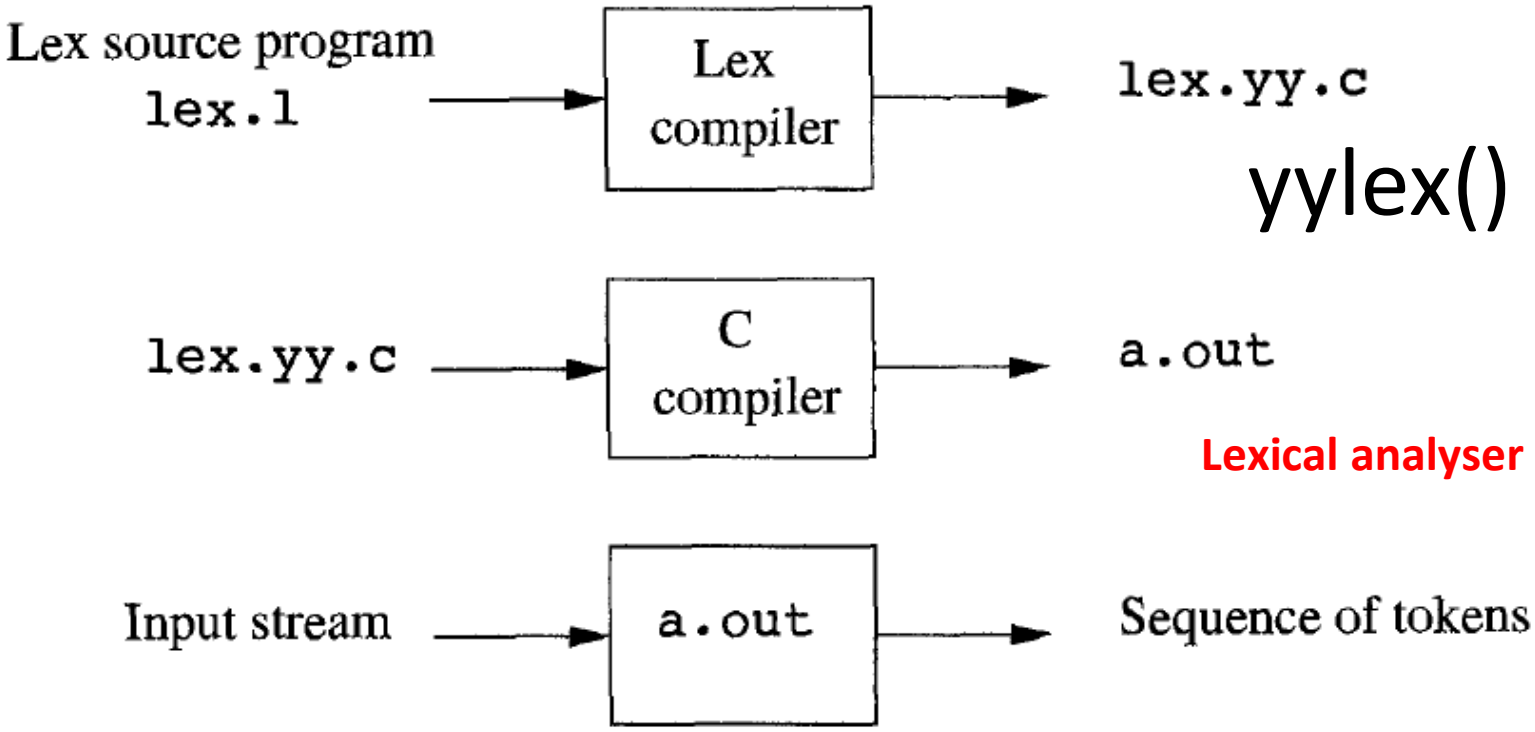
Lex or flex

Lex or flex

- Allows one to **specify** and **construct a lexical analyzer** by
    - Specifying **regular expressions** to describe **patterns for tokens**.
- The **input notation** for the Lex tool is referred to as the **Lex language**
    - **Specify the patterns**

- Lex compiler **transforms** the **input patterns** into a **transition diagram**

    - Generates **code**, in a file called **lex.yy.c**, that simulates this transition diagram.


Lex source program
lex.l
Lex compiler
lex.yy.c
yylex()
lex.yy.c
C compiler
a.out
Lexical analyser
Input stream
a.out
Sequence of tokens

Structure of Lex Programs

A Lex program has the following form:

(a) auxiliary declarations

(b) regular definitions

declarations
%%
translation rules
%%
auxiliary functions

yylex()

Pattern { Action }

Additional functions --- say main()
etc

Structure of Lex Programs

A Lex program has the following form:

declarations ⟸ **Optional**
%%
translation rules
%%
auxiliary functions

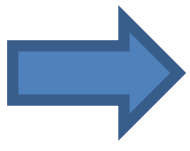
(a) auxiliary declarations
(i)declaration of **variable, functions**
(ii) inclusion of **header file**,
(iii)Defining **macro**

- Enclosed within %{ and %}
- Auxiliary declarations are copied as such by LEX to the output lex.yy.c file.
    - Not processed by the LEX tool.

(b) regular definitions

Structure of Lex Programs

A Lex program has the following form:



declarations
%%
translation rules
%%
auxiliary functions

Lex.yy.c
yylex()

Pattern { Action }

Additional functions --- say main()
etc

(a) Each pattern is a **regular expression**, which may use the **regular definitions** of the **declaration section**.
(b) The **actions** are **fragments of C code**

- yylex() function **checks the input stream** for the **first match to one of the patterns**
- **Executes code** in the **action part** corresponding to the pattern.

Input file

```
if  + 78 else  0
```

Tokens: if, else, op (+,-), number, other

Lex – example

```
%{
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

%}
```

Auxiliary declarations

```
/* Declarations*/
%%
if      return (IF);
else    return (ELSE);
[0-9]+  return(NUM);
[+-]    return(OP);
.       return(ERR);
```

Regular expressions

Actions

Any char %%

Generates yylex()

Structure of Lex Programs

A Lex program has the following form:

(a) auxiliary declarations

(b) regular definitions

declarations
%%
translation rules
%%
auxiliary functions

yylex()

Pattern { Action }

Additional functions --- say main()
etc

- **LEX generates C code** for the rules specified in the **Rules section** and places this code into a single function called **yylex()**.
- **In addition** to this LEX generated code, the programmer may wish to add **his own code** to the lex.yy.c file.
- The **auxiliary functions section** allows the programmer to achieve this.

Lex – example

```
%{
#include<stdio.h>
%}



/* Declarations*/
%%
if      printf("if\n");
else    printf("else\n");
[0-9]+  printf("number %s\n",yytext);
[+-]    printf("operator %s\n",yytext);
.       printf("other\n");
```

Any char

```
%%
/* Auxiliary functions */


int main()
{
    yylex();

        return 1;
}

int yywrap(void)
{
        return(1);
}
```

Auxiliary functions

yylex()

```
%{
#include<stdio.h>
%}



/* Declarations*/
%%
if      printf("if\n");
else    printf("else\n");
[0-9]+  printf("number %s\n",yytext);
[+-]    printf("operator %s\n",yytext);
.       printf("other\n");


%%
/* Auxiliary functions */


int main()
{
    yylex();

        return 1;
}

int yywrap(void)
{
        return(1);
}
```

yylex()

- When **yylex() is invoked**, it **reads the input file** and scans through the input looking for a **matching pattern**.
- When the **input or a part of the input matches** one of the given **patterns**, yylex() **executes the corresponding action** associated with the pattern as specified in the Rules section.
- yylex() **continues scanning** the input
  (a) till one of the actions corresponding to a matched pattern **executes a return statement** or
  (b) till the **end of input** has been encountered.
- Note that **if none of the actions** in the Rules section executes a return statement, yylex() **continues scanning for more matching patterns** in the input file **till the end of the file**.

Lex – example-1

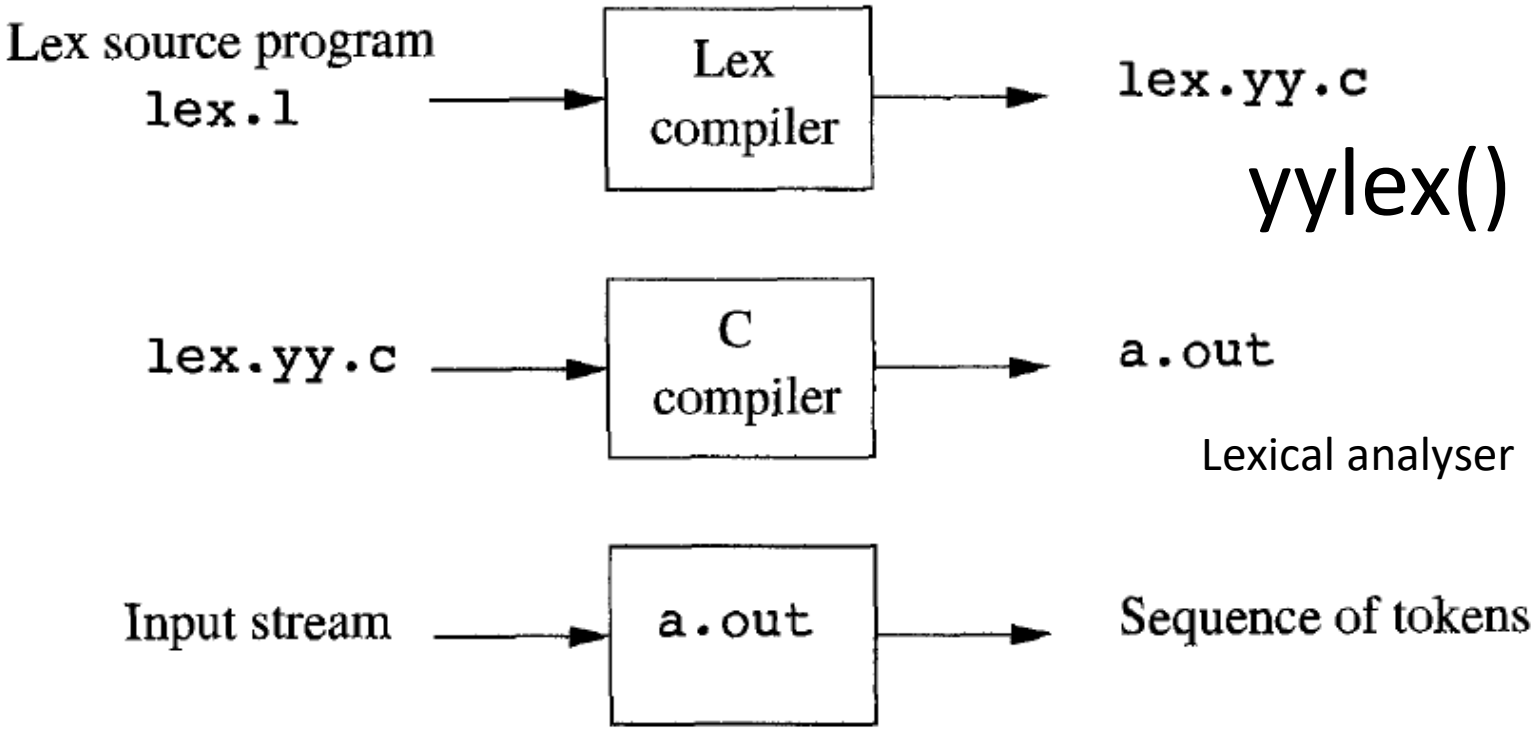
```
%{
#include<stdio.h>
%}


/* Declarations*/
%%
if      printf("if\n");
else    printf("else\n");
[0-9]+  printf("number %s\n",yytext);
[+-]    printf("operator %s\n",yytext);
.       printf("other\n");


%%
/* Auxiliary functions */


int main()
{
    yylex();

        return 1;
}

int yywrap(void)
{
        return(1);
}
```

yytext is the string (of type *char*) indicating the* ***lexeme*** currently found. [like **LexemeBegin**]

Each invocation of the function ***yylex()*** results in ***yytext*** carrying a pointer to the **lexeme** found in the input stream

yyleng is a variable of the type **int** and
it stores the length of the lexeme pointed to by *yytext*.

lex_first.l


Lex source program
lex.l
Lex compiler
lex.yy.c
yylex()
lex.yy.c
C compiler
a.out
Lexical analyser
Input stream
a.out
Sequence of tokens

bivasm@cpusrv-gpu-108: ~/lex

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_first.l
bivasm@cpusrv-gpu-108:~/lex$ gcc lex.yy.c
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_first
if
other
other
operator +
other
number 78
other
else
other
other
number 0

bivasm@cpusrv-gpu-108:~/lex$
```

Lex – example-1

Input file – **input_first**

```
if  + 78 else  0
```

Tokens: if, else, op (+,-), number, other

Lex – example-2

lex_first_v1.l

```
%{#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5%}
/* Declarations*/

%%
if                  return (IF);
Else                return (ELSE);
[0-9]+              return(NUM);
[+-]                return(OP);
.                   return(ERR);
%%
```

```
%%
/* Auxiliary functions */


int main()
{
   int ntoken;
   do{
        ntoken=yylex();

        if(ntoken==0)
                break;

        if(ntoken==IF)
            printf("The IF token is %s\n",yytext);

        else if(ntoken==ELSE)
            printf("The ELSE token is %s\n",yytext);

        else if(ntoken==NUM)
            printf("The NUM token is %s\n",yytext);
        else if(ntoken==OP)
            printf("The OP token is %s\n",yytext);
        else
            printf("The ERR token is %s\n",yytext);
        }
  while(1);
return 1;
}

int yywrap(void)
{
        return(1);
}
```

Input file – **input_first**

if + 78 else 0

bivasm@cpusrv-gpu-108: ~/lex

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_first_v1.l
bivasm@cpusrv-gpu-108:~/lex$ ls
a.out  head.h  input_f  input_first  le  lex_check.l
bivasm@cpusrv-gpu-108:~/lex$ gcc lex.yy.c
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_first
The IF token is if
The ERR token is
The ERR token is
The OP token is +
The ERR token is
The NUM token is 78
The ERR token is
The ELSE token is else
The ERR token is
The ERR token is
The NUM token is 0

bivasm@cpusrv-gpu-108:~/lex$
```

Lex – example-3

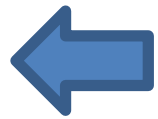
lex_first_v2 - Notepad

File Edit Format View Help

```
%{
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

%}



/* Declarations*/
%%
if      return (IF);
else    return (ELSE);
[0-9]+  return(NUM);
[+-]    return(OP);
.       printf("The error token is %s\n",yytext);


%%
```

lex_first_v2.l

```
int main()
{
   int ntoken;
   do{
        ntoken=yylex();

        if(ntoken==0)
                break;

        if(ntoken==IF)
            printf("The IF token is %s\n",yytext);

        else if(ntoken==ELSE)
            printf("The ELSE token is %s\n",yytext);

        else if(ntoken==NUM)
            printf("The NUM token is %s\n",yytext);
        else if(ntoken==OP)
            printf("The OP token is %s\n",yytext);
        else
                printf("I don't know\n");
        }
  while(1);
return 1;
}

int yywrap(void)
{
        return(1);
}
```



bivasm@cpusrv-gpu-108: ~/lex

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_first_v2.l
bivasm@cpusrv-gpu-108:~/lex$ gcc lex.yy.c
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_first
The IF token is if
The error token is
The error token is
The OP token is +
The error token is
The NUM token is 78
The error token is
The ELSE token is else
The error token is
The error token is
The NUM token is 0

bivasm@cpusrv-gpu-108:~/lex$
```

Lex – example-4

File Edit Format View Help

```
%{
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

%}
```

lex_first_v3.l

```
/* Declarations*/
%%
if      return (IF);
else    return (ELSE);
[0-9]+  return(NUM);
[+-]    return(OP);
.       printf("The error token is %s\n",yytext);


%%
/* Auxiliary functions */


int yywrap(void)
{
        return(1);
}
```

Note: main() is missing

File Edit Format View Help

```
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

extern int yylex();
extern char* yytext;

int main()
{
   int ntoken;
   do{
        ntoken=yylex();

        if(ntoken==0)
                break;

        if(ntoken==IF)
            printf("The IF token is %s\n",yytext);

        else if(ntoken==ELSE)
            printf("The ELSE token is %s\n",yytext);

        else if(ntoken==NUM)
            printf("The NUM token is %s\n",yytext);
        else if(ntoken==OP)
            printf("The OP token is %s\n",yytext);
        else
                printf("I don't know\n");
        }
  while(1);
return 1;
}
```

scanner_v1.c

bivasm@cpusrv-gpu-108: ~/lex

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_first_v3.l
bivasm@cpusrv-gpu-108:~/lex$ gcc lex.yy.c scanner_v1.c
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_first
The IF token is if
The error token is
The error token is
The OP token is +
The error token is
The NUM token is 78
The error token is
The ELSE token is else
The error token is
The error token is
The NUM token is 0

bivasm@cpusrv-gpu-108:~/lex$
```

Input file – **input_first**

if + 78 else 0

Lex – example-5

Input file – **input_f**

```
db_type: mysql
db_name: testdata
db_table_prefix: test_
db_port: 1091
```

Lex – example-5

lex_check.l

```
%{
#include "head.h"
%}

%%

:       return COLON;
"db_type"       return TYPE;
"db_name"       return NAME;
"db_table_prefix"       return TABLE_PREFIX ;
"db_port"       return PORT;
[a-zA-Z][_a-zA-Z0-9]*   return IDENTIFIER;

[0-9][0-9]*     return INTEGER;
[ \t\n]         ;
.       printf("unexpeced\n");

%%
int yywrap(void)
{
        return 1;
}
```

Recognition of Tokens

Regular Definitions for terminals

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

stripping out whitespace, by recognizing the "token" ***ws***

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

ASCII chars

- Token **ws** is different from the other tokens in that,
  - Once we recognize it, we **do not return it to the parser**,
- Rather **restart the lexical analysis** from the character that follows the whitespace. It is the following token that gets returned to the parser.

```
#define TYPE 1
#define NAME 2
#define TABLE_PREFIX 3
#define PORT 4
#define COLON 5
#define IDENTIFIER 6
#define INTEGER 7
```

head.h

```
#include <stdio.h>
#include "head.h"
extern int yylex();
extern int yylineno;
extern char* yytext;
char *names[] = {NULL, "db_type", "db_name", "db_table_prefix", "db_port"};
int main(void)
{
        int ntoken, vtoken;
        ntoken = yylex();
        while(ntoken) {
                printf("%d\n", ntoken);
                if(yylex() != COLON) {
                        printf("Syntax error in line %d, Expected a ':' but found %s\n", yylineno, yytext);
                        return 1;
                }
                vtoken = yylex();
                switch (ntoken) {
                case TYPE:
                case NAME:
                case TABLE_PREFIX:
                        if(vtoken != IDENTIFIER) {
                                printf("Syntax error in line %d, Expected an identifier but found %s\n", yylineno, yytext);
                                return 1;
                        }
                        printf("%s is set to %s\n", names[ntoken], yytext);
                        break;
                case PORT:
                        if(vtoken != INTEGER) {
                                printf("Syntax error in line %d, Expected an integer but found %s\n", yylineno, yytext);
                                return 1;
                        }
                        printf("%s is set to %s\n", names[ntoken], yytext);
                        break;
                default:
                        printf("Syntax error in line %d\n",yylineno);
```

scanner.c

```
                break;
            default:
                printf("Syntax error in line %d\n",yylineno);
            }
            ntoken = yylex();
        }
        return 0;
}
```

Input file – **input_f**

```
db_type mysql
db_name: testdata
db_table_prefix: test_
db_port: 1091
```

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_check.l
bivasm@cpusrv-gpu-108:~/lex$ gcc lex.yy.c scanner.c
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_f
1
Syntax error in line 1, Expected a ':' but found mysql
bivasm@cpusrv-gpu-108:~/lex$ vi input_f
bivasm@cpusrv-gpu-108:~/lex$
```

Input file – **input_f**

```
db_type: mysql
db_name: testdata
db_table_prefix: test_
db_port: 1091
```

```
bivasm@cpusrv-gpu-108:~/lex$ ./a.out<input_f
1
db_type is set to mysql
2
db_name is set to testdata
3
db_table_prefix is set to test_
4
db_port is set to 1091
bivasm@cpusrv-gpu-108:~/lex$
```

Lexical analyzer to recognize the following tokens

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

```
%{
    /* 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}+)?

%%

{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);}

%%
```

macros

(a) auxiliary declarations

(b) regular definitions

Symbols

{} for symbols usage
\ for meta-symbols (., * etc)

Regular definitions {} are used to define the RE of Rules

Seq. is important

Pattern { Action }

yylval: attributes

Recognition of Tokens

Regular Definitions for terminals

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

stripping out whitespace, by recognizing the "token" ***ws***

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

ASCII chars

- Token **ws** is different from the other tokens in that,
  - Once we recognize it, we **do not return it to the parser**,
- Rather **restart the lexical analysis** from the character that follows the whitespace. It is the following token that gets returned to the parser.

Auxiliary section

```
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 */
}
```

Conflict Resolution in Lex

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:

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.

yyin variable

yyin is a variable of the type FILE* and points to the input file.

- Defacto -- LEX assigns yyin to stdin(console input)

```
if( ! yyin )
yyin = stdin;
```

- If the programmer assigns an input file to yyin in the auxiliary functions section, then yyin is set to point to that file..

```
1
2       /* Declarations */
3   %%
4       /* Rules */
5   %%
6
7   main(int argc, char* argv[])
8   {
9           if(argc > 1)
10          {
11                  FILE *fp = fopen(argv[1], "r");
12                  if(fp)
13                          yyin = fp;
14          }
15          yylex();
16          return 1;
17  }
```

$./a.out input_file

yywrap()

- LEX **declares** the function yywrap() of return-type int in the file **lex.yy.c** .
- LEX **does not provide any definition** for yywrap().
- yylex() makes a **call to yywrap()** when it encounters the **end of input**.
- If **yywrap() returns zero** (indicating false) yylex() assumes **there is more input** and it **continues scanning** from the location **pointed to by yyin**.
- If **yywrap() returns a non-zero** value (indicating true), **yylex() terminates the scanning** process and **returns 0** (i.e. “wraps up”).
- If the programmer wishes to **scan more than one input file** using the generated lexical analyzer, it can be simply done by **setting yyin to a new input file in yywrap()** and **return 0**.
- As LEX does not define yywrap() in lex.yy.c file but **makes a call to it under yylex()**, the programmer **must define it** in the Auxiliary functions section **OR**
- provide **%option noyywrap** in the declarations section.
    - This options removes the call to yywrap() in the lex.yy.c file.

Homework

```
{
        int x;
        int y;
        x = 2;
        y = 3;
        x = 5 + y * 4;
}
```

I/P Character Stream	O/P Token Stream
{	<SPECIAL SYMBOL, {>
int x;	<KEYWORD, int> <ID, x> <PUNCTUATION, ;>
int y;	<KEYWORD, int> <ID, y> <PUNCTUATION, ;>
x = 2;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
y = 3;	<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
x = 5 + y * 4;	<ID, x> <OPERATOR, => <INTEGER CONSTANT, 5> <OPERATOR, +>
}	<ID, y> <OPERATOR, *> <INTEGER CONSTANT, 4> <PUNCTUATION, ;>
	<SPECIAL SYMBOL, }>

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT         "int"
ID          [a-z][a-z0-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]

%%
{INT}       { printf("<KEYWORD, int>\n"); /* Keyword Rule */ }
{ID}        { printf("<ID, %s>\n", yytext); /* Identifier Rule */}
"+"         { printf("<OPERATOR, +>\n"); /* Operator Rule */ }
"*"         { printf("<OPERATOR, *>\n"); /* Operator Rule */ }
"="         { printf("<OPERATOR, =>\n"); /* Operator Rule */ }
"{"         { printf("<SPECIAL SYMBOL, {>\n"); /* Scope Rule */ }
"}"         { printf("<SPECIAL SYMBOL, }>\n"); /* Scope Rule */ }
{PUNC}      { printf("<PUNCTUATION, ;>\n"); /* Statement Rule */ }
{CONST}     { printf("<INTEGER CONSTANT, %s>\n",yytext); /* Literal Rule */ }
{WS}        /* White-space Rule */ ;
%%
```

Complete example

```
%{
#define INT         10
#define ID          11
#define PLUS        12
#define MULT        13
#define ASSIGN      14
#define LBRACE      15
#define RBRACE      16
#define CONST       17
#define SEMICOLON   18
%}

INT         "int"
ID          [a-z][a-z0-9]*
PUNC        [;]
CONST       [0-9]+
WS          [ \t\n]

%%
{INT}    { return INT; }
{ID}     { return ID; }
"+"      { return PLUS; }
"*"      { return MULT; }
"="      { return ASSIGN; }
"{"      { return LBRACE; }
"}"      { return RBRACE; }
{PUNC}   { return SEMICOLON; }
{CONST}  { return CONST; }
{WS}     {/* Ignore
            whitespace */}
%%
```

```
main() { int token;
    while (token = yylex()) {
        switch (token) {
            case INT: printf("<KEYWORD, %d, %s>\n",
                token, yytext); break;
            case ID: printf("<IDENTIFIER, %d, %s>\n",
                token, yytext); break;
            case PLUS: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case MULT: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case ASSIGN: printf("<OPERATOR, %d, %s>\n",
                token, yytext); break;
            case LBRACE: printf("<SPECIAL SYMBOL, %d, %s>\n",
                token, yytext); break;
            case RBRACE: printf("<SPECIAL SYMBOL, %d, %s>\n",
                token, yytext); break;
            case SEMICOLON: printf("<PUNCTUATION, %d, %s>\n",
                token, yytext); break;
            case CONST: printf("<INTEGER CONSTANT, %d, %s>\n",
                token, yytext); break;
        }
    }
}
```