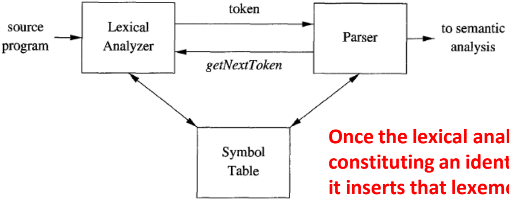
# Lexical Analysis

# 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**.

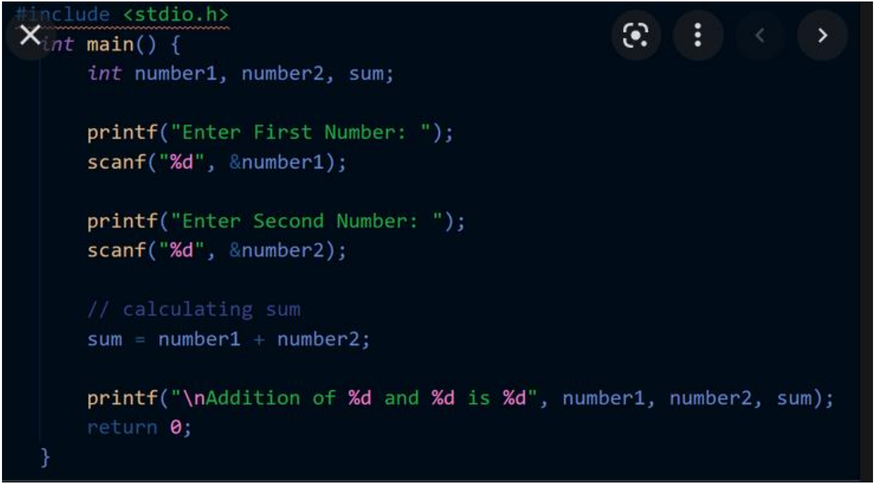
  It is identified by the lexical analyzer as an instance of that token

- **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.

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

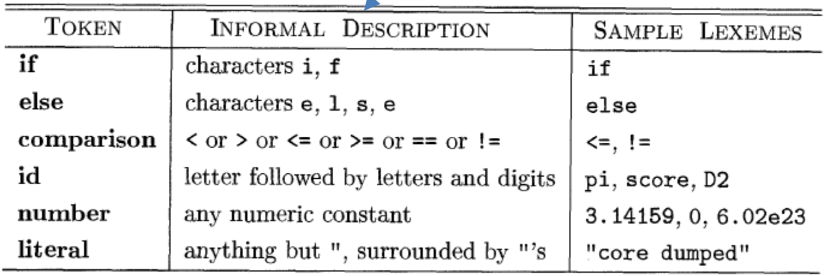
  - The token name as an **abstract symbol** represents the kind of **lexical unit/lexeme** (keyword/identifier, operator symbol etc)
  - Processed by parser



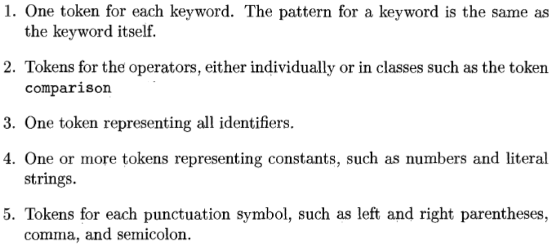
```c
#include <stdio.h>
int main() {
    int number1, number2, sum;

    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



| | 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"` |

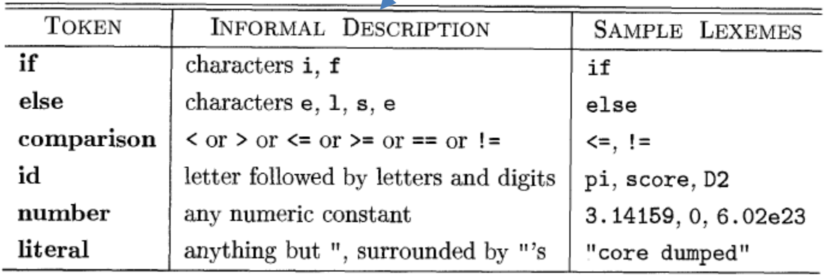


1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token `comparison`
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal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

## Example of tokens

Pattern

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"` |

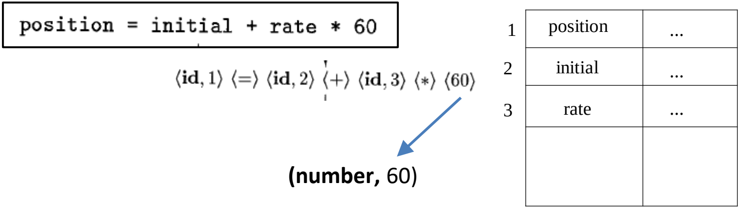
# Find the tokens

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

# Attribute for tokens

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

- **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
```

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

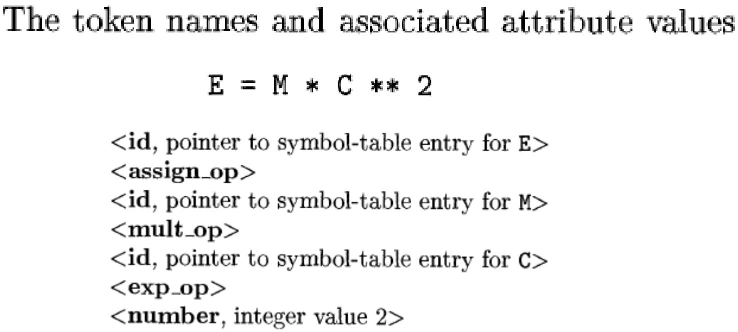
**(number,** 60)

| | | |
|---|---|---|
| 1 | position | ... |
| 2 | initial | ... |
| 3 | rate | ... |
| | | |

# Attribute for tokens

⟨*token-name*, *attribute-value*⟩

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



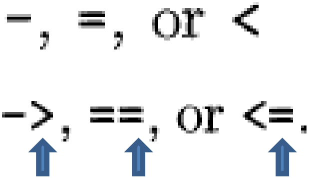
The token names and associated attribute values

```
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 <

−>, ==, or <=.

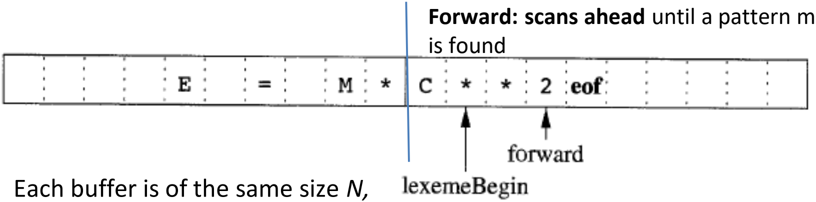
**Note the challenge!**

# Scanning input from the source file

## Two buffer solution

**lexemeBegin**: marks the **beginning** of the current lexeme

**Forward: scans ahead** until a pattern match is found



- Each buffer is of the same size *N,*
- *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**,

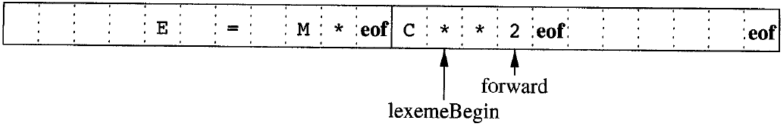
(b) if so, we must **reload the other buffer** from the input, and move forward to the beginning of the newly loaded buffer.

# 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.

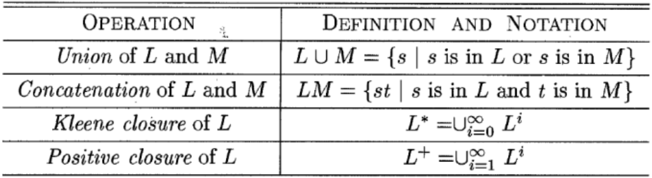
# Scanning input from the source file

## 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 $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$
- **digit** stands for any digit $\{0, 1, \ldots 9\}$
- the language/**RE** of **C identifiers** $letter\_ \; ( \; letter\_ \mid digit \; )^*$



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

# Specification of Tokens

**Regular Definitions**

Regular expressions

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

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

**Regular definition for the language of C identifiers**

$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^* \end{array}$$

# Specification of Tokens

## Regular Definitions

### Unsigned numbers (integer or floating point)

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

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{optionalFraction} & \rightarrow & .\ \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\ \mathtt{E}\ (\ + \mid - \mid \epsilon\ )\ \mathit{digits}\ ) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits}\ \mathit{optionalFraction}\ \mathit{optionalExponent}
\end{array}
$$

# Specification of Tokens

**Notational extensions**

*One or more instances.* The unary, postfix operator $^+$

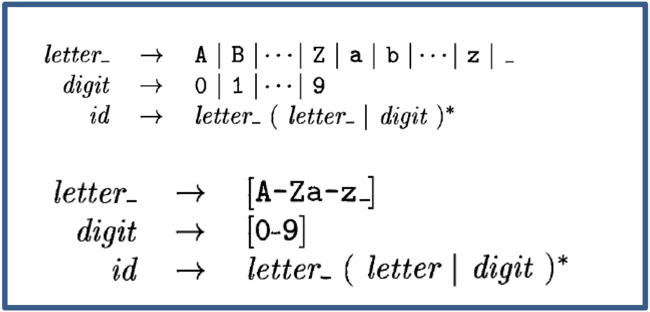
$$r^* = r^+|\epsilon \text{ and } r^+ = rr^*$$

*Zero or one instance.* The unary postfix operator ?

$r?$ is equivalent to $r|\epsilon$,

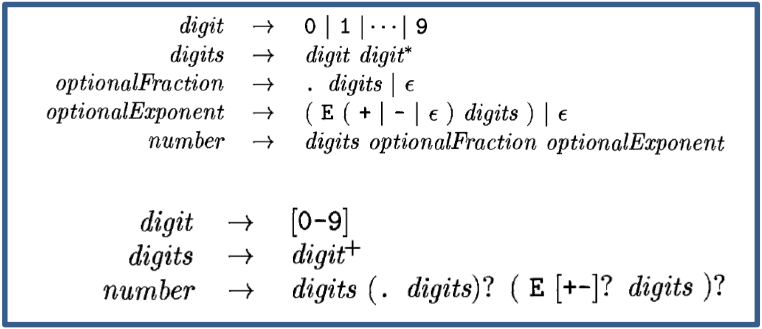
*Character classes.*

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



$$\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^{*}
\end{array}$$

$$\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & [\texttt{A-Za-z\_}] \\
\mathit{digit} & \rightarrow & [\texttt{0-9}] \\
\mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*}
\end{array}$$



$$\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{optionalFraction} & \rightarrow & \texttt{.}\ \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ \mathit{digits}\ ) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits}\ \mathit{optionalFraction}\ \mathit{optionalExponent}
\end{array}$$

$$\begin{array}{rcl}
\mathit{digit} & \rightarrow & [\texttt{0-9}] \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \ \mathit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \ \mathit{digits}\ )?
\end{array}$$

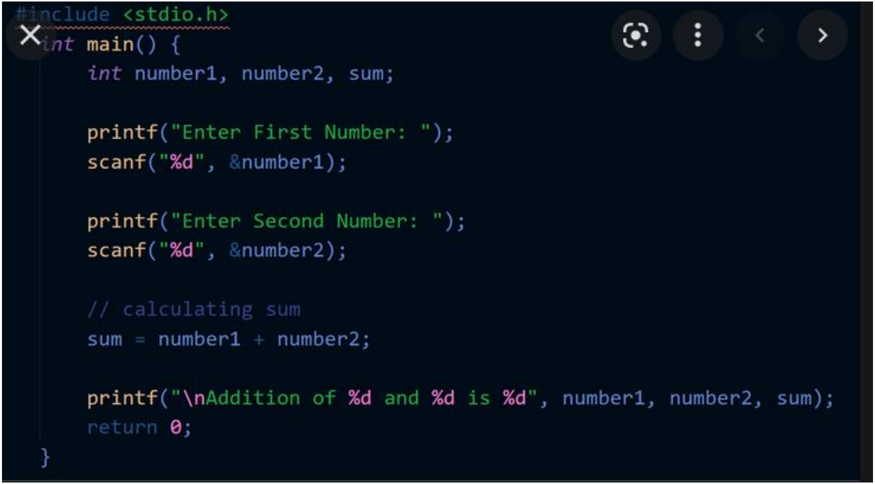
# Recognition of Tokens

**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**

$$\begin{array}{lcl}
stmt & \rightarrow & \mathbf{if}\ expr\ \mathbf{then}\ stmt \\
 & | & \mathbf{if}\ expr\ \mathbf{then}\ stmt\ \mathbf{else}\ stmt \\
 & | & \epsilon \\
expr & \rightarrow & term\ \mathbf{relop}\ term \\
 & | & term \\
term & \rightarrow & \mathbf{id} \\
 & | & \mathbf{number}
\end{array}$$

- 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

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

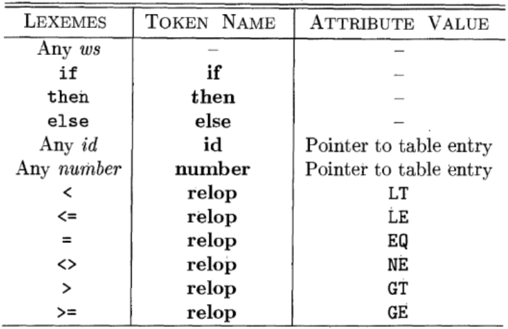
stripping out whitespace, by recognizing the "token" ***ws***

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

**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.

# The goal for the lexical analyzer

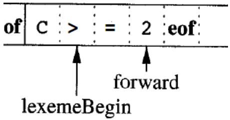


| LEXEMES | TOKEN NAME | ATTRIBUTE VALUE |
|---|---|---|
| Any *ws* | – | – |
| `if` | **if** | – |
| `then` | **then** | – |
| `else` | **else** | – |
| Any *id* | **id** | Pointer to table entry |
| Any *number* | **number** | Pointer to table entry |
| `<` | **relop** | `LT` |
| `<=` | **relop** |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

# Construction of the lexical analyzer

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

**Scanning the input** looking for a lexeme

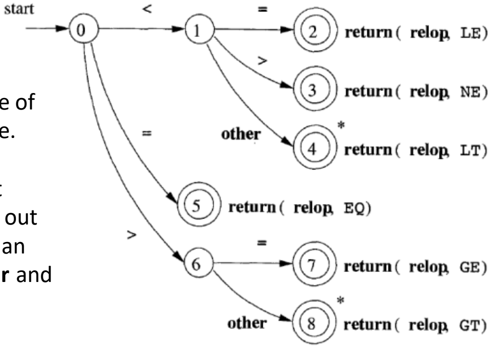


## Finite Automata

Collection of nodes, called **states**

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

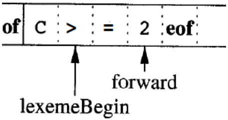
If we are in some **state S**, and the next input symbol is **a**, we look for an **edge** out of state **S** labeled by **a**. If we find such an edge, we **advance the forward pointer** and enter the next state **T**



# Construction of the lexical analyzer

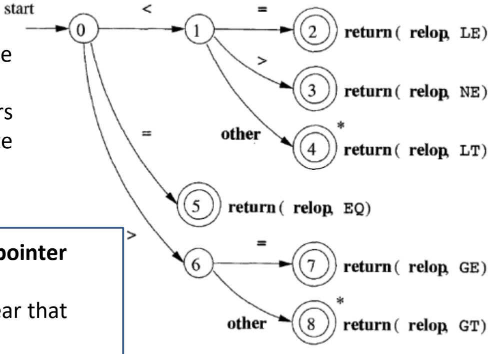
We first **convert patterns** into "**transition diagrams**" --- **Finite Automata**

**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

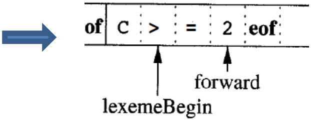


- If necessary, **retract** the **forward pointer** one position
    - additionally place a * near that accepting state

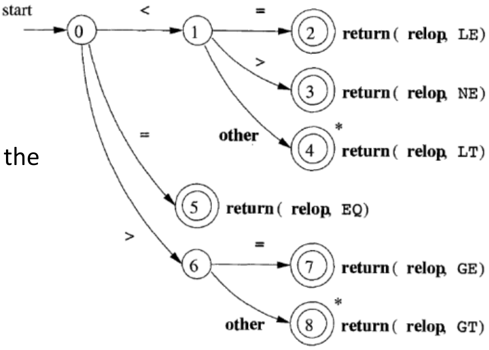
# Construction of the lexical analyzer: **token relop**

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

Scanning the input looking for a lexeme



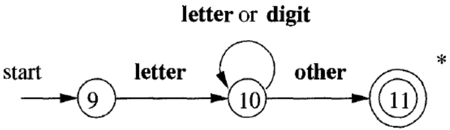
*relop* → < | > | <= | >= | = | <>



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

# Construction of the lexical analyzer:
# **Keywords and Identifiers**

Challenge: Discriminate between **Keywords and Identifiers**



$$letter\_ \; ( \; letter\_ \mid digit \; )^*$$

**return** (*getToken*( ), *installID* ( ))

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

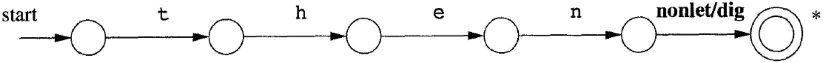
Install the **keywords** in the symbol table initially, with **tokens**

- 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**

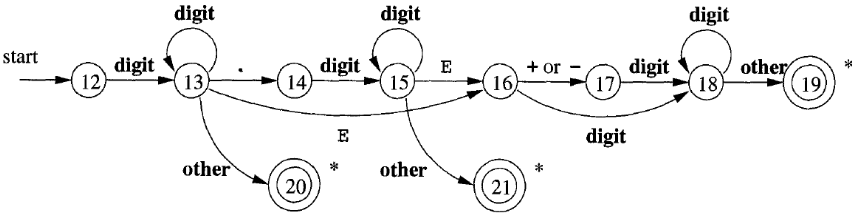
Create separate transition diagrams for each keyword

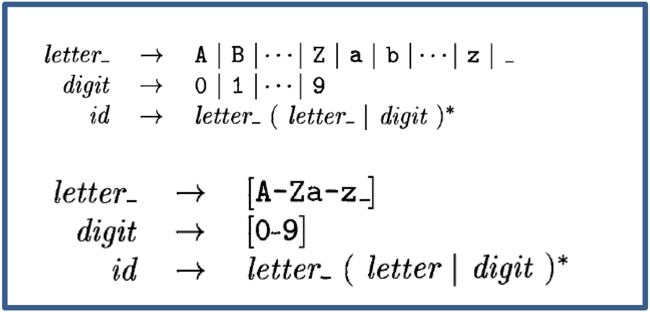


Differentates **then** and **then_value**

Keyword generating transition diagrams gets priority over ID

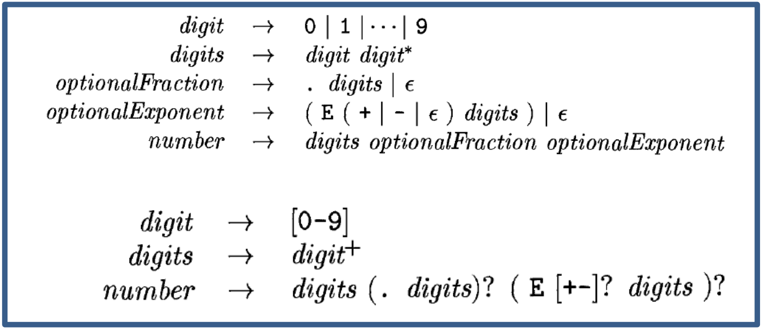
# Construction of the lexical analyzer:
# : **Unsigned numbers**





$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^* \end{array}$$

$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \texttt{[A-Za-z\_]} \\ \mathit{digit} & \rightarrow & \texttt{[0-9]} \\ \mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter} \mid \mathit{digit} \;)^* \end{array}$$

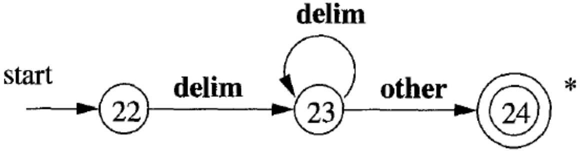


$$\begin{array}{rcl} \mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{digits} & \rightarrow & \mathit{digit} \; \mathit{digit}^* \\ \mathit{optionalFraction} & \rightarrow & \texttt{.} \; \mathit{digits} \mid \epsilon \\ \mathit{optionalExponent} & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\ \mathit{number} & \rightarrow & \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent} \end{array}$$

$$\begin{array}{rcl} \mathit{digit} & \rightarrow & \texttt{[0-9]} \\ \mathit{digits} & \rightarrow & \mathit{digit}^+ \\ \mathit{number} & \rightarrow & \mathit{digits} \; (\texttt{.} \; \mathit{digits})? \; (\; \texttt{E} \; \texttt{[+-]}? \; \mathit{digits} \;)? \end{array}$$

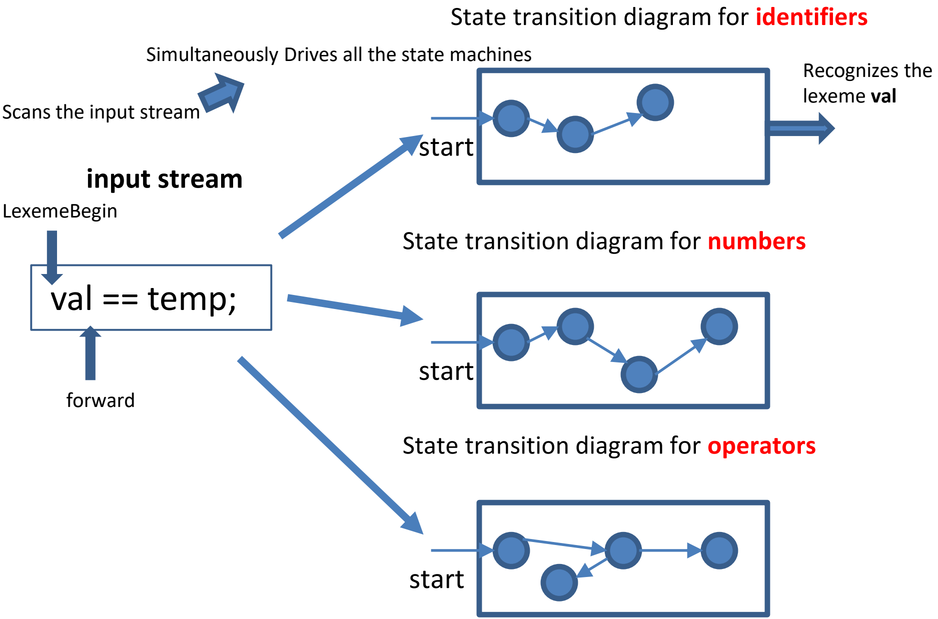
# Construction of the lexical analyzer:
# : **whitespace**

$$ws \rightarrow (\ \mathbf{blank}\ |\ \mathbf{tab}\ |\ \mathbf{newline}\ )^{+}$$



- 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.

# Lexical analyzer in action



Simultaneously Drives all the state machines

Scans the input stream

**input stream**

LexemeBegin

val == temp;

forward

State transition diagram for **identifiers**

start

Recognizes the lexeme **val**

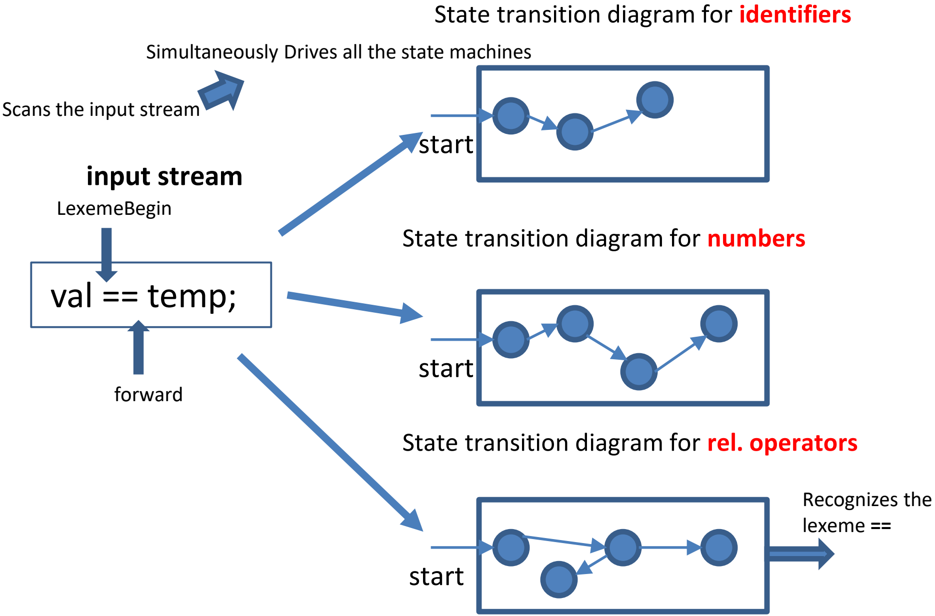
State transition diagram for **numbers**

start

State transition diagram for **operators**

start

# Lexical analyzer in action



State transition diagram for **identifiers**

Simultaneously Drives all the state machines

Scans the input stream

**input stream**

LexemeBegin

val == temp;

forward

start

State transition diagram for **numbers**

start

State transition diagram for **rel. operators**

start

Recognizes the lexeme ==