Lecture 4 : Lexical Analyzer using Flex
Compiler Design (CS 3007)

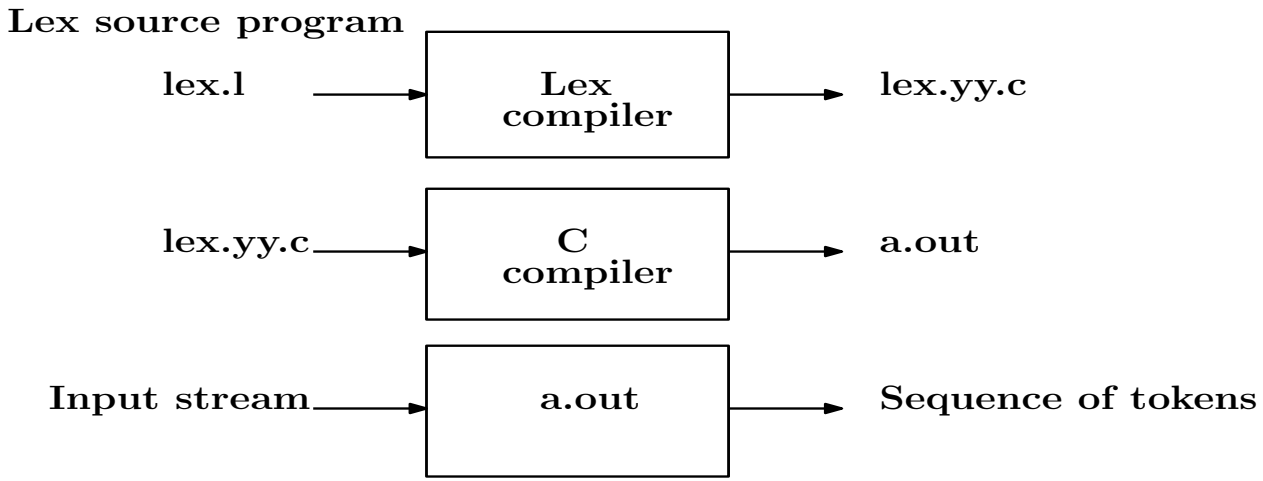
S. Pyne[1] & T. K. Mishra[2]

Assistant Professor[1,2]
Computer Science and Engineering Department
National Institute of Technology Rourkela
{pynes,mishrat}@nitrkl.ac.in

August 13, 2020

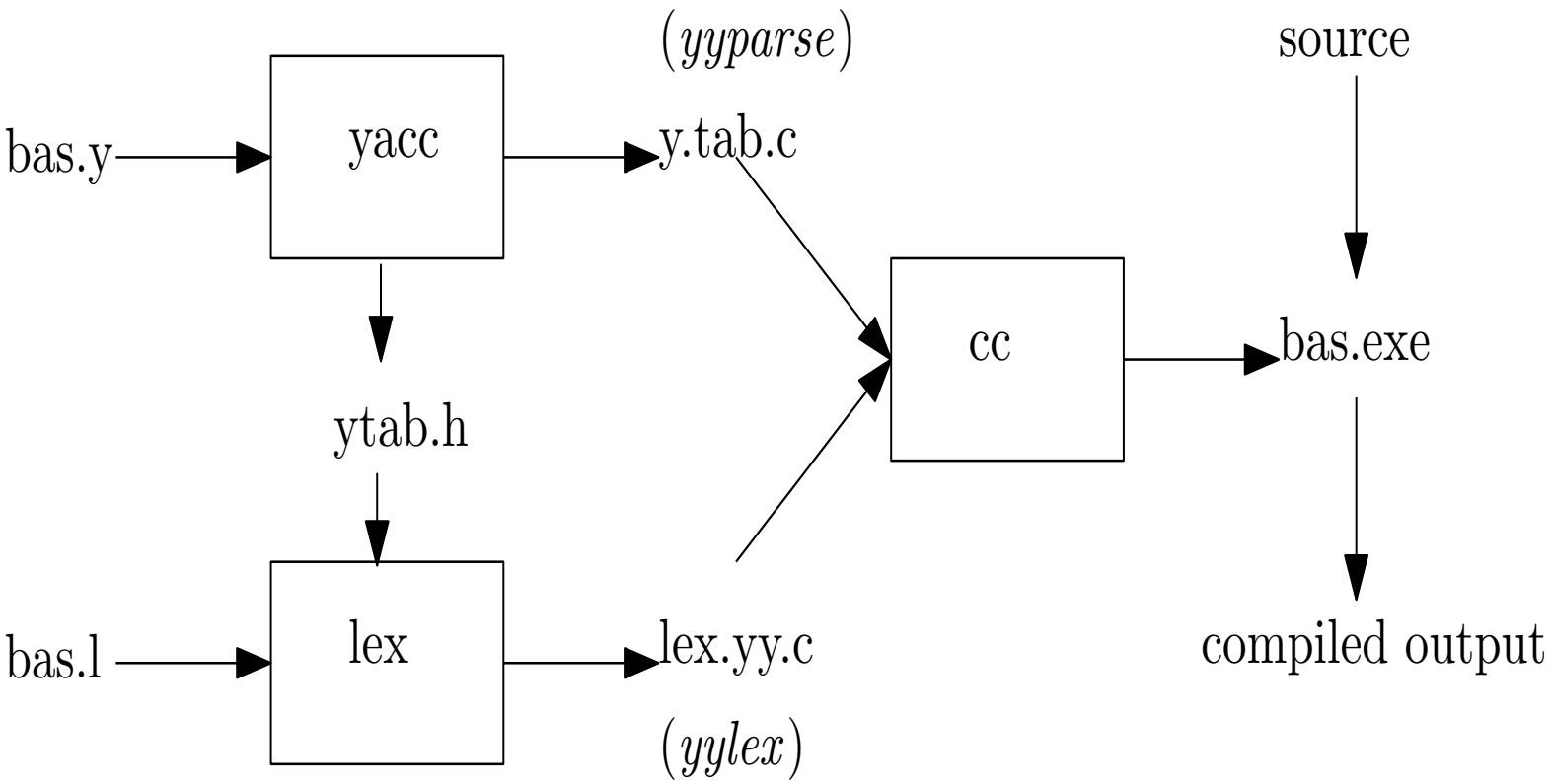
Lexical-Analyzer Generator Lex

- *Lex* tool = *Lex* language + *Lex* compiler
- More recent implementation is Flex
- Alternatives - JFlex uses Java, PyPI uses Python



- lex.l contains the regular expression
- lex.yy.c contains the lexical analyzer code in C language
- lex.yy.c is compiled to produce the lexical analyzer
- yylval - a global variable in lex.yy.c stores token attributes
- Token attribute - a numeric code, a pointer to symbol table or nothing
- yylval is shared between lexical analyer and parser

Building a Complier with Lex and Yacc



Structure of Lex Programs

- Format of a Lex program

```
declarations
%%
translation rules
%%
auxiliary functions
```

- declarations - variables, manifest contants and regular definitions
- translation rules - patterns with actions
- auxiliary functions - additional functions are used in the actions

- Format of each translation rule

```
Pattern    { Action }
```

- Each pattern is a regular expression (may use regular definitions)
- Actions are fragment of code written in C

A Lex program to reconize following tokens

Token	Code	Value
begin	1	–
end	2	–
if	3	–
then	4	–
else	5	–
identifier	6	Pointer to symbol table
constant	7	Pointer to symbol table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

Tokens recognized

A Lex program - token_recognizer.l

```
%{
    #include<stdio.h>
    /* definitions of manifest contants */
    enum yytokentype {LT = 1, LE, EQ, NE, GT, GE,
        BEGIN, END, IF, THEN, ELSE, ID, CONSTANT};
%}
    /* regular definitions */
delim           [ \t\n]
ws              {delim}+
letter          [A-Z,a-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
constant        {digit}+
%%
{ws}            {/* no action and no return */}
begin           {printf("BEGIN\n"); return BEGIN;}
end             {printf("END\n"); return END;}
if              {printf("IF\n"); return IF;}
then            {printf("THEN\n"); return THEN;}
else            {printf("ELSE\n"); return ELSE;}
{id}            {printf("ID\n"); yylval = (int) installID(); return ID;}
{constant}      {printf("CONSTANT\n"); yylval = (int) installConst();
                  return CONSTANT;}
"<"             {printf("LT\n"); yylval = LT; return RELOP;}
"<="            {printf("LE\n");yylval = LE; return RELOP;}
"="             {printf("LT\n"); yylval = EQ; return RELOP;}
"<>"            {printf("NE\n"); yylval = NE; return RELOP;}
">"             {printf("LT\n"); yylval = GT; return RELOP;}
">="            {printf("GE\n"); yylval = GE; return RELOP;}
%%
int installID(){ /* function to install the lexeme, whose
                    first character is pointed to yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer thereto*/
    }
int installConst(){   /* similar to installID, but puts numerical
                         constants into a separate table */
    }
int main(){
        yylex(); /* scanner routine */
        return 0;
    }
```

Running token_recognizer.l

```
$ flex token_recognizer.l
$ gcc lex.yy.c -lfl
$ ./a.out
if total > 50 then
IF
ID
GT
CONSTANT
THEN
begin 22 end
BEGIN
CONSTANT
END
^D
$
```

- lex.yy.c is linked with flex library, -lfl
- Each time the program needs a token, it calls yylex()
- yylex() reads an input, matches pattern and returns token
- yylex() called again for next token

Flex Library -lfl

- default main routine with "while(yylex()!=0);"
- input() - lexer calls input() to fetch each of the matched characters
- unput() - unput(c) returns character c to input stream
- yyinput() and yyunput() - input() and unput() in C++ scanner
- yytext() - matched token is stored in null-terminated string yytext
- yyleng(yytext) - length of yytext
- yyless(n) - "push back" all but the first n characters of the token.
- yylex() and YY_DECL - yylex has no arguments, interacts through global variables. YY_DECL declares calling sequence and adds whatever arguments wanted for yytext
- yymore() - tell lex to append the next token to current one
- yyrestart() - yyrestart(f) makes the scanner read from open stdio file f
- yywrap() - on reaching EOF lexer calls yywrap() to find next job, returns 0 for continuing scanning, returns 1 for EOF

Regular Expressions

- [] - any character within [] except \n
  - First character ^- any character except the ones within []
  - Dash in [] - character range
  - $[0-9]$ means [0123456789], $[a-z]$ means any lowercase letter
  - $[a-z]\{-\}[jv]$ - any character in $\{a, b, \cdots, z\} - \{j, v\}$
- $ - end of a line as the last character of a regular expression
- $\{\}$ - min and max number of times the previous pattern can match
  - $A\{1, 3\}$ matches one to three occurrences of the letter A
  - $0\{5\}$ matches 00000
- \ - escape metacharacters as part of the usual C escape sequences
  - \n is a newline character, * is a literal asterisk.
- $*$ - zero or more copies of the preceding expression
- $+$ - one or more copies of the preceding expression
- ? - zero or one occurrence of the preceding regular expression
  - $-?[0-9]+$ - signed number including an optional leading minus sign

Regular Expressions

- | - preceding or following regular expression
  - faith|hope|charity - any of the three virtues
  - ″ . . . ″ - anything within the quotation marks is treated literally
- () - groups a series of REs together into a new RE
  - (01) matches the character sequence 01
  - $a(bc|de)$ matches *abc* or *ade*
- Few near misses in patterns
  - $[-+]?[0-9.]^+$ - matches too much, like 1.2.3.4
  - $[-+]?[0-9]^+\.?[0-9]^+$ - matches too little, misses .12 or 12.
  - $[-+]?[0-9]^*\.?[0-9]^+$ - doesn't match 12.
  - $[-+]?[0-9]^+\.?[0-9]^*$ - doesn't match .12
  - $[-+]?[0-9]^*\.?[0-9]^*$ - matches nothing, or a dot with no digits at all

How Flex Handles Ambiguous Patterns

- Ambiguous - multiple patterns that can match same input
- Match longest possible string every time scanner matches input
- Tie breaker - use pattern that appears first in the program
- For example - consider the following code snippet
  “+” { return ADD;}
  “=” { return ASSIGN; }
  “+=” { return ASSIGNADD; }
  “if” { return KEYWORDIF; }
  “else” { return KEYWORDELSE; }
  [a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
- += is matched as one token, since += is longer than +
- patterns for keywords precede patterns for identifier

File I/O in Flex Scanners

- A scanner reads from the stdio FILE called yyin
- To read a single file, set it before the first call to yylex
- Programs with simple I/O reads each input file from beginning to end
- In flex yyrestart(f) tells the scanner to read from stdio file f
- For multiple files
  - For each file open file
  - Use yyrestart() to make it the input to the scanner
  - Call yylex() to scan it

Word count program - one input file

- Word count, reading one file

```
%option noyywrap
%{
int chars = 0,words = 0, lines = 0;
%}
%%
[a − zA − Z]+        { words++; chars += strlen(yytext); }
\n                   { chars++; lines++; }
.                    { chars++; }
%%
main(int argc, char **argv){
   if(argc > 1) {
      if(!(yyin = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return (1);
    }
  }
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
}
```

Word count program - many input files

- Word count, reading many files

```
%option noyywrap %{ int chars = 0, words = 0,lines = 0,totchars = 0,totwords = 0, totlines = 0; %} %%
[a − zA − Z]+      { words++; chars += strlen(yytext); }
\n                 { chars++; lines++; }
.                  { chars++; }
%%
main(int argc, char **argv){
  int i;
  if(argc < 2) { /* just read stdin */
    yylex();       printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
  }
  for(i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");
    if(!f) {
    perror(argv[i]);
    return (1);
   }
   yyrestart(f);
   yylex();
   fclose(f);
   printf("%8d%8d%8d %s\n",lines, words, chars, argv[i]);
   totchars += chars; chars = 0;
   totwords += words; words = 0;
   totlines += lines; lines = 0;
  }
  if(argc > 1) /* print total if more than one file */
  printf("%8d%8d%8d total\n", totlines, totwords, totchars);
  return 0;
}
```

The I/O Structure of a Flex Scanner

- Reads from an input source and optionally writes to an output sink
- By default, the input and output are stdin and stdout
- Lex read from yyin one character at a time
- Flex has three-level input allows user to handle any input structure
- Flex reads input using stdio from a file or standard input
- To handle input a flex uses a structure YY_BUFFER_STATE
- YY_BUFFER_STATE describes a single input source
- YY_BUFFER_STATE contains - a string buffer, variables and flags
- YY_BUFFER_STATE for scanning file or string already in memory
- Flex scanner output - by default copies unmatched input to yyout

  . ECHO;

  #define ECHO fwrite(yytext, yyleng, 1, yyout)

The I/O Structure of a Flex Scanner

- Default input behavior of a flex scanner:
  YY_BUFFER_STATE bp;
  extern FILE* yyin;
  $\cdots$ *whatever the program does before the first call to the scanner*
  if(!yyin) yyin = stdin; *default input is stdin*
  bp = yy_create_buffer(yyin,YY_BUF_SIZE);
  YY_BUF_SIZE defined by flex, typically 16K
  yy_switch_to_buffer(bp); *tell it to use the buffer we just made* yylex();
  or yyparse() or whatever calls the scanner

Thank you