

Lex and Flex

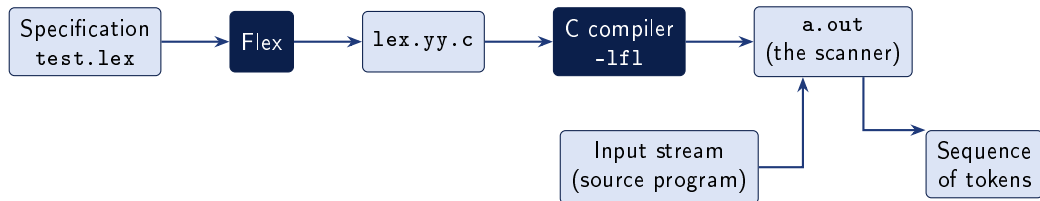
Compiler Construction, Lecture 4

K.R. Chowdhary, Professor of CS

July 19, 2019 | Fall 2019

Lexical Analyser Generator: Workflow

Flex (Fast LEXical analyser generator) builds scanners from regular-expression specifications. Written by Mike Lesk and Eric Schmidt in 1975; now open source.



Commands

```
$ flex test.lex                                # outputs lex.yy.c
$ gcc lex.yy.c -lfl                             # compile and link with the flex library
$ ./a.out                                       # type input, end with Ctrl-D
```

Structure of a Lex Program

Three sections

declarations

%%

translation rules

%%

auxiliary functions

Rule form: *Pattern* {Action}

- **Declarations:** variables, manifest constants (token names) and regular definitions. Text inside `%{ ... %}` is copied straight into `lex.yy.c`
- **Rules:** each pattern is a regular expression, which may use regular definitions in braces; the action is C code
- **Auxiliary functions:** copied into `lex.yy.c` and used by actions (e.g. `installID()`)

Built-in names

`yylex()`: the scanner routine, called by the parser
`yytext`, `yylen`: start and length of the lexeme
`yyval`: passes extra information about the lexeme

Example 4.1: A `wc` Clone

test.lex

```
%{
int chars = 0;
int words = 0;
int lines = 0;
}%
%%
[a-zA-Z]+ {words++; chars += strlen(yytext);}
\n        {chars++; lines++;}
.         {chars++;}
%%
int main()
{
    yylex();
    printf("%d %d %d\n", lines, words, chars);
    return 0;
}
```

- Counters live in the `%{ %}` block
- Letter runs count as words and add `yytext` length to `chars`
- Newline and any other character update the counts
- `main()` calls `yylex()` once, then prints `lines`, `words`, `chars`

```
$ ./a.out < test.lex
prints 21 33 266
```

Example 4.2: Recognising Tokens

test.lex

```
%{ /* constants: LT EQ LE GT GE NE IF THEN ELSE ID NUMBER RELOP */ %}  
delim  [ \t\n]  
ws     {delim}+  
letter [A-Za-z]  
digit  [0-9]  
id      {letter}({letter}|{digit})*  
number  {digit}+(\.{digit}+)?(E[+-]?{digit}+)?  
%%  
{ws}    { /* no action and no return */}  
if       {return(IF);}   
then     {return(THEN);}   
else     {return(ELSE);}   
{id}     {yylval = (int) installID(); return(ID);}   
{number} {yylval = (int) installNum(); return(NUMBER);}   
"<"     {yylval = LT; return(RELOP);} /* ... */  
%%  
int installID() { /* lexeme at yytext, length yyleng */}  
int installNum() { /* like installID, separate table */}
```

- `delim` is a character class;
`ws` is one or more delimiters
- Whitespace has no action, so nothing goes to the parser
- `installID()` puts the lexeme in the symbol table and returns a pointer, stored in `yylval`
- `installNum()` does the same for numbers

How Lex Resolves Matching Conflicts

Two rules

1. Prefer the **longest prefix** over a shorter one
2. If the longest prefix matches several patterns, prefer the pattern **listed first**

Consequences:

- `>=` is one lexeme, not two
- Identifiers absorb all following letters and digits
- Keywords are reserved only if listed *before* `id`

- For `if` followed by a non-letter, the scanner returns `IF`; `ifx` becomes an `ID`
- Lex reads one character past the lexeme, then **retracts** so only the lexeme is consumed
- A pattern can require a following character, e.g. a semicolon in `C`

Whitespace matches `ws`, has no action, and scanning simply continues.

Exercises

1. Add the C keyword `switch` to Example 4.2
2. Allow hyphen and underscore as extra characters
3. Copy a file, replacing each tab by `\t`
4. Copy a file, replacing each `int` by `float`
5. Define token, token name, lexeme, attribute; give examples
6. Find lexemes and tokens in `int x; x = 100;`
7. Explain the two matching conflicts and their resolution
8. Flex spec and finite automaton for identifiers and integers

Resources

- epaperpress.com/lexandyacc
- dinosaur.compilertools.net
- Dr. Dobb's: Lex and Yacc