls yylex() , the name that flex gives to the scanner routine, and then prints the results.

In the absence of any other arrangements, the scanner reads from the standard input.

To read from a file use yyin=fopen("abc.txt","r");



```
$ flex fb1-1.l
$ cc lex.yy.c -lfl
$ ./a.out
The boy stood on the burning deck
shelling peanuts by the peck
^D
2 12 63
$
```

First we tell flex to translate our program, and in classic Unix fashion since there are no errors, it does so and says nothing.

Then we compile lex.yy.c , the C program it generated; link it with the flex library, -lf

The actual wc program uses a slightly different definition of a word, a string of non-whitespace characters.

Once we look up what all the whitespace characters are, we need only replace the line that matches words with one that matches a string of non-whitespace characters:

```
[^ \t\n\r\f\v]+     { words++; chars += strlen(yytext); }
```

The ^ at the beginning of the character class means to match any character other than the ones in the class, and the + once again means to match one or more of the preceding patterns.

This demonstrates one of flex's strengths—it's easy to make small changes to patterns and let flex worry about how they might affect the generated code.

\t is a horizontal tab, \v is a vertical tab and \r is a carriage return.