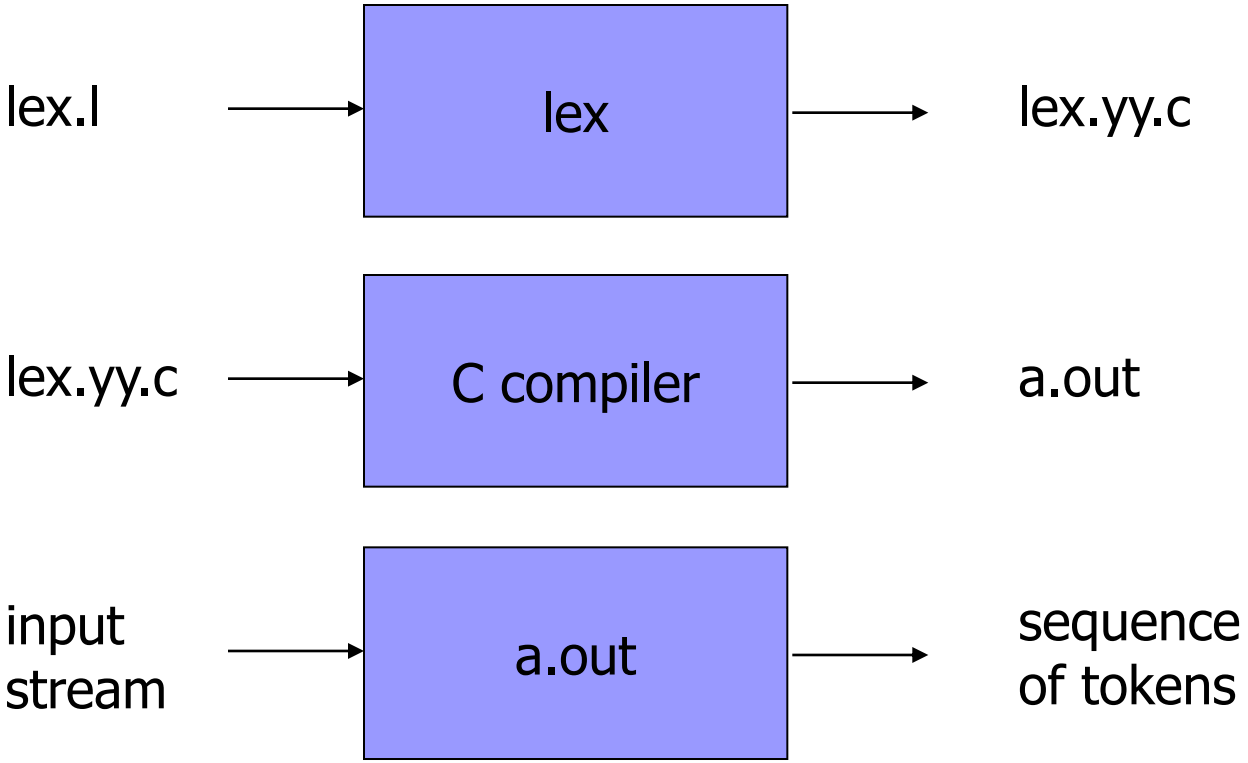
Lex Compiler

- Creating lexical analyzer with lex
- Sensitivity
- Operators

Creating a Lexical Analyzer with lex



lex foo.l; cc lex.yy.c –ll; a.out < test.c;

lex specification

- A lex program consists of three parts:

```
declarations
%%
translation rules
%%
auxiliary procedures
```

Example:

```
%%
[ □t]+$  ;          // delete all blanks and tabs at the
%%                  // end of lines from the input
```

Three Parts

- Declarations
  - variables, constants, regular definitions
- Translation rules:
  - Sequence of p_i {$action_i$}, where p_i is a regular expression and $action_i$ is a C program fragment describing the action the lexer takes when the token p_i is found
- Auxiliary procedures
  - Functions needed by the actions

Cooperation bet'n Lexer & Parser

- When activated by the parser, the lexer matches the longest lexeme and perform an action
- Typically, the action gives control back to parser via `return(Token_Type)`
- Otherwise, lexer finds more lexemes until an action returns
- Lexer returns token to the parser and can also pass an attribute via a global variable `yylval`
- Two reserved variables `yytext` (pointer to the first char of lexeme) and `yyleng` (length of the string)

An Example

```
%{
    /* whatever is included here will be included in lex.yy.c */
    #include subc.h
    #include y.tab.h
int commentdepth = 0;
%}
Letter              [a-z-A-Z]
Digit               [0-9]
Id                  {Letter}({Letter}|{DIgit})*
%%
{Id}                {yylval = install_id(); return(ID);
%%
install_id()   {   /* function to include the id in the symbol table */   }
```

Lookahead Operator: Right Sensitivity

- Remember the need of “lookahead” in some programming languages
  - In lex, an expression form r1/r2, where r1 and r2 are regular expressions, means that r1 matches only if followed by a string in r2
  - Example: Fortran DO loop (e.g., DO I = 1, 5)

```
DO/({letter} | {digit})* = ({letter} | {digit})*,
```

If matched: *yytext = “DO” and yyleng = 2

Start Conditions: Left Sensitivity

- Different lexical rules for different cases in input
  - lex provides start conditions on rules
  - E.g., copy input to output, changing the word “magic” to “first” then changing it to “second” alternately

```
%start AA BB
%%
<AA> magic      {printf(“first”); BEGIN BB}
<BB> magic      {printf(“second”); BEGIN AA}
%%
main() { BEGIN AA; yylex(); }
```

Operator Characters in lex

- “ “ : take as text characters (ex: xyz”++”)
- ₩ : make operators as texts (ex: xyz₩+₩+)
- ^ : complemented character set (ex: [^abc])
  - [^a-zA-Z]: matches any character that is not a letter
- . : arbitrary character (ex: . printf(“bad input”);)
- Context sensitivity : ^ and $
  - If the first character of an expression is ^ , it is matched only at the beginning of a line
  - If the last character is $, the expression is matched only at the end of line (ex: ab$ = ab/₩n)