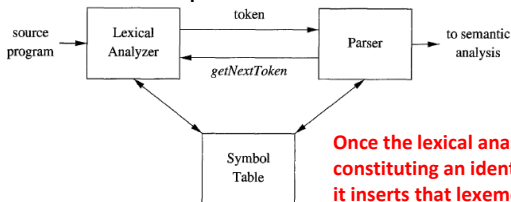


Lexical Analysis

Lexical Analysis

- ▶ The main task of the lexical analyzer is to
 - ▶ **read the input characters** of the source program,
 - ▶ **group** them into **lexemes**, and
 - ▶ produce as **output a sequence of tokens** for the source program.
 - ▶ **stripping out comments and whitespace** (blank, newline, tab etc), that are used to separate tokens in the input.
- ▶ Parser invokes the lexical analyzer by ***getNextToken*** command
- ▶ Lexical analyzer reads the characters from input until it finds the next lexeme and produce token



Once the lexical analyzer discovers a lexeme constituting an identifier, it inserts that lexeme into the symbol table.

Tokens, Patterns and Lexemes

- ▶ **Lexeme** : It is a **sequence of characters** in the source program that matches the **pattern**.

It is identified by the lexical analyzer as an instance of that token

- ▶ **Pattern**: Description of the form that the lexemes may take.
 - In the case of a **keyword**, the pattern is just the **sequence of characters** that form the keyword.
 - For **identifiers** and some other tokens, the pattern is a more complex structure that is matched by many strings.

- ▶ **Token** : It is a pair consisting of a token name and an optional attribute value.

$\langle token-name, attribute-value \rangle$

- The token name as an **abstract symbol** represents the kind of **lexical unit/lexeme** (keyword/identifier, operator symbol etc)
- Processed by parser

```
#include <stdio.h>
```

```
int main() {
```

```
    int number1, number2, sum;
```

```
    printf("Enter First Number: ");
```

```
    scanf("%d", &number1);
```

```
    printf("Enter Second Number: ");
```

```
    scanf("%d", &number2);
```

```
    // calculating sum
```

```
    sum = number1 + number2;
```

```
    printf("\nAddition of %d and %d is %d", number1, number2, sum);
```

```
    return 0;
```

```
}
```

Example of tokens

Pattern



relop

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i , f	if
else	characters e , l , s , e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token **comparison**
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Example of tokens

Pattern



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
relop comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

Find the tokens

```
printf("Total = %d\n", score);
```

Attribute for tokens

$\langle token\text{-}name, attribute\text{-}value \rangle$

- ▶ **Attribute** provides **additional piece of information** about a **lexeme**
 - ▶ Important for the code generator to know which lexeme was found in the source program
- ▶ Example: For the token **identifier id**, we need to associate with
 - ▶ its lexeme, its type, and the location at which it is first found
 - ▶ Attribute value for an identifier **id** is essentially a **pointer to the symbol-table** entry for that identifier
- ▶ Example: For the token **number**, attributes can be the respective numbers (1.3, 0 etc)

```
position = initial + rate * 60
```

$\langle id, 1 \rangle \langle = \rangle \langle id, 2 \rangle \langle + \rangle \langle id, 3 \rangle \langle * \rangle \langle 60 \rangle$

$\langle number, 60 \rangle$

1	position	...
2	initial	...
3	rate	...

Attribute for tokens

$\langle token\text{-}name, attribute\text{-}value \rangle$

- ▶ Attribute provides additional piece of information about a lexeme
 - ▶ Important for the code generator to know which lexeme was found in the source program

The token names and associated attribute values

E = M * C ** 2

<id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<number, integer value 2>

Scanning input from the source file

- Fast reading of the source program from disk
- **Challenge** to find lexemes
 - We often have to look one or more characters beyond the next lexeme
 - To ensure we have the right lexeme.

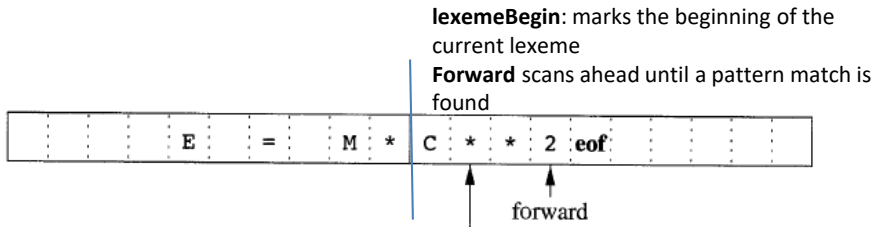
`- , = , or <`

Note the challenge!

`-> , == , or <= ,`


Scanning input from the source file

Two buffer solution



- Each buffer is of the same size N , **lexemeBegin**
- N is usually the size of a disk block (4KB).
- If fewer than N characters remain in the input file, then a special character, represented by **eof**

Advancing **forward** requires that

- (a) we first **test** whether we have reached the **end of one of the buffers**,
- (b) if so, we must **reload the other buffer** from the input, and move forward to the beginning of the newly loaded buffer.

```
#include <stdio.h>
```

```
int main() {
```

```
    int number1, number2, sum;
```

```
    printf("Enter First Number: ");
```

```
    scanf("%d", &number1);
```

```
    printf("Enter Second Number: ");
```

```
    scanf("%d", &number2);
```

```
    // calculating sum
```

```
    sum = number1 + number2;
```

```
    printf("\nAddition of %d and %d is %d", number1, number2, sum);
```

```
    return 0;
```

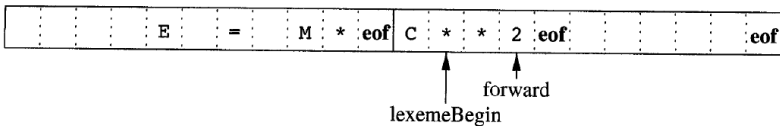
```
}
```

Scanning input from the source file

Sentinels (eof)

Each time we advance **forward**, we make two tests:

- (a) if we reached at the **end of the buffer**, and
- (b) determine what character is read----**test if the next lexeme** is determined;



- (a) We extend each buffer to hold a *sentinel eof* character at the end
- (b) **eof** retains its use as a marker for the end of the entire input.

Scanning input from the source file

Sentinels

```
switch ( *forward++ ) {  
    case eof:  
        if ( forward is at end of first buffer ) {  
            reload second buffer;  
            forward = beginning of second buffer;  
        }  
        else if ( forward is at end of second buffer ) {  
            reload first buffer;  
            forward = beginning of first buffer;  
        }  
        else /* eof within a buffer marks the end of input */  
            terminate lexical analysis;  
        break;  
    Cases for the other characters  
}
```

Specification of Tokens – Patterns

- **Regular expressions** are an important notation for specifying **lexeme patterns**.
 - A **string** over an **alphabet** is a finite sequence of symbols drawn from that alphabet
 - Represent all the **valid strings** with **Regular expressions**
- Suppose we wanted to describe the set of **valid C identifiers**
- **letter_** stands for any letter or the underscore $\{A, B, \dots, Z, a, b, \dots, z\}$
- **digit** stands for any digit $\{0, 1, \dots, 9\}$
- the language/RE of **C identifiers** $letter_ (letter_ | digit)^*$

OPERATION	DEFINITION AND NOTATION
<i>Union of L and M</i>	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
<i>Concatenation of L and M</i>	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
<i>Kleene closure of L</i>	$L^* = \bigcup_{i=0}^{\infty} L^i$
<i>Positive closure of L</i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Specification of Tokens

Regular Definitions

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\dots$$
$$d_n \rightarrow r_n$$

Regular expressions



1. Each d_i is a new symbol, not in Σ and not the same as any other of the d 's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$.

Regular definition for the language of C identifiers

$$letter_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$$
$$digit \rightarrow 0 \mid 1 \mid \dots \mid 9$$
$$id \rightarrow letter_ (letter_ \mid digit)^*$$

Specification of Tokens

Regular Definitions

Unsigned numbers (integer or floating point)

5280, 0.01234, 6.336E4, or 1.89E-4.

digit → 0 | 1 | ... | 9

digits → *digit digit**

optionalFraction → . *digits* | ϵ

optionalExponent → (E (+ | - | ϵ) *digits*) | ϵ

number → *digits optionalFraction optionalExponent*

Specification of Tokens

Notational extensions

One or more instances. The unary, postfix operator $^+$
 $r^* = r^+|\epsilon$ and $r^+ = rr^*$

Zero or one instance. The unary postfix operator $?$
 $r?$ is equivalent to $r|\epsilon$.

Character classes.

A regular expression $a_1|a_2|\cdots|a_n \Rightarrow [a_1a_2\cdots a_n] \Rightarrow a_1-a_n$

$$\begin{aligned}
 \textit{letter_} &\rightarrow \text{A} \mid \text{B} \mid \dots \mid \text{Z} \mid \text{a} \mid \text{b} \mid \dots \mid \text{z} \mid _ \\
 \textit{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
 \textit{id} &\rightarrow \textit{letter_} (\textit{letter_} \mid \textit{digit})^*
 \end{aligned}$$

$$\begin{aligned}
 \textit{letter_} &\rightarrow [\text{A-Za-z}] \\
 \textit{digit} &\rightarrow [0-9] \\
 \textit{id} &\rightarrow \textit{letter_} (\textit{letter} \mid \textit{digit})^*
 \end{aligned}$$

$$\begin{aligned}
 \textit{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
 \textit{digits} &\rightarrow \textit{digit} \textit{digit}^* \\
 \textit{optionalFraction} &\rightarrow . \textit{digits} \mid \epsilon \\
 \textit{optionalExponent} &\rightarrow (\text{E} (+ \mid - \mid \epsilon) \textit{digits}) \mid \epsilon \\
 \textit{number} &\rightarrow \textit{digits} \textit{optionalFraction} \textit{optionalExponent}
 \end{aligned}$$

$$\begin{aligned}
 \textit{digit} &\rightarrow [0-9] \\
 \textit{digits} &\rightarrow \textit{digit}^+ \\
 \textit{number} &\rightarrow \textit{digits} (. \textit{digits})? (\text{E} [+-]? \textit{digits})?
 \end{aligned}$$

Recognition of Tokens

Objective:

- Take the **patterns** for **all** the needed **tokens**
- Build a **tool** that examines the **input string** and **finds the lexeme** matching one of the **patterns**

<i>stmt</i>	→	if <i>expr</i> then <i>stmt</i>
		if <i>expr</i> then <i>stmt</i> else <i>stmt</i>
		ε
<i>expr</i>	→	<i>term</i> relop <i>term</i>
		<i>term</i>
<i>term</i>	→	id
		number

- The **terminals** of the grammar, --- **if, then, else, relop, id, number**,
 - lexical analyzer recognizes the terminals – Tokens

```
#include <stdio.h>
```

```
int main() {
```

```
    int number1, number2, sum;
```

```
    printf("Enter First Number: ");
```

```
    scanf("%d", &number1);
```

```
    printf("Enter Second Number: ");
```

```
    scanf("%d", &number2);
```

```
    // calculating sum
```

```
    sum = number1 + number2;
```

```
    printf("\nAddition of %d and %d is %d", number1, number2, sum);
```

```
    return 0;
```

```
}
```

Recognition of Tokens

Regular Definitions for terminals

digit → [0-9]
digits → *digit*⁺
number → *digits* (. *digits*)? (E [+ -]? *digits*)?
letter → [A-Za-z]
id → *letter* (*letter* | *digit*)^{*}
if → **if**
then → **then**
else → **else**
relop → < | > | <= | >= | = | <>

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

ws → (**blank** | **tab** | **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.

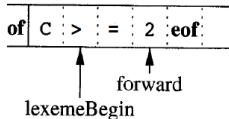
The goal for the lexical analyzer

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

Construction of the lexical analyzer

We first **convert patterns** into "transition diagrams" --- **Finite Automata**

Scanning the input looking for a lexeme

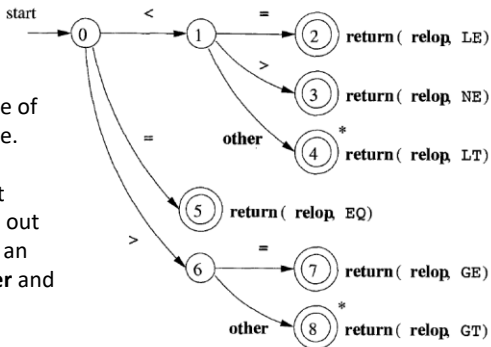


Finite Automata

Collection of nodes, called **states**

Edges are directed links from one state of the transition diagram to another state.

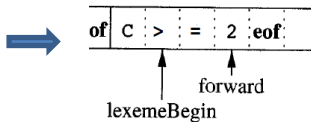
If we are in some **state S**, and the next input symbol is **a**, we look for an **edge** out of state **S** labeled by **a**. If we find such an edge, we **advance the forward pointer** and enter the next state **T**



Construction of the lexical analyzer

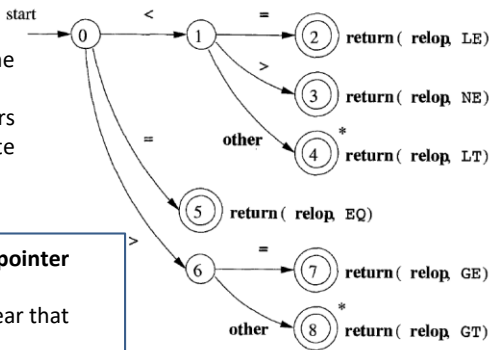
We first convert patterns into "transition diagrams" --- **Finite Automata**

Scanning the input looking for a lexeme



Finite Automata

- Certain states are said to be **accepting**
- These states indicate that a lexeme has been found between the **lexemeBegin** and **forward** pointers
- **Returning a token** and an attribute value to the parser



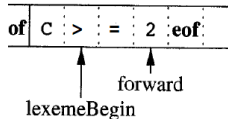
- If necessary, **retract** the **forward pointer** one position
 - additionally place a * near that accepting state

Construction of the lexical analyzer:

token relop

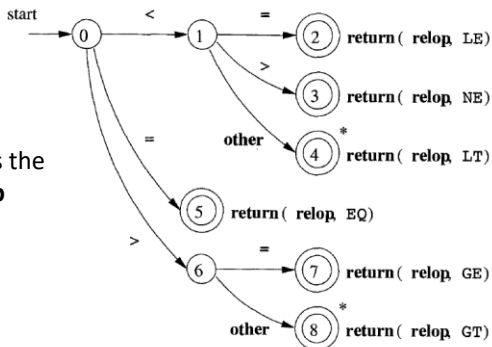
We first convert patterns into "transition diagrams" --- **Finite Automata**

Scanning the input looking for a lexeme



relop → < | > | <= | >= | = | <>

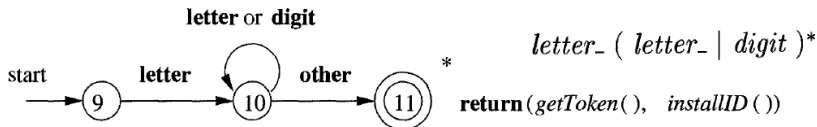
transition diagram that recognizes the lexemes matching the **token relop**



Construction of the lexical analyzer:

Keywords and Identifiers

Challenge: Discriminate between **Keywords** and **Identifiers**



Lexeme	Token	Attrb
if	IF	
else	ELSE	
count	ID	float, ..

Install the **keywords** in the symbol table initially, with **tokens**

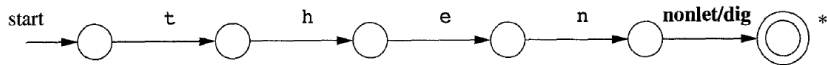
- Once we find an **identifier**, we invoke **installID** to insert it in the symbol table if it is not already in symbol table
- returns a pointer to the symbol-table entry

The function **getToken** examines the symbol table entry for the **lexeme found**, and returns whatever token name — either **ID** or one of the **keyword** tokens

Construction of the lexical analyzer:

Keywords and Identifiers

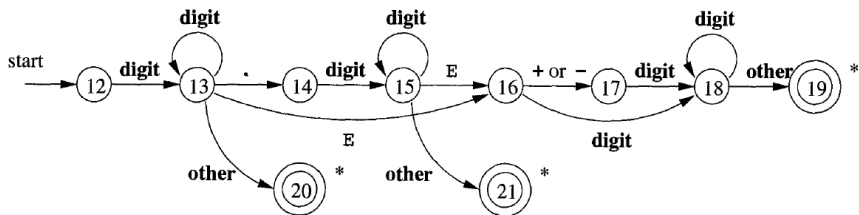
Create separate transition diagrams for each keyword



Differentiates **then** and
then_value

Construction of the lexical analyzer:

: Unsigned numbers



$$\begin{aligned}
 \text{letter_} &\rightarrow \text{A} \mid \text{B} \mid \dots \mid \text{Z} \mid \text{a} \mid \text{b} \mid \dots \mid \text{z} \mid _ \\
 \text{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
 \text{id} &\rightarrow \text{letter_} (\text{letter_} \mid \text{digit})^*
 \end{aligned}$$

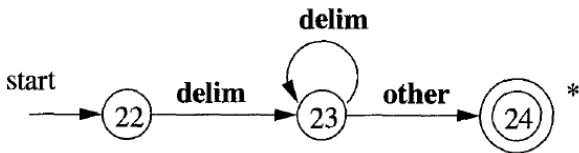
$$\begin{aligned}
 \text{letter_} &\rightarrow [\text{A-Za-z}] \\
 \text{digit} &\rightarrow [0-9] \\
 \text{id} &\rightarrow \text{letter_} (\text{letter} \mid \text{digit})^*
 \end{aligned}$$

$$\begin{aligned}
 \text{digit} &\rightarrow 0 \mid 1 \mid \dots \mid 9 \\
 \text{digits} &\rightarrow \text{digit} \text{ digit}^* \\
 \text{optionalFraction} &\rightarrow . \text{ digits} \mid \epsilon \\
 \text{optionalExponent} &\rightarrow (\text{E} (+ \mid - \mid \epsilon) \text{ digits}) \mid \epsilon \\
 \text{number} &\rightarrow \text{digits optionalFraction optionalExponent}
 \end{aligned}$$

$$\begin{aligned}
 \text{digit} &\rightarrow [0-9] \\
 \text{digits} &\rightarrow \text{digit}^+ \\
 \text{number} &\rightarrow \text{digits} (. \text{ digits})? (\text{E} [+-]? \text{digits})?
 \end{aligned}$$

Construction of the lexical analyzer: : whitespace

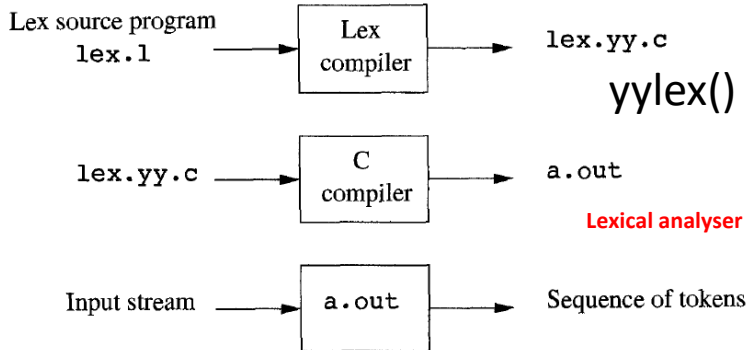
$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$



- When we **recognize ws**, we **do not return it to the parser**, but rather **restart the lexical analysis** from the character that follows the whitespace.
- It is the following token that gets returned to the parser.

Lex or flex

- Allows one to **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**
 - Tool itself is the **Lex compiler**
- Lex compiler **transforms** the **input patterns** into a **transition diagram**
 - Generates **code**, in a file called **lex.yy.c**, that simulates this transition diagram.



Structure of Lex Programs

A Lex program has the following form:

(a) auxiliary declarations

(b) regular definitions

declarations

%%

translation rules

%%

auxiliary functions

Additional functions --- say main()

etc

yylex()

Pattern { Action }


Structure of Lex Programs

A Lex program has the following form:

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

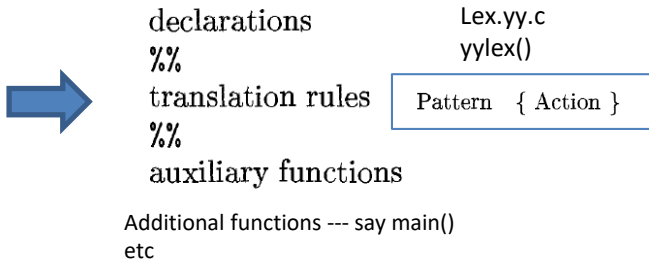
- Enclosed within %{ and %}
- Auxiliary declarations are copied as such by LEX to the output lex.yy.c file.
 - Not processed by the LEX tool.

(b) regular definitions

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

Structure of Lex Programs

A Lex program has the following form:



(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()

Recognition of Tokens

Regular Definitions for terminals

digit → [0-9]
digits → *digit*⁺
number → *digits* (. *digits*)? (E [+ -]? *digits*)?
letter → [A-Za-z]
id → *letter* (*letter* | *digit*)^{*}
if → **if**
then → **then**
else → **else**
relop → < | > | <= | >= | = | <>

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

ws → (**blank** | **tab** | **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.

Specification of Tokens

Notational extensions

One or more instances. The unary, postfix operator $^+$
 $r^* = r^+|\epsilon$ and $r^+ = rr^*$

Zero or one instance. The unary postfix operator $?$
 $r?$ is equivalent to $r|\epsilon$.

Character classes.

A regular expression $a_1|a_2|\cdots|a_n \Rightarrow [a_1a_2\cdots a_n] \Rightarrow a_1-a_n$

Structure of Lex Programs

A Lex program has the following form:

(a) auxiliary declarations

(b) regular definitions

declarations

%%

translation rules

%%

auxiliary functions

Additional functions --- say main()
etc

yylex()

Pattern { Action }



- **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>  
%}
```

yylex()

```
/* 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);  
}
```

- 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

Input file – **input_first**

if + 78 else 0

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

Lex – example-1

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

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

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

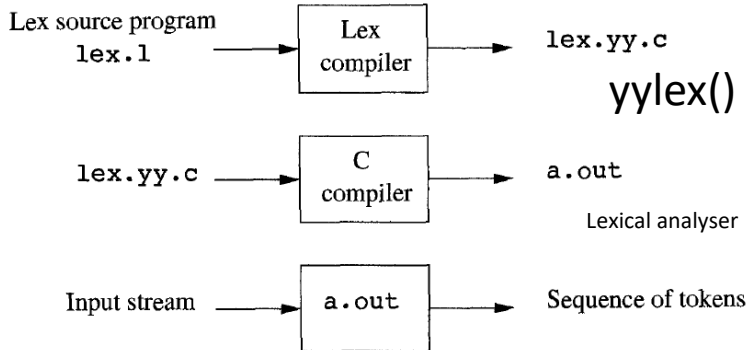
Each invocation of the function **yylex()** results in **yytext** carrying a pointer to the **lexeme** found in the input stream by **yylex()**

```
%%  
/* Auxiliary functions */
```

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

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

lex_first.l





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

```
%{#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);
%%
```

lex_first_v1.l


```

%%
/* 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_v1.1
bivasm@cpusrv-gpu-108:~/lex$ ls
a.out  head.h  input_f  input_first  le  lex_check.1
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
```

lex_first_v2.l

```
%}
```

```
/* Declarations*/
```

```
%%
```

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

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



 bivasm@cpusrv-gpu-108: ~/lex

```
bivasm@cpusrv-gpu-108:~/lex$ lex lex_first_v2.1
```

```
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

```
%{  
#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 */
```

Note: main() is missing

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

```
#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 v3.1
```

```
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

```
%{  
#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("unexpected\n");  
  
%%  
int yywrap(void)  
{  
    return 1;  
}
```

lex_check.l

Recognition of Tokens

Regular Definitions for terminals

digit → [0-9]
digits → *digit*⁺
number → *digits* (. *digits*)? (E [+ -]? *digits*)?
letter → [A-Za-z]
id → *letter* (*letter* | *digit*)^{*}
if → **if**
then → **then**
else → **else**
relop → < | > | <= | >= | = | <>

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

ws → (**blank** | **tab** | **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 <i>ws</i>	-	-
if	if	-
then	then	-
else	else	-
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	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 */
}%
```

macros

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

(a) auxiliary declarations

(b) regular definitions

Symbols

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

```
%%
```

```
{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); }
```

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

Pattern { Action }

yylval: attributes

Seq. is →
important

```
%%
```

Recognition of Tokens

Regular Definitions for terminals

digit → [0-9]
digits → *digit*⁺
number → *digits* (. *digits*)? (E [+ -]? *digits*)?
letter → [A-Za-z]
id → *letter* (*letter* | *digit*)^{*}
if → **if**
then → **then**
else → **else**
relop → < | > | <= | >= | = | <>

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

ws → (**blank** | **tab** | **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 the programmer assigns an input file to yyin in the auxiliary functions section, then yyin is set to point to that file..

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

```
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”). **[you define, you specify the return type of yywrap()!!]**
- 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;  
}
```

```

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

```

```

<SPECIAL SYMBOL, {>
<KEYWORD, int> <ID, x> <PUNCTUATION, ;>
<KEYWORD, int> <ID, y> <PUNCTUATION, ;>
<ID, x> <OPERATOR, => <INTEGER CONSTANT, 2> <PUNCTUATION, ;>
<ID, y> <OPERATOR, => <INTEGER CONSTANT, 3> <PUNCTUATION, ;>
<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;
            }
        }
    }
```