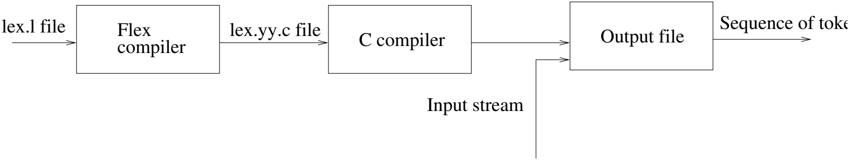
Towards automating lexical analysis

Sudakshina Dutta

Indian Institute of Technology Goa

9^{th} February, 2022

- Flex and Bison help in writing programs that transform structured input
- There are two tasks
    1. Dividing the input into meaningful unit
    2. Discovering relationships among the unit
- Division into units (tokens) is known as lexical analysis
- *F*lex helps us by taking a set of possible patterns and producing a C routine which is called *L*exical Analyzer or a *L*exer



- The token description that flex uses are regular expressions
- As the input is divided into tokens, a program often needs to find relationship among the token
- It is done by grammar rules
- Bison takes the grammar and produces parsing routines written in C

- A flex file looks like the following

```
··· definitions ···

%%

··· rules ···

%%

··· user subroutine ···
```

An example flex program

```
%{
int charCount = 0, lineCount = 0;
}%
%%
. {charCount++;}
\n {lineCount++; charCount++;}
%%
int main()
{
    yylex();
    printf("There were %d characters in %d lines \n", charCount, lineCount);
    return0;
}
```

Compilation steps

- Compilation steps :
  1. flex word_count.l
  2. cc lex.yy.c -o first -lfl
- Flex specification is translated into a C source file which is called lex.yy.c
- It is compiled and linked with the flex library -lfl
- Lex always tries to match the longest possible strings; but if there are two rules which match the strings of the same length, the first rule is considered
- **Portability** Flex and the C compiler need not run on the same machine

Definition Section

- It introduces any initial C program code we want copied into the final program
- Definitions are like macros and have the following form
  1. digit {0-9}
  2. number $\{digit\}\ \{digit\}^*$
- Included code is all code included between %{ and %}

  %{

  $float\ \ number;$

  $int\ \ count = 0;$

  %}

Rules and User Subroutine Section

- Rules Section
  - Each rule is made up of two parts: a pattern and an action, separated by whitespace
    {pattern} {C code}
  - The C-code is copied verbatim to the generated C-file
  - The patterns are UNIX-style regular expressions

    —A slightly extended version of the same expression used by the tool grep, sed, awk
  - Patterns are converted to NFA, then it is converted to DFA and it is optimized
  - The action is executed when the pattern is recognized
- User Subroutine Section
  - It consists of any legal C code
  - This section is copied after the end of the flex generated code

Some keywords

- Default action is to copy input to output
- **yytext :** contains the text matched against a pattern. ECHO is used to copy yytext
- **yyleng :** provides the number of characters matched
- Flex tries to find the rules where the longest match is performed
- **yywrap** When yylex() reaches the end of its input file, it calls yywrap();it returns the value 0 or 1
  1. If the value is 1, then the input is read and no more input is left
  2. If the value is 0, the lexer assumes that there is another file from where reading can be continued
- Overloading yywrap() is possible

Regular expression

- **.** : Matches any character except the newline characters
- ***** : Matches zero or more copies of the preceding expression
- [] : It matches any character within the bracket. If the first character is circumflex, then anything other than the remaining characters will be matched. "-" in these brackets depict range
- **$** : Matches the last character of the regular expression
- {} : Indicates how many times the previous pattern can appear. For example : $A\{1,3\}$ implies that A can appear 3 times. Also it refers to substitution of patterns for a name

Regular expression

- **+** : Matches one or more occurrence of the preceding regular expression
  - $[0-9]+$
- **?** : Matches one or more occurrence of the preceding regular expression
  - $-?[0-9]+$
- $|$: Matches either of the regular expression
  - "if"|"then"|"else"

Examples of Regular Expressions

- $[0-9]$: A single integer from 0 to 9
- $[0-9]+$: Any positive integer
- $[0-9]^*$: No integer or any integer
- $-?[0-9]+$: Optional unary minus is allowed for any integer
- $[0-9]^*\backslash.[0-9]+$: Matches the pattern like 1.5
- $-?(([0-9]+)|([0-9]^*\backslash.[0-9]+)([eE][-+]?[0-9]+)?)$: Exponent part is optional

An example program

```
%{
int hex = 0; int oct = 0; int regular = 0;
}%

letter [a-zA-Z_]
digit [0-9]
digits {digit}+
digit_oct [0-7]
digit_hex [0-9A-F]
int_qualifier [uUlL]
blanks [ \t]+
identifier {letter}({letter}|{digit})*
integer {digits}{int_qualifier}?
hex_const 0[xX]{digit_hex}+{int_qualifier}?
oct_const 0{digit_oct}+{int_qualifier}?

%%
```

```
if {printf("reserved word : %s\n", yytext);}
else {printf("reserved word : %s\n", yytext);}
while {printf("reserved word : %s\n", yytext);}
switch {printf("reserved word : %s\n", yytext);}
{identifier} {printf("identifier : %s\n", yytext);}
{hex_const} {sscanf(yytext, "%i", &hex);
                    printf("hexconstant : %s = %i\n", yytext, hex);}
{oct_const} {sscanf(yytext, "%i", &oct);
                    printf("octconstant : %s = %i\n", yytext, oct);}
{integer} {sscanf(yytext, "%i", &regular);
                    printf("integer : %s = %i\n", yytext, regular);}
.|\n                        ;


%%
yywrap(){}
int main(){yylex();}
```

Study Materials

- http://web.mit.edu/gnu/doc/html/ flex_1.html
- Lex & Yacc by John R. Levine, Tony Mason and Doug Brown (O'REILLY)
- Any tutorial from internet