



■ Generate a scanner with Flex

★★★★☆

[30 mn of reading - published 04/05/2004 10:02:37 - Target : Débutant]

Author

Seng-Houth HENG

Student-Engineer Supinfo Paris
SUPINFO graduate year 2005

- ▶ Write to the person
- ▶ All projects of the same author
- ▶ Mini-CV of the author



Introduction



Introduction

This article has been made to present the power of Flex : Fast Lexical Analyzer Generator. Flex is part of a family of lexical analyze generators, each one generating lexical analyzers (also called scanners) in a particular programming language. It is the C-code generator version that is studied there.

Willing to be more accessible to beginners than the online manual, this text intends to guide you during your first steps with Flex.

Frequently used associated with Bison in a programming environment using Free softwares, Flex had first been made to accelerate the implementation of the first stage of a compiler. Nevertheless, it is also useful to make personalized tools to suit specific needs.

This article can be read like a tutorial to learn Flex functions as fast as possible.

Introduction



1. A scanner

1.1. Quick start

Flex is a tool to generate a scanner. A scanner is a program that matches the input text with patterns which are associated with user-defined actions. With the .lex file specified on the command line, Flex creates an output file `lex.yy.c` which contains a function named `yylex()`. This file is then compiled with `gcc` to get, at last, the scanner.

Here is an example which replaces every phrase "<date>" (without the quotes but with the lower and greater symbols) by the current date and time. Everything below is to be typed into a flat text file called `example.lex` :

```
%option noyywrap

%%
"<date>" {
    time_t t;
    t = time();
    printf("%s", ctime(&t));
}

%%
#include <time.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
    return 0;
}
```

The first line specifies that the `yywrap()` function must not be generated. This function is not necessary within the brief examples presented there.

The section that starts with the double percentage symbol ("%") is the rules section. There is first the pattern to match, then separated by a space, the action to execute when the pattern is matched. The action is there spread into several lines within a C-style block. The next section, separated with the rules by another double percentage symbol, is copied character by character into the output C file. These few lines replace the default scanner input file pointer (`stdin`) with the first argument of the program.

The default action when a text can not be matched by user-defined patterns is to output the text onto `stdout`. So the global behavior of the program is to copy the entire input file to the output, replacing only every <date> pattern with local date and time.

Given the flat text file called `input` that contains the sentence :

```
current date and time is <date>
```

The commands to be executed in the shell are :

```
bash$ flex example.lex
bash$ gcc -o example lex.yy.c
bash$ ./example input
```

The screen displays this line :

```
current date and time is Mon May 3 14:18:18 2004
```

1.2. Structure of a .lex file

The .lex file has a very simple structure :

```
definitions
%%
rules
%%
user code
```

The definitions section contains substitution declarations, like macros in C. The structure of a definition is :

```
name definition
```

The name is an alphanumerical string, underscore ("_") and hyphens ("-") characters are allowed, but the first character must be a letter or an underscore. In the rules (see below), every occurrence of {name} is replaced by Flex with (definition), braces and parenthesis included. For example, with the following definition :

```
DIGIT [0-9]
```

every occurrence of {DIGIT} in patterns are replaced with ([0-9]). To better understand this notation between brackets and learn a built-in notation for a single digit, see the chapter dealing with Rules.

This definitions section can also contain subscanner declarations, see the chapter about A subscanner.

The rules section contains every rule of the scanner to be generated. A rule is always made of a pattern and an action :

```
pattern action
```

The pattern must start at the first column, while the action must start at the same line than the pattern. More details in the next chapter : Rules.

In the definitions and rules sections, code can be inserted "as is" in the output file lex.yy.c. Any indented text (not starting at the first column) or any text put between "%{" and "%}" (placed on lines by themselves) will be copied.

Any text inserted this way in the rules section, before any rule, is placed at the beginning of the yylex() function (useful for local variables declaration, then used in the actions); if the text is inserted elsewhere in the rules section, the result is undefined. Any text inserted this way in the definitions section is placed in the global scope, outside any function.

The last technique is particularly useful for pre-processor instructions :

```
%{
#include <stdio.h>
#include <time.h>
%}
%%
/* rules ... */
```

At last, the user-code section is copied into the file lex.yy.c. This section is optionnal, and so is the second double percentage symbol that announces it, if this section is missing. This section can be used to write functions called by an action or, as in the very first example, to write a main() function which calls the yylex() scanner.

1.3. Scanner's variables

The scanner is generated into the yylex() function. A few variables are also declared along with this function :

- **FILE *yyin** : the file pointer from which the scanner reads input text. Global scope; by default, yyin is affected with stdin.
- **FILE *yyout** : the output file pointer in which is made the default action (that is copying into it any unmatched text). Global scope; by default, yyout is affected with stdout.
- **char *yytext** : text recognized by the current pattern. Its scope is local, so the yytext variable can only be used in actions.
- **int yyleng** : length of the yytext string. Its scope is local, so the yyleng variable, as yytext, can only be used in actions.

2. Rules

2.1. Patterns

Within a rule, a pattern starts from the first column of the line and stops at the first whitespace (space, tabulation or even line feed, if the associated action is empty) not put between quotes or just after a backslash character ('\').

The writing of each pattern must follow a strict syntax : regular expressions' one. The table below summarizes each possible syntax element of a regular expression, from the highest precedence to the lowest :

Syntax	Recognized expression
(r)	the regular expression "r", the parenthesis are used to override precedence
c	a letter 'c'
.	any letter, except a line feed ('\n')
[abc]	a character class : any character put between the brackets; in this case a letter 'a', 'b' or 'c'; in a character class, any special character like the brace loses its operator meaning, except '\', '-' and ']', which must then be written just after a backslash to have a literal meaning
[Abg-kM]	a character class with a range in it; in this case a letter 'A', 'b', 'M', or a letter between 'g' and 'k' ('g', 'h', 'i', 'j' or 'k')
[^cR-Wz]	a negated character class : any character except those put between the brackets; in this case any character except 'c', 'z' and the letters between 'R' and 'W'; the character '^' must be the first character of the class to obtain a negated character class
c*	zero, one or more letters 'c'
c+	one or more letters 'c'
c?	zero or one letter 'c'
c{5}	exactly five letters 'c'
c{3,6}	three, four, five or six letters 'c'
c{2,}	two or more letters 'c'
{name}	the definition of "name" in the definitions section
\x	if 'x' is one of the letters 'a', 'b', 'f', 'n', 'r', 't' or 'v', then it is the ANSI-C interpretation of '\x'; else it is exactly the character 'x' (useful to specify a quote or a star, with their literal meaning but not their special operator meaning)
\123	the character with an ASCII octal value of 123
\x6a	the character with an ASCII hexadecimal value of 6a
rs	the concatenation of the regular expressions "r" and "s"
r s	the expression "r" or the expression "s"
r/s	the expression "r" only if it is followed by the expression "s"; the expression "s" is taken into account when applying the longest match rule (see below), but it is invisible to the associated action, since the action sees only "r"
^c	a letter 'c' only if it is at the beginning of a line : whether at the beginning of the file or just after a line feed
c\$	a letter 'c' only if it is at the end of a line; equivalent to "c\n" or "c\r\n" depending on the system
<<EOF>>	end of input file

Beware of the concatenation : it is a form that has a lower precedence than some operators. For example, this pattern :

```
dog|cat*
```

is interpreted this way by Flex :

```
(dog) | (ca (t*))
```

There are also special character classes :

```
[ :alnum: ]
[ :alpha: ]
[ :blank: ]
[ :cntrl: ]
[ :digit: ]
[ :graph: ]
[ :lower: ]
[ :print: ]
[ :punct: ]
[ :space: ]
[ :upper: ]
[ :xdigit: ]
```

These classes recognize characters that return a non-zero value when they are given as arguments to the corresponding isXXX() function in C, for example isdigit().

Last statements about patterns :

When two patterns match a text, the pattern that matches the longest text applies.

When two patterns match the same text, so the same length for both, the pattern that appears first in the .lex file applies.

When no pattern matches a character, it is given to the default rule : it is copied to the output. So it is assumed that the following rule is added at the end of the rules section of every .lex file :

```
.| \n    fprintf(yyout, "%s", yytext);
```

2.2. Actions

The action starts where the corresponding pattern ends : from the first whitespace to the end of the line. That's why an empty action eats the text recognized by the pattern.

If the action contains an opening brace, the action ends only after the next closing brace and can span over multiple lines.

An action which is only the character '|' means : "the same action than the next rule's one".

An action can even call the instruction return, as required when Flex is used in combination with Bison. In this case, the execution flow exits from the yylex() function. Every time the yylex() function is called, the scanner continues its analysis starting from the point where it stopped.

A few special statements can be put in the actions :

- ◆ **ECHO** : equivalent to `fprintf(yyout, "%s", yytext)`
- ◆ **BEGIN(XXX)** : starts the XXX subscanner (see next chapter on A subscanner)
- ◆ **REJECT** : jumps to the next choice of pattern, for example if the current pattern has been chosen against another, following the above rules (most of the time, it is about the rule of first appearance in the .lex file).

3. A subscanner

3.1. Syntax

A subscanner is a scanner included in the main scanner. It is used to apply specific rules in special cases that occur during the main scan.

Subscanners are defined in the definitions section, with a line starting with "%s" for inclusive subscanners and "%x" for exclusive subscanners. The difference between them two is that an exclusive subscanner provides rules only for its own subscanner while an inclusive subscanner provides rules that can also be used in the main subscanner.

A short example is better than a long speech :

```
%x STRING
%%
\"          fprintf(yyout, "("); BEGIN(STRING);
<STRING>[^"]*  ECHO;
<STRING>\\"    fprintf(yyout, "..."); BEGIN(INITIAL);
```

The generated scanner with those specifications will replace each text between quotes in the input text with a text followed by three dots and enclosed into braces.

The first line declares an exclusive subscanner named STRING.

The first and only rule of the main scanner is to start the STRING subscanner when a quote is detected, after having written an opening brace.

The first rule of the STRING subscanner is to rewrite the read string.

The second and last rule of the STRING subscanner is to start the main scanner, after having written a closing brace.

Here are some syntax points to know about subscanners :

- ◆ A subscanner is called with the instruction `BEGIN(SUBSCANNER)`.
- ◆ The main scanner is actually named `INITIAL`. It is then possible to get back to the main scanner with the instruction `BEGIN(INITIAL)`.
- ◆ The pattern `<SS>pat` belongs to the SS subscanner.
- ◆ The pattern `<SS,AA>pat` belongs to the subscanners SS and AA.
- ◆ The pattern `<*>pat` belongs to every subscanner, even exclusive ones, even the main scanner.
- ◆ The pattern `<SS>pat`, SS being an inclusive subscanner, is equivalent to the pattern `<SS,INITIAL>pat`, SS being an exclusive subscanner.
- ◆ A more convenient syntax is available when writting several rules belonging to a subscanner :

```
<SS>{
    pattern1  action1;
    pattern2  action2;
}
```

Is equivalent to :

```
<SS>pattern1  action1;
<SS>pattern2  action2;
```

3.2. Usage

Assuming that a scanner removing every comment in a C source file is required, given that the source file is correct from the C syntax point of view. At first, it seems to be simple to write :

```
%%  
"/**(.|\n)***/" /* comment discarded */
```

After a while, it is easy to notice that the generated scanner does not fit the needs. With the following input file :

```
#include <stdio.h>  
/* comment #1 */  
#include <time.h>  
/* comment #2 */  
int main(void) {}
```

The following output is produced after the scanner's analysis :

```
#include <stdio.h>  
int main(void) {}
```

Where is the line including time.h ? It has already been said : Flex tries to match the longest string with any pattern. With the preceding input file, the longest match starts at the beginning of the first comment and ends at the end of the second comment ! Flex has strictly applied the specifications of the scanner to be generated.

Is Flex poorly programmed to match each time the longest text corresponding to patterns this way ? No, the rule `a+` requires this kind of implementation of regular expressions. The longest match is an obligatory rule to obtain a deterministic algorithm.

Here is the most understandable way to remove C comments :

```
%x COMMENT  
%%  
"/**"      BEGIN(COMMENT);  
<COMMENT>(  
    "**/"    BEGIN(INITIAL);  
    .|\n  
)
```

Generally, a subscanner is well suited when it is about matching the shortest text but not the longest. However, it is not the unique usage of a subscanner : it remains a powerful tool among Flex capabilities, able to respond to more complex needs (that will not be studied yet).

3. A subscanner



4. Extended operation

4.1. Using Flex with Bison

Bison is a syntax analyzer generator. A syntax analyzer, also called a parser, is a program that checks if a text grammatically respects the given rules.

Bison calls the `yylex()` function to get the next word (called a token) from the input file. The `yylex()` function must return the token type and a value representing this token in the global variable `yylval`.

To use Flex with Bison, the `-d` option must be specified to Bison. It will produce a `y.tab.h` file containing the definition, among others, of all of the tokens given in input to Bison. This header file will then be included in Flex in order to combine those two programs. The following is a `.lex` file that could be used with Bison :

```
%{
#include "y.tab.h"
%}
%%
[:digit:]+    yyval = atoi(yytext); return TOK_NUMBER;
```

4.2. Flex' options

Here are some options for Flex in command line :

- ◆ `-i` : the generated scanner is case-insensitive. Patterns and recognized text can be whether in upper or lower case, the scanner will not bother. However, the text in `yytext` will remain unchanged (it will not be put all in lower or upper case).
- ◆ `-w` : warning messages are not displayed.
- ◆ `-o out.c` : the generated scanner is written to the `out.c` file instead of `lex.yy.c`.
- ◆ `-P flex` : all of the functions and variables of the generated scanner have the `flex` prefix instead of `yy`.

5. References

1. ***Flex online manual***, Vern Paxson with the help of various authors
http://www.gnu.org/software/flex/manual/html_mono/flex.html
An almost exhaustive manual, even if it has been written a long time ago (1995 !). It needs a few corrections : there are some typing mistakes and conversion errors.
2. ***modern compiler implementation in ML***, Andrew W. Appel, Cambridge University Press
<http://www.cs.princeton.edu/~appel/modern/>
A beautiful book that I must recommend : it is almost the Bible of programming languages compilation. Available in English only, there are three versions : with implementation code in ML, C or Java.

Conclusion

Flex is a powerful tool to generate scanners. It would take an infinite time to explore all of its capabilities.

So this article built the basis that allow anybody to develop very quickly text scanners with extended functionalities. Nevertheless, in order to go deeper in this matter, the entire online manual must be read first. Then it is recommended to browse the various newsgroups about compilation to find a helpful person that can enlighten some dark aspects of Flex.

Who knows ? I could be this helpful person...

Conclusion 