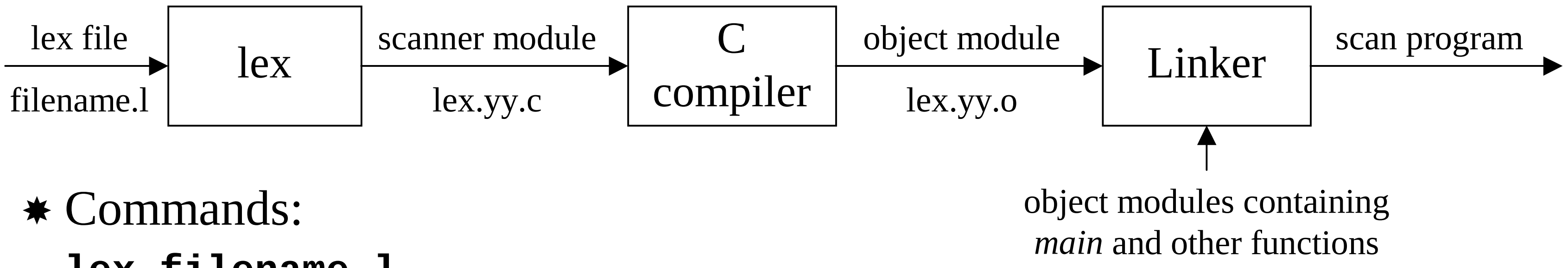
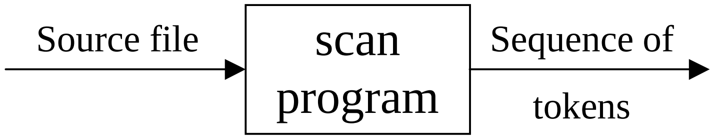
Using the Lex Scanner Generator

- Lex is a popular scanner (lexical analyzer) generator
  - Developed by M.E. Lesk and E. Schmidt of AT&T Bell Labs
  - Other versions of Lex exist, most notably flex (for Fast Lex)
- Input to Lex is called Lex specification or Lex program
  - Lex generates a scanner module in C from a Lex specification file
  - Scanner module can be compiled and linked with other C/C++ modules



- Commands:

```
lex filename.l
cc –c lex.yy.c
cc lex.yy.o other.o –o scan
scan infile outfile
```



A TINY Language

- A TINY program is a sequence of statements terminated by semicolons
- There are no procedures and no declarations
- All variables are of integer type; Variables need not be declared
- There are two control statements:
  - An **if** statement has an optional **else** part and is terminated with **end**
  - A **while** statement is terminated with **end**
    - An arbitrary number of statements can be nested inside an **if** or **while** statement
- There is a **read** and a **write** statement that perform input/output
  - **read** inputs integer variables only; variables are separated by commas
  - **write** outputs integer variables and string literals, separated by commas
  - String literals are enclosed in double quotes
- Comments are enclosed in curly brackets **{** and **}** and cannot be nested
- Expressions are limited to Boolean and integer arithmetic expressions
  - Boolean expressions can be used only as tests in control statements
- Relational operators are: **`<    <=    >    >=    =    <>`**
- Arithmetic expressions involve integer constants, variables, **`()`**, **`+`**, **`-`**, **`*`**, and **`/`**

A Sample TINY Program

- The TINY language lacks many of the essential programming features
  - No functions or procedures
  - No arrays or records
  - No floating-point, string, or character computation
- It is designed to illustrate the different phases of compilation
- The following is a sample TINY program:

```
{Sample program in TINY language – Factorial}
write "Enter an integer value: ";
read x;
factorial := 1;
count := x;
while count > 1 do
  factorial := factorial * count;
  count := count-1;
end;
write "factorial of " , x , " = " , factorial;
```

Lex Specification

- A Lex specification file consists of three sections:

```
definition section
%%
rules section
%%
auxiliary functions
```

- The definition section contains a **literal block** and **regular definitions**
- The literal block is C code delimited by **`%{`** and **`%}`**
  - Contains variable declarations and function prototypes
- A regular definition gives a name to a regular expression
  - A regular definition has the form: **`name   expression`**
  - A regular definition can be used by writing its name in braces: **`{name}`**
- The rules section contains regular expressions and C code; it has the form:

$$\begin{array}{ll} \mathtt{r}_1 & \mathtt{action}_1 \\ \mathtt{r}_2 & \mathtt{action}_2 \\ \quad . \; . \; . & \\ \mathtt{r}_n & \mathtt{action}_n \end{array}$$

- $\mathtt{r}_\mathtt{i}$ is a regular expression and $\mathtt{action}_\mathtt{i}$ is C code fragment
- When $\mathtt{r}_\mathtt{i}$ matches an input string, $\mathtt{action}_\mathtt{i}$ is executed
- $\mathtt{action}_\mathtt{i}$ should be in **`{}`** if more than one statement exists

Lex Operators (Meta-characters)

`\` C escape sequence: `\n` is newline, `\t` is tab, `\\` is backslash, `\"` is double quote, etc.

`*` Matches zero or more of the preceding expression; `x*` matches ε, `x`, `xx`, ...

`+` Matches one or more of the preceding expression; `(ab)+` matches `ab`, `abab`, ...

`?` Matches zero or one occurrence of the preceding expression; `(ab)?` matches ε or `ab`

`|` Matches either the preceding or the subsequent expression; `a|b` matches `a` or `b`

`.` Matches any single character except the newline character

`( )` Used for grouping sub-expressions in a regular expression

`[ ]` Matches any one of the characters within brackets
A range of characters is indicated with the **–** (dash operator)
`[0-9]` matches any decimal digit; `[xyz]` means `(x|y|z)`
If first character after `[` is `^`, it complements the character class
`[^A-Za-z]` matches all characters which are NOT letters
Meta-characters other than C escape sequences loose their meaning inside `[ ]`

`" "` Matches everything within the quotation marks literally; `"x*"` matches only `x*`
Meta-characters other than C escape sequences loose their meaning inside `" "`

`{ }` `{name}` refers to a regular definition from the first section
`[A-Z]{3}` matches strings of exactly 3 capital letters
`[A-Z]{1,3}` matches strings of 1, 2, or 3 capital letters

`/` The lookahead operator; matches the left regular expression but only if followed by the right regular expression
`0/1` matches `0` in `01`, but not in `02` ; Only one slash is permitted per regular expression

`^` As the first character of a regular expression, `^` matches the beginning of a line

`$` As the last character of a regular expression, `$` matches the end of a line; Same as `/\n`

Lex Specification of a TINY Scanner

```
%{                              /* Literal C block */
#include "scan.h"               /* Scanner header file */
int lineno = 1;                 /* Current line number */
TokAttr tokval;                 /* Token attribute value */
void lex_err(char*s1,char*s2);  /* Reports lexical errors */
%}

letter  [A-Za-z]                /* Regular definitions */
digit   [0-9]

%%

\n                              { lineno++; }
[ \t]+                          { /* skip spaces and tabs */ }

[Ii][Ff]                        { return IF; }
[Tt][Hh][Ee][Nn]                { return THEN; }
[Ee][Ll][Ss][Ee]                { return ELSE; }
[Ee][Nn][Dd]                    { return END; }
[Ww][Hh][Ii][Ll][Ee]            { return WHILE; }
[Dd][Oo]                        { return DO; }
[Rr][Ee][Aa][Dd]                { return READ; }
[Ww][Rr][Ii][Tt][Ee]            { return WRITE; }

{letter}({letter}|{digit}|_)*   { tokval.str = yytext; return ID; }
```

More Regular Expressions and Actions

```
_({letter}|{digit}|_)* {lex_err(yytext,"is not a valid identifier");
                        tokval.str = yytext; return ID;}

{digit}+               {tokval.num = atoi(yytext); return INTLIT;}

\"([^\"\n])*\"         {tokval.str = yytext; return STRLIT;}

\"([^\"\n])*$          {lex_err(yytext,"is not terminated");
                        tokval.str = yytext; return STRLIT;}

"+"                    { tokval.op = PLUS;  return ADDOP; }
"-"                    { tokval.op = MINUS; return ADDOP; }
"*"                    { tokval.op = MULT;  return MULOP; }
"/"                    { tokval.op = DIV;   return MULOP; }
"="                    { tokval.op = EQ;    return RELOP; }
"<>"                   { tokval.op = NE;    return RELOP; }
"<"                    { tokval.op = LT;    return RELOP; }
"<="                   { tokval.op = LE;    return RELOP; }
">"                    { tokval.op = GT;    return RELOP; }
">="                   { tokval.op = GE;    return RELOP; }

","                    { return COMMA; }
";"                    { return SEMICOL; }
```

Auxiliary Functions

```
":="    { return ASSIGN; }
"("     { return LP; }
")"     { return RP; }

"{"     { char c; char comment[40];
          sprintf(comment,"Comment starting at line %d",lineno);
          for (c = input(); c != 0 && c != '}'; c = input())
            if (c == '\n') lineno++;
          if (c == 0) lex_err(comment,"is not terminated"); }

.       { lex_err(yytext,"is not recognized"); }

%%

// To report an error message
void lex_err(char *s1, char *s2) {
  fprintf(stderr,"Error at line %d: %s %s\n", lineno, s1, s2);
}

// To finish scanning at end of file
int yywrap() {
  return 1;
}
```

Lex Output File and yylex()

- Lex generates a C file containing the scanner function: **`int yylex()`**
  - Tokens are returned by **`yylex()`** as integer values
- The literal C block is copied verbatim to the output file
  - The literal block appears near the beginning before the **`yylex()`** function
- The third section with auxiliary functions is also copied to the output file
- When called, **`yylex()`** matches input characters against regular expressions
  - If a match occurs, the action associated with the matched expression is executed
  - If the action specifies a **return value** then it will be the value returned by **`yylex()`**
  - Otherwise, scanning continues until an action with a return statement is executed
  - Input characters not matched by any expression are copied to output file
- Regular expressions are allowed to overlap – match same input sequence
  - In case of an overlap, two or more regular expressions apply
  - First, the longest possible match is performed
  - Second, if two expressions match the same string, the first expression listed is used

Internal Names Generated in Lex Output File

- An input sequence matched by a regular expression is stored in:
  - String **`yytext`** whose length is **`yyleng`**
  - String **`yytext`** changes value every time **`yylex()`** is called
- **`yylex()`** calls three user-defined routines to handle character input/output:
  - **`input()`** retrieves a single character, 0 on end of file
  - **`output(c)`** writes a single character *c* to the output
  - **`unput(c)`** puts a single character *c* back to input, to be re-read
- **`input`**, **`output`**, and **`unput`** can be called also in the user-defined actions
- **`input`** reads input characters from **`yyin`** and **`output`** writes to **`yyout`**
  - **`yyin`** is by default **`stdin`**, and **`yyout`** is by default **`stdout`**
  - **`yyin`** can be associated with a input file: **`yyin = fopen(infile,"r");`**
  - **`yyout`** can be associated with a output file: **`yyout = fopen(outfile,"w");`**
- When **`yylex`** encounters end of file, it calls a user-supplied function **`yywrap`**
  - If **`yywrap`** returns 1, **`yylex`** returns the token 0 to report the end of file
  - If **`yywrap`** returns 0, it indicates more input; **`yyin`** must associate with another file

Scanner Header File

- Header file **"scan.h"** has the following definitions and function prototypes:

```
typedef enum  { IF = 300, THEN, ELSE, END, WHILE, DO, READ,
                WRITE, ID, INTLIT, STRLIT, COMMA, SEMICOL,
                ADDOP, MULOP, RELOP, ASSIGN, LP, RP } TokenType;
```

- The **IF** token has value 300, **THEN** will be 301, and so on
  - We could have chosen different values as long as the 0 token (EOF) is not used
  - We can also define the token values as constants or as **#define** macros

```
typedef enum  { PLUS = 1, MINUS, MULT, DIV,
                EQ, NE, LT, LE, GT, GE } OpType;

typedef union {
  OpType op;             // Operator value
  int    num;            // Integer literal value
  char * str;            // Points to yytext for IDs and Strings
} TokAttr;

extern int lineno;      // Current line number
extern TokAttr tokval;  // Attribute value of current token
extern int yylex();     // Scanner function
```

Main Function

- The *main* function can be supplied in a separate file
  - Initializes *yyin* to read from an input file
  - Initializes *yyout* to write to an output file
  - Calls the *yylex* function repeatedly until *yylex* returns 0 (end-of-file token)

```
#include "scan.h"
#include <stdio.h>
#include <string.h>

extern FILE *yyin, *yyout;          // Defined in lex.yy.c

void main(int argc, char *argv[]){
  int  token;

  if (argc < 2) { ... }             // Input/Output filenames NOT given
  if (argc < 3) { ... }             // Output filename is NOT given
  yyin = fopen(argv[1],"r");        // Initialize yyin
  yyout = fopen(argv[2],"w");       // Initialize yyout
  if (yyin == 0) { ... }            // Unable to open input file
  if (yyout == 0) { ... }           // Unable to open output file
  token = yylex();
  while (token) {
    ...                             // Write tokens to output file
    token = yylex();
  }
}
```

Practical Considerations – Reserved Words

- Virtually all programming languages have special words called **keywords**
  - Keywords are normally reserved – cannot be used as identifiers
- We can write a regular expression for each keyword in the lex specification
  - A general expression for identifiers must be placed after the keyword expressions
  - A keyword will be matched by a keyword expression because it is listed first
  - Lex handles exceptions by placing specialized rules before a general one
  - This approach is simple and effective for a small number of reserved words
  - The DFA size is huge when the number of reserved words is large
- An alternative solution is to treat keywords as identifiers
  - One general regular expression is used to match keywords and identifiers
  - A separate lookup table is searched to detect keywords
    - A sorted list of keywords can be used with binary search
    - A better approach is to use a hash table with a perfect hash function (no collisions)
    - A perfect hash function can be designed because keywords are known
  - The DFA size is tremendously reduced

Lexical Error Recovery

- Occasionally, a scanner will detect a lexical error
  - It is unreasonable to stop compilation because of such a minor error
  - Typically, a lexical error is caused by the appearance of an illegal character
  - An illegal character cannot be matched by any regular expression
  - By default, Lex writes unmatched characters to *yyout*
  - The **.** (dot) as a last regular expression can match all illegal characters
- Certain lexical errors can be repaired
  - Special regular expressions can be written to match illegal char sequences
    - Error messages and flags are generated
    - Valid tokens are returned for normal parsing
  - For example, runaway strings can be repaired
    - A runaway string is detected by reaching the end of line
    - An error message is generated, but a correct token can be returned
  - Multi-line comments should be given special attention
    - A runaway comment is not detected until the end of file is reached