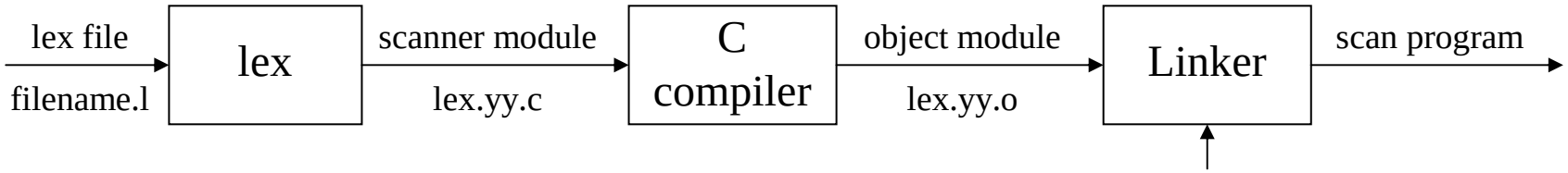
Using the Lex Scanner Generator

- 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
```

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:

r_1 $action_1$
r_2 $action_2$
. . .
r_n $action_n$

  - r_i is a regular expression and $action_i$ is C code fragment
  - When r_i matches an input string, $action_i$ is executed
  - $action_i$ 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`**, **`ababab`**, ...
`?`	Matches zero or one occurrence of the preceding expression: **`(ab)?`** matches ε or **`ab`**
`|`	Matches the preceding or the subsequent expression: **`a|b`** matches **`a`** or **`b`**
`( )`	Used for grouping sub-expressions in a regular expression
`[ ]`	Matches any one of the characters within brackets **`[xyz]`** means **`(x|y|z)`** A range of characters is indicated with the dash operator (–) **`[0-9]`** matches any decimal digit, **`[A-Za-z]`** matches any letter If first character after **`[`** is **`^`**, it complements the character class **`[^A-Za-z]`** matches all characters which are NOT letters Meta-characters other than **`\`** loose their meaning inside **`[ ]`**

Lex Operators – cont'd

`.`	Matches any single character except the newline character
`" "`	Matches everything within the quotation marks literally `"x*"` matches exactly `x*` Meta-characters, other than \ , loose their meaning inside `" "` C escape sequences retain 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 expression but only if followed by the right expression `0/1` matches `0` in `01`, but not in `02` Only one slash is permitted per regular expression
`^`	As first character of a regular expression, `^` matches beginning of a line
`$`	As last character of a regular expression, `$` matches end of a line Same as `/\n`

A TINY Language

- A TINY program is a sequence of statements ended by semicolons
- Comments are enclosed in braces **{** and **}** and cannot be nested
- 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**
- There is a **read** and **write** statements 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 and appear only in write statements
- There is an assignment statement for integer variables only
- Statements can be nested

TINY Language – cont'd

- Expressions are limited to Boolean and arithmetic expressions
- Boolean expressions are used as tests in control statements
- Relational operators are: **< <= > >= = <>**
- Arithmetic operators are: **+**, **-**, *****, and **/**
- Parenthesis **(** and **)** can be used for grouping terms
- There are no procedures and no declarations
- The TINY language lacks many essential programming features
  - No functions or procedures
  - No arrays or records
  - No floating-point, string, or character computation
- It is used only to illustrate the different phases of compilation

A Sample TINY Program

- Calculating the factorial of an input number *x*

```
{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;
```

Tokens of the TINY Language

- The TINY language specifies a number of tokens
- Every keyword has a different token: **`IF_TK`**, **`THEN_TK`**, etc.
- All identifiers are represented by one token: **`ID`**
- Each type of literal constant has a unique token
  - **`INTLIT`** token represents all integer literals
  - **`STRLIT`** token represents all string literals
- Operators of same precedence can have the same token
  - **`ADDOP`** token represents addition and subtraction
  - **`MULOP`** token represents multiplication and division
  - **`RELOP`** token represents all relational operators
- Every delimiter has a unique token: `','` , `';'` , `'('` , and `')'`.

Token Type and Operator Type

- A token type can be defined as an enumeration type

```
enum TokenType { IF_TK=300, THEN_TK,  ELSE_TK,
                 WHILE_TK,  DO_TK,    END_TK,
                 READ_TK,   WRITE_TK, ID,
                 ADDOP,     MULOP,    RELOP,
                 ASGNOP,    INTLIT,   STRLIT  };
```

- **IF_TK** has value 300, **THEN_TK** will be 301, and so on
  - We may choose different token values. However, be careful…
  - The 0 token is reserved by lex as the end-of-file token
  - Characters can be treated as tokens and their codes (1-255) are reserved
- We also need an operator type to distinguish between operators

```
enum OpType    { ADD=1, SUB, MUL, DIV,
                 EQ, NE, LT, LE, GT, GE, ASGN };
```

Lex Specification of a TINY Scanner

```
%{                                      /* Literal block */
#include "scan.h"                       /* Scanner header file */
int line = 1;                           /* Current line number */
int pos = 0;                            /* Start position of token */
int epos = 1;                           /* End position of token */
OpType op;                              /* Last operator scanned */
void lex_err(char*, char*);             /* Reports lexical errors */
%}

/* Regular definitions */
letter          [A-Za-z_]    /* letter can be underscore */
digit           [0-9]
blank_str       [ \t]+
identifier      {letter}({letter}|{digit})*
int_literal     {digit}+
str_literal      \"([^\"\n])*\"

/* Regular definitions to match invalid tokens */
open_string     \"([^\"\n])*$
```

Regular Expressions and Actions

```
%%                          /* Second section */

\n                          {line++; epos = 1;}
{blank_str}                 {epos += yyleng;}

[Ii][Ff]                    {pos = epos; epos += 2; return IF_TK;}
[Tt][Hh][Ee][Nn]            {pos = epos; epos += 4; return THEN_TK;}
[Ee][Ll][Ss][Ee]            {pos = epos; epos += 4; return ELSE_TK;}
[Ee][Nn][Dd]                {pos = epos; epos += 3; return END_TK;}
[Ww][Hh][Ii][Ll][Ee]        {pos = epos; epos += 5; return WHILE_TK;}
[Dd][Oo]                    {pos = epos; epos += 2; return DO_TK;}
[Rr][Ee][Aa][Dd]            {pos = epos; epos += 4; return READ_TK;}
[Ww][Rr][Ii][Tt][Ee]        {pos = epos; epos += 5; return WRITE_TK;}

{identifier}                {pos = epos; epos += yyleng;
                             return ID;}
{str_literal}               {pos = epos; epos += yyleng;
                             return STRLIT;}
{int_literal}               {pos = epos; epos += yyleng;
                             return INTLIT;}
```

More Regular Expressions and Actions

```
"+"    {op = ADD;  pos = epos; epos += 1; return ADDOP;}
"-"    {op = SUB;  pos = epos; epos += 1; return ADDOP;}
"*"    {op = MUL;  pos = epos; epos += 1; return MULOP;}
"/"    {op = DIV;  pos = epos; epos += 1; return MULOP;}
"="    {op = EQ;   pos = epos; epos += 1; return RELOP;}
"<>"   {op = NE;   pos = epos; epos += 2; return RELOP;}
"<"    {op = LT;   pos = epos; epos += 1; return RELOP;}
"<="   {op = LE;   pos = epos; epos += 2; return RELOP;}
">"    {op = GT;   pos = epos; epos += 1; return RELOP;}
">="   {op = GE;   pos = epos; epos += 2; return RELOP;}
":="   {op = ASGN; pos = epos; epos += 2; return ASGNOP;}

","    {pos = epos; epos += 1; return ',' ;}
";"    {pos = epos; epos += 1; return ';' ;}

"("    {pos = epos; epos += 1; return '(' ;}
")"    {pos = epos; epos += 1; return ')' ;}
```

Lex Output File and *yylex*

- By default, lex generates a C file: *lex.yy.c*
- We will rename the output file as: *scan.cpp*
  - Output file will be compiled using a C++ compiler, instead of C
  - We would like to link the scanner module to other C++ files
  - In Flex, the **`-oscan.cpp`** option specifies the output file name
- The output file contains the scanner function: **`int yylex()`**
  - Tokens returned by **`yylex()`** are integer values
- The literal block is copied verbatim to the output file
  - The literal block appears before the **`yylex()`** function
  - The **return** statement in an action specifies the result of **`yylex()`**
- The third section is also copied verbatim to the output file
  - Auxiliary functions may be placed in the third section

Matching Input Characters

- When called, **`yylex()`** matches chars against regular expressions
  - If a match occurs, the associated action is executed
  - If the action specifies a **return value**, it will be the return value of **`yylex()`**
  - Otherwise, scanning continues until a return statement is executed
  - Input characters not matched by any expression are copied to output file
- Regular expressions are allowed to overlap
  - More than one regular expression may 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 same string, the one listed first is used
- The matched text is saved in **yytext**, and its length in **yyleng**
  - **`yytext`** and **`yyleng`** are modified on every call to **`yylex()`**

Internal Names in the Lex Output File

- **`yylex`** reads input characters from **`yyin`**
  - **`yyin`** is by default **`stdin`**
  - Can associate with an input file: **`yyin = fopen(infile,"r");`**
- **`yylex`** writes output characters to **`yyout`**
  - **`yyout`** is by default **`stdout`**
  - Can associate with an output file: **`yyout = fopen(outfile,"w");`**
- **`yylex`** calls **`yyinput`** to read next input character
  - **`yyinput`** retrieves a single character, or EOF on end of file
  - **`yyinput`** can be called directly in actions
- When **`yylex`** encounters end of file, it calls **`yywrap`**
  - If **`yywrap`** returns 1, **`yylex`** returns the token 0 to report end of file
  - If **`yywrap`** returns 0, it indicates more input
    - **`yyin`** must associate with another file

Handling Multi-Line Comments

```
"{"     { int c;
          int start_line = line;   /* comment start line */
          pos  = epos;             /* comment start position */
          epos++;                  /* for "{" */
          c = yyinput()
          while (c != EOF) {
            epos++;
            if (c == '}') break;
            if (c == '\n') {line++; epos=1;}
            c = yyinput();
          }
          if (c == EOF) {          /* reached EOF */
            int eof_line = line;
            line = start_line;
            lex_err("Open Comment");
            line = eof_line;
          }
        }
```

Start Conditions

- Typically, rules are written with NO start condition

  expression {corresponding action}

- A rule with NAME as a start condition is written as follows

  <NAME>expression {corresponding action}

  Applied only when start condition NAME is activated

- Start conditions are declared in the first section

  %s NAME declares **inclusive** start conditions

  %x NAME declares **exclusive** start conditions

  Introduce multiple start states to the underlying DFA

- The default initial start condition is called INITIAL

- Rules with NO start condition have INITIAL as start condition

Examples on Start Conditions

```
%s A          /* Declaring inclusive start state A */
%x B          /* Declaring exclusive start state B */
%%
expr1         /* active if start condition is INITIAL or A, but not B */
<A>expr2      /* active if start condition is A only */
<B>expr3      /* active if start condition is B only */
<A,B>expr4    /* active if start condition is A or B */
<*>expr5      /* always active regardless of start condition */
```

- A start condition is activated using the BEGIN(NAME) action
  - Rules with the given start condition will be active
  - Rules with other start conditions will be inactive
  - Rules with no start condition will be active if and only if activated start condition is **inclusive**

Handling Multi-line Comments – Revisited

- With start conditions, we can easily handle multi-line comments
  - While keeping track of line number and character position
  - We need an exclusive start condition COMMENT
  - Rules with no start condition will Not interfere with <COMMENT>

```
%x COMMENT              /* Exclusive start state */
%%
"/*"                    {pos=epos; epos+=2; BEGIN(COMMENT);}
<COMMENT>[^*\n]+        {epos+=yyleng;}
<COMMENT>"*"+[^*/\n]+   {epos+=yyleng;}
<COMMENT>"*"*\n         {line++; epos=1;}
<COMMENT>"*"+"/"        {epos+= yyleng; BEGIN(INITIAL);}
<COMMENT><<EOF>>        {lex_err("Unexpected EOF");
                         return 0;}
```

Lexical Error Recovery

- Occasionally, a scanner will detect a lexical error
  - Typically caused by an illegal character that cannot be matched
  - It is unreasonable to stop compilation because of such an error
  - 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 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 a line
    - An error message is generated, but a correct token is returned
  - Multi-line comments should be given special attention
    - A runaway comment is not detected until the end of file is reached

Handling Illegal Tokens and Characters

- An open string literal is handled by …
  - Writing a special regular expression to match it

```
{open_string}     { pos = epos;
                    epos += yyleng;
                    lex_err("Open String", yytext);
                    return STRLIT;
                  }

.                 { pos = epos;
                    epos += 1;
                    lex_err("Unknown Character", yytext);
                  }
```

- Other erroneous tokens can be handled in a similar way
  - Open character literals
  - Floating-point literals with no digit before or after decimal point

Auxiliary Functions

- Auxiliary functions are required for …
  - Reporting error messages
  - Initializing scanner to read from a source file and to write to an output file
  - Further processing of tokens
- Auxiliary functions are written in the third section

```
%%
// Third Section
// Report a lexical error and count it
void lex_err(char* s1, char* s2){
  fprintf(stderr, "l%d, c%d: %s %s\n", line, pos, s1, s2);
  totalErrors++;
}
// yywrap is required by lex
// return 1 means no wrap to another source file
int yywrap() { return 1; }
```

Initializing yyin and yyout

- The function *lex_init* initializes *yyin* and *yyout*
  - *yyin* is associated with *srcfile*, and *yyout* with *outfile*
  - *lex_init* return **true** if files were open successfully and **false** otherwise

```
bool lex_init(char* srcfile, char* outfile) {
  yyin  = fopen(srcfile,"r");  // Initialize yyin
  yyout = fopen(outfile,"w");  // Initialize yyout
  bool done = true;            // Until proven otherwise
  if (yyin == 0) {
    fprintf(stderr,"Can't open %s\n", srcfile);
    done = false;
  }
  if (yyout == 0) {
    fprintf(stderr,"Can't open %s\n", outfile);
    done = false;
  }
  return done;
}
```

Main Function for Testing *yylex*

- The *main* function can be supplied in a separate file
  - Calls *lex_init* to initialize *yyin* and *yyout*
  - Calls the *yylex* function repeatedly until *yylex* returns 0 (end-of-file token)

```
#include "scan.h"
#include . . .

int main(int argc, char* argv[]) {
  if (argc == 1) { ... }              // Filenames NOT given
  else if (argc == 2) { ... }         // Outfilename NOT given

  if (!lex_init(argv[1],argv[2]))     // Cannot open files
    return 1;

  while (int token = yylex()) {       // Call yylex repeatedly
    // Display token information
  }

  return 0;
}
```

Reserved Words

- All programming languages have special words called **keywords**
  - Keywords are normally reserved – cannot be used as identifiers
- One way to handle keywords is to …
  - Write a regular expression for each keyword in the lex specification
  - A regular expression for identifiers is placed after the keyword expressions
  - A keyword is matched by a keyword expression because it is listed first
  - This approach is simple and effective for a small number of reserved words
  - However, the DFA size is huge when number of reserved words is large
- In general, Lex handles exceptions by …
  - Placing specialized regular expressions before a general one

Lookup Table for Reserved Words

- An alternative solution is to treat keywords as identifiers
- One regular expression is used for keywords and identifiers
  - The DFA size is reduced tremendously
- A separate lookup table is searched to detect keywords
- Various implementations can be used
  - A sorted list of keywords with binary search
  - A hash table with a perfect hash function
- A perfect hash function has no collisions
  - Can be designed because keywords are known
  - Keywords are hashed initially before inserting them into the lookup table
  - Exactly one entry in the lookup table is searched
  - A table entry stores a keyword and its associated token number
  - Lookup has a constant complexity: $O(1)$

Tables for Literals and Identifier Names

- The scanner needs two tables …
  - A literal table, **`LitTable`**, for storing all literal constants
  - A name table, **`NameTable`**, for storing all identifier names
- The **`install`** method is used to …
  - Enter new literals and names conditionally, if they are not found
  - Return a pointer to the literal or name symbol

```
{identifier}   {pos = epos; epos += yyleng;
                sym = NameTable.install(yytext,yyleng);
                return ID;}
{str_literal}  {pos = epos; epos += yyleng;
                sym = LitTable.install(yytext,yyleng);
                return STRLIT;}
{int_literal}  {pos = epos; epos += yyleng;
                int num = atoi(yytext);
                sym = LitTable.install(num);
                return INTLIT;}
```

Token Attributes and Token Structure

- We need to preserve additional **attributes** for each token
- Token attributes include
  - Line number and character position of each token
  - Pointer to a name symbol for an identifier
  - Pointer to a literal symbol for a literal constant
  - Exact operator for each operator token
- A token structure holds a token number and its attributes

```
struct TokenStruct {  // Token Structure
  TokenType tok;      // Token number
  int line, pos;      // Line and character position
  OpType  op;         // Operator type
  Symbol* sym;        // Pointer to name or literal symbol
};
```

A Scan Function Based on *yylex*

- A *scan* function is designed based on *yylex*
  - Returns a token structure holding a token number and its attributes
  - The operator attribute, **op**, is used when token is an operator
  - The symbol attribute, **sym**, is used when token is an identifier or literal

```
TokenStruct scan() {
  TokenStruct token;
  token.tok  = yylex();   // yylex for next token
  token.line = line;      // Current line number
  token.pos  = pos;       // Character position
  token.op   = op;        // Token operator
  token.sym  = sym;       // Name or Literal symbol
  return token;
}
```