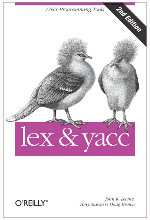
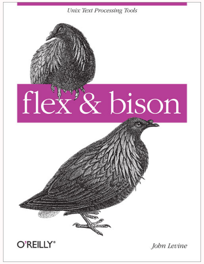
The Flex Scanner Generator

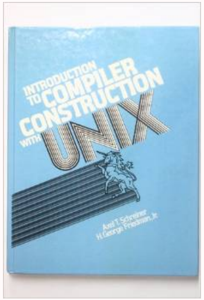
August 29, 2018

Brian A. Malloy



1. Text Books





Go Back
Full Screen
Quit

2. Overview of Scanners

- A scanner usually reads input and matches patterns
- Scanners can be used in editing, software testing, and parsing.
- Historically, parsing was broken into phases; scanning and parsing were the first 2 phases.
- The scanner provided a sequence of tokens to the parser by reading characters from the input stream and building the tokens.

2.1. Scanner tasks during parsing

- Find extraneous chars,
- store names in symbol table,
- strip white space,
- recognize tokens.

2.2. How to build a scanner?

- Write it by hand,
- use a tool,
- We choose the second option: flex.

3. flex

- flex is a scanner generator
- flex reads the input
- scans for patterns, expressed as regular expressions, and takes corresponding action
- writes all input not matched by a pattern

3.1. Flex sections

```
%{
  C/C++ code
%}
Scanner declarations

%%
Token definitions and semantic actions

%%
C/C++ subroutines
(need prototype in C/C++ code section
```

A Simple Flex Example:

```
%{ // what does this program do?
%}

%%
"a"

%%
int yywrap() { return 1; }
```

3.2. yywrap()

- called on eof by yylex;
  - if yywrap returns 1, then flex terminates;
  - Otherwise, flex makes another pass.

```
int yywrap() {
  std::cout << "terminating flex" << std::endl;
  return 1;
}
```

3.3. What flex generates:

- flex generates yylex, a C fn that implements a DFA, based on the flex specification
- yylex reads from **stdin** and returns a number (token) associated with the matched pattern.
- To match more than one pattern, call yylex repeatedly (yylex returns 0 at eof):

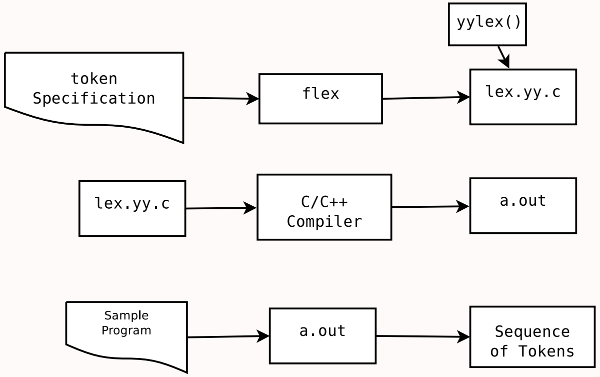
```
int main() {
  int token = yylex();
  while ( token ) {
    std::cout << "token: " << token << std::endl;
    token = yylex();
  }
}
```

3.4. flex can read from a file

```
#include <iostream>
#include <fstream>

void main(int argc, char * argv[]) {
  if (argc != 2) {
    cout << "usage: " << argv[0] << "<filename>\n";
  }
  FILE * infile; // Must use C-style I/O
  infile = fopen(argv[1], "r");
  if (!infile) {
    cout << "Could not open: " <<  argv[1] << endl;
  }
  yyin = infile;
  yylex();
}
```

4. Process for generating flex scanner



4.1. Convenient to use a Makefile

```
CCC = g++
LEX = flex
CXXFLAGS=-g -W -Wall -std=c++11 -Weffc++ -Wextra -O0

LEXFLAGS = -Wno-unused
FLEXDEBUG = -d
OBJS = main.o lex.yy.o

run: $(OBJS)
    $(CCC) $(CFLAGS) -o run $(OBJS)
main.o: main.cpp
    $(CCC) $(CFLAGS) -c main.cpp
lex.yy.c: scan.l
    $(LEX) $(FLEXDEBUG) -i scan.l
lex.yy.o: lex.yy.c
    $(CCC) $(CFLAGS) $(LEXFLAGS) -c lex.yy.c
```

Go Back
Full Screen
Quit

5. Regular Expressions

Flex characters have special meanings:

1. . matches any single char except newline
2. [] character class, matches any char w/in brackets; if first char is $\wedge$ it matches any char except those in bracket.
3. $\wedge$ matches the beginning of a line as first char in regular expr.
4. $ matches the end of line as last char
5. \ escapes metacharacters
6. * matches 0 or more
7. + matches 1 or more
8. ? matches 0 or 1 occurrence
9. | is alternation

10. () group
    ****
11. {} if numbers, specifies how many ($A\{1,3\}$ matches 1 to 3 consecutive A's), and ($A\{2\}$ matches 2 consecutive A's)
12. (?s:pattern) apply option s while interpreting pattern. frequently used options:
    - $i \Rightarrow$ case insensitive;
    - $-i \Rightarrow$ case sensitive
13. `(?# comment )` $\Rightarrow$ comments in specs

Some Pattern Examples:

```
%{
#include <iostream>
%}
letter      [a-zA-Z]

%%
(?i:c)        { std::cout << "upper or lower case c" << std::endl; }
(?-i:d)       { std::cout << "only lower case d" << std::endl; }
(?-i:E)       { std::cout << "only upper case E" << std::endl; }
c             { std::cout << "never gets here!" << std::endl;  }
ab            { std::cout << "Matching ab" << std::endl;       }
%{ Question: %}
(?# why does it choose the next rule over first 3?)
^0            { std::cout << "match 0 at eol" << std::endl;    }
{letter}$     { std::cout << "match  1 letter at eol" << std::endl; }
{letter}*     { std::cout << "bunches of letters" << std::endl;}
(?#: this is a commment)
.             { /* matches everything except \n */ }
```

5.1. RE Examples

a+b+	1 or more a's, followed by 1 or more b's
a|b	either an a or a b
x	the character x
`[abc]`	a, b, or c
`[0-9]+`	an integer
`[-+]?[0-9]+`	integer with opt sign (- must come 1st)
`[ \t\n]`	whitespace
`[mM]`	use this rather than (?i:m)
{*word*}	whatever *word* is defined as
∧r	an r, only at begin of line
r$	an r, only at end of line
r{3}	exactly 3 r's
r{1,3}	1 to 3 r's
r{2,}	2 or more r's

6. Ambiguity

- If multiple patterns match a given input:
  - Match longest string,
  - In case of tie, match first pattern in specification.

```
%%
[0-9]+    { std::cout << "matched 9" << std::endl; }
9         { std::cout << "no way!" << std::endl; }
```

7. Start States

- Permits control of what gets matched
- \x defines the start state
- When scanner is in a state, it can only match the patterns specified in that state.
- Can define as many start states as needed
- The macro BEGIN switches states
- BEGIN(INITIAL) or BEGIN(0) return to start state

7.1. C Comments

```
%x COMMENT

%%
"/*"             { BEGIN(COMMENT); ++comments; }
<COMMENT>"*/"    { BEGIN(0); do_newline(); }
<COMMENT>\n      { do_newline(); }
<COMMENT>.       { ; }
```

8. Debugging Flex

- The -d flag, shown on slide 4.1, tells flex to go into debug mode:
  `flex -d scan.l`
- Flex will then print the rules that are matched:

```
for a+b+ on line 12, and \n on line 14:

aaab
--accepting rule at line 12 ("aaab")
match: aaab
--accepting rule at line 14 ("
")
```