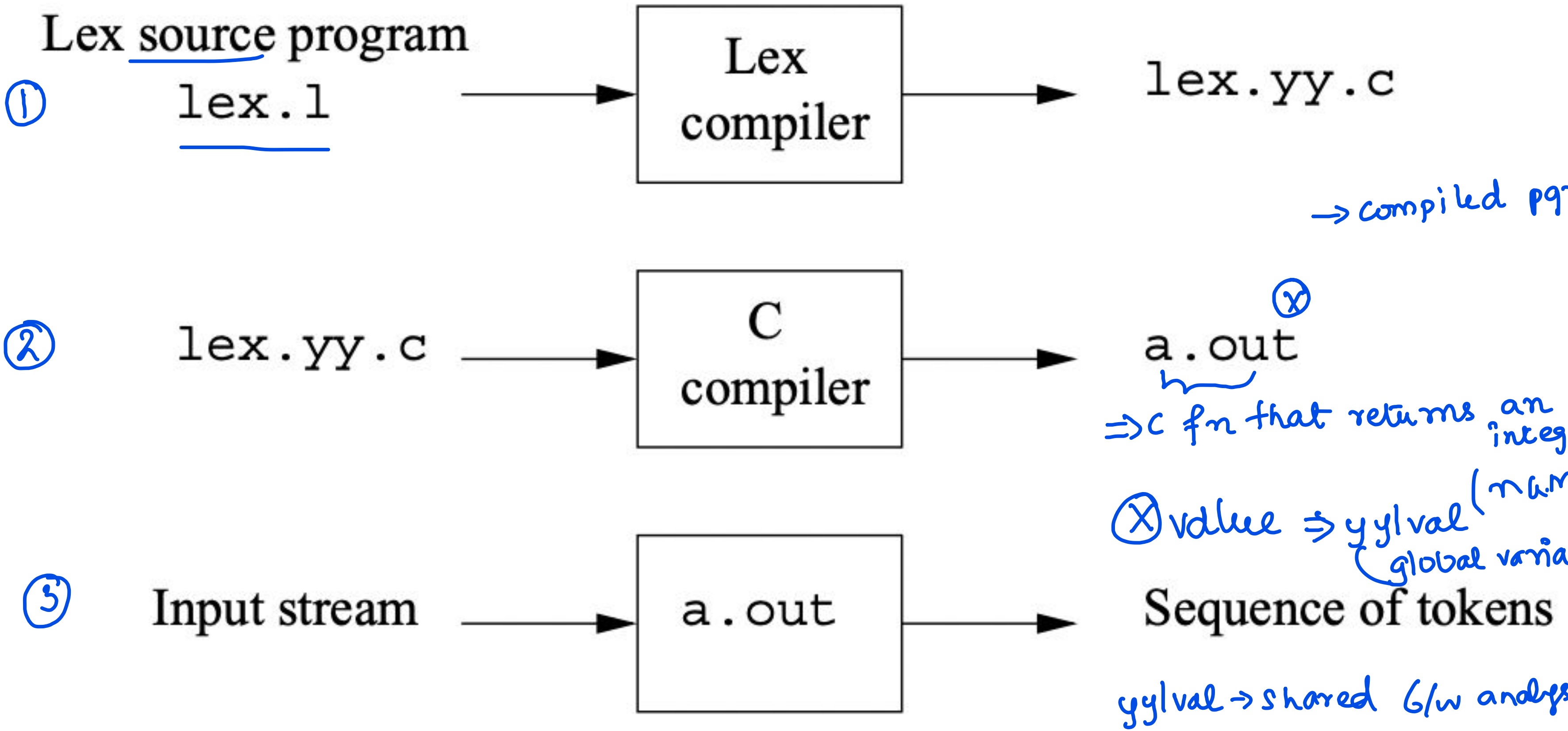
LEX TOOL

allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens



→ compiled pgm

a.out ⇒ c fn that returns an integer

Ⓧ value ⇒ yylval (name) (global variable)

yylval → shared b/w analyser & parser

Figure 3.22: Creating a lexical analyzer with Lex

Working

- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.

1. An input file, which we call lex.l, is written in the Lex language and describes the lexical analyzer to be generated.
2. The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c.
3. The latter file is compiled by the C compiler into a file called a.out, as always.
4. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

Structure of Lex Program

declarations → variables, constants

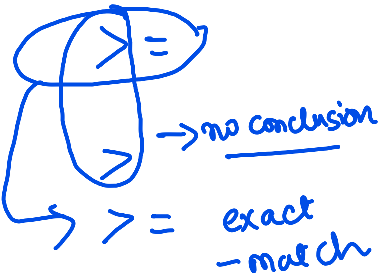
%%

translation rules → pattern + action

%%

auxiliary functions

↳ extra



The translation rules each have the form

Pattern { Action }

1) Reads i/p

2) Matches w.r.t all patterns.

3) Selects the longest prefix of i/p that matches

Example

r "a.* b"

↳ anything

*copied to lex.yy.c

⊗ not treated as regular defn

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

\t - tab

\n - newline

Variable →

↳ backslash

Lex pgm

name

value

yylval

```
%%

{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP);}
"<="        {yylval = LE; return(RELOP);}
"="         {yylval = EQ; return(RELOP);}
"<>"        {yylval = NE; return(RELOP);}
">"         {yylval = GT; return(RELOP);}
">="        {yylval = GE; return(RELOP);}

%%
```

keyword

type conversion

① inserts symbol into symbol table & returns position

name

name

```
%%

int installID() {/* function to install the lexeme, whose
                   first character is pointed to by yytext,
                   and whose length is yyleng, into the
                   symbol table and return a pointer
                   thereto */
}

int installNum() {/* similar to installID, but puts numer-
                    ical constants into a separate table */
}
```

→lexeme Begin

. The action taken when *id* is matched is threefold:

1. Function `installID()` is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:

   (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.

   (b) `yyleng` is the length of the lexeme found.

3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`. □

3.5.3 Conflict Resolution in Lex

>=

pat
>
>=

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

X 1. Always prefer a longer prefix to a shorter prefix.

2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

>=

pat
1 >= ✓
>=- ✓

Lookahead Operator

Count $\underset{\uparrow}{x}$

venkat@gmail.com

Lex automatically reads one character ahead of the last character that forms the selected lexeme, but consumes only the part that matches.

Match based on subsequent characters too using /:

- Slash in a pattern indicates the end of the part of the pattern that matches the lexeme.
- What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

Example 3.13: In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where IF is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where IF is a keyword. Fortunately, we can be sure that the keyword IF is always followed by a left parenthesis, some text — the condition — that may contain parentheses, a right parenthesis and a letter. Thus, we could write a Lex rule for the keyword IF like:

subsequent

```
IF / \( .* \) {letter}
```

👍 SUBSCRIBE 🔔

THANK YOU