

COP 3402 Systems Software

Eurípides Montagne
University of Central Florida

COP 3402 Systems Software

Lexical analysis

Outline

1. Lexical analyzer
2. Designing a Scanner
3. Regular expressions
4. Transition diagrams

Lexical Analyzer

The purpose of the scanner is to decompose the source program into its elementary symbols or tokens.

1. Read input one character at a time
2. Group characters into lexical units
3. Remove white spaces, comments and control characters
4. Encode lexical units into token types
5. Detect errors and generate error messages

Lexical analyzer

The stream of characters in the assignment statement

`\tfahrenheit := 32 + celsious * 1.8;\n` `/* Hello */`

↑ ↑ ↑ ↑

control characters white spaces control characters comments

is read in by the scanner and the scanner translates it into a stream of tokens in order to ease the task of the Parser.

`[id, 1] [: =] [int, 32] [+] [id, 2] [*] [int, 1.8] [;]`

`02 fahrenheit 08 03 32 12 02 celsious 09 03 1.8 17`

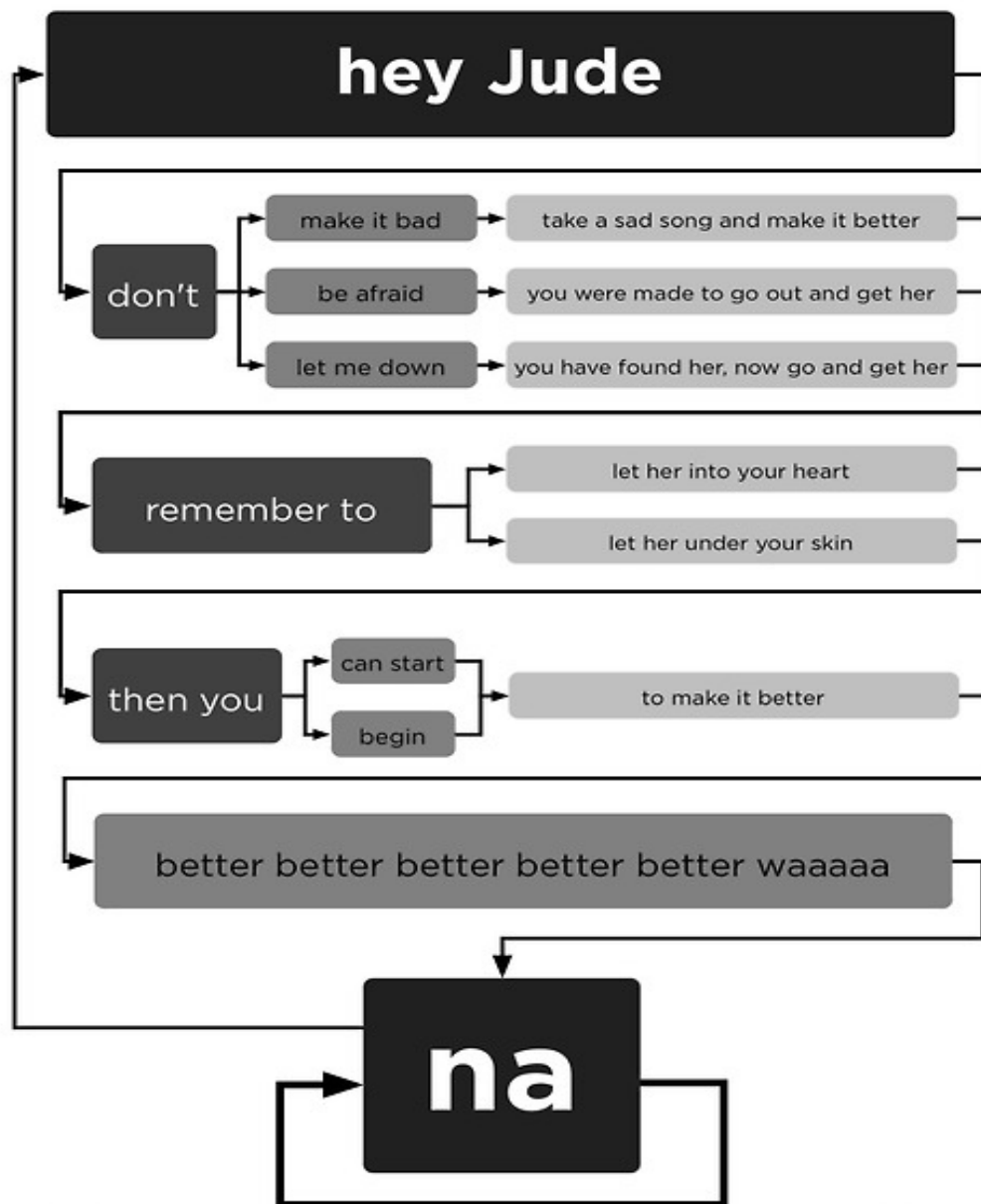
Scanner eliminates white spaces, comments, and control characters.

Lexical Analyzer

1. Lookahead plays an important role to a lexical analyzer.
2. It is not always possible to decide if a token has been found without looking ahead one character.

For instance, if only one character, say “i”, is used it would be impossible to decide whether we are in the presence of identifier “i” or at the beginning of the reserved word “if”.

3. We need to ensure a unique answer and that can be done knowing what is the character ahead.



loveallthis.tumblr.com

lyrics © sony atv

Designing a scanner

Define the token types (internal representation)

Create tables with initial values:

Reserved words name table: **begin, call, const, do, end, if, odd, procedure, then, var, while.**

Special symbols table: **‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’, ‘=’, ‘,’ , ‘:’, ‘<’, ‘>’, ‘;’.**

External name table (usually known as the symbol table)

Designing a scanner

Examples:

```
#define norw    15      /* number of reserved words */
#define imax   32767    /* maximum integer value */
#define cmax    11      /* maximum number of chars for ids */
#define strmax  256     /* maximum length of strings */
```

Internal representation of PL/0 Symbols

token types example:

```
typedef enum { nulSYM = 1, idSYM, numSYM, plusSYM, minusSYM,
multSYM, slashSYM, oodSYM, eqSYM, neqSYM, lessSYM, leqSYM,
gtrSYM, geqSYM, lparenSYM, rparenSYM, commaSYM, semicolonSYM,
periodSYM, becomeSYM, beginSYM, endSYM, ifSYM, thenSYM,
whileSYM, doSYM, callSYM, constSYM, varSYM, procSYM, writeSYM
} token_type;
```

Designing a scanner

/ list of reserved word names */*

```
char *word [ ] = { "null", "begin", "call", "const", "do", "else", "end", "if",  
                  "odd", "procedure", "read", "then", "var", "while", "write"};
```

/ internal representation of reserved words */*

```
int wsym [ ] = { nulsym, beginsym, callsym, constsym, dosym, elsesym, endsym, ifsym,  
               oddsym, procsym, readsym, thensym, varsym, whilesym, writesym};
```

/ list of special symbols */*

```
Int ssym[256]
```

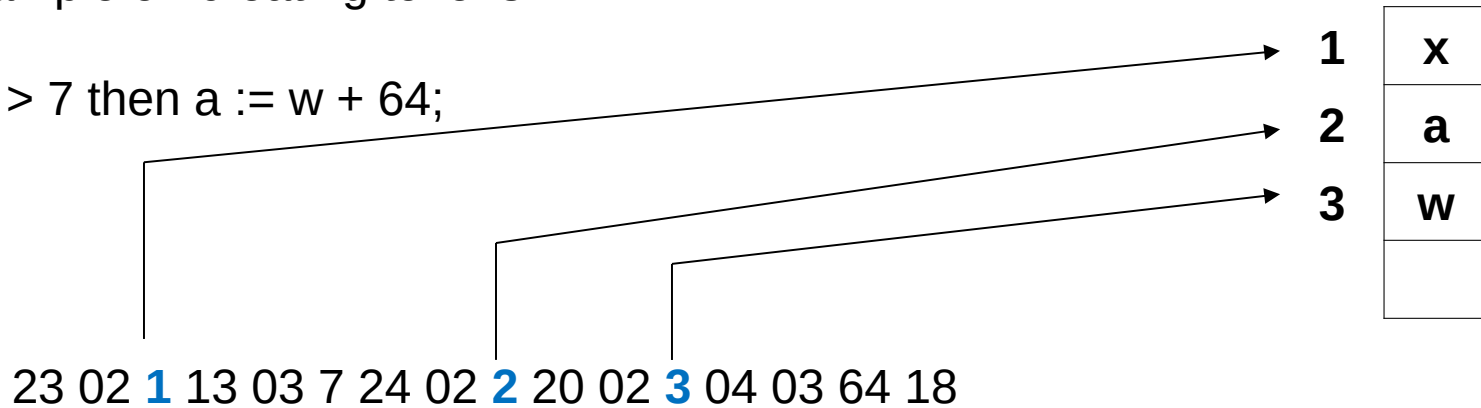
ssym['+']=plus;	ssym['-']=minus;	ssym['*']=mult;
ssym['/']=slash;	ssym['(']=lparen;	ssym[')']=rparen;
ssym['=']=eq;	ssym[',']=comma;	ssym['.']=period;
ssym['#']=neq;	ssym['<']=lss;	ssym['>']=gtr;
ssym['\$']=leq;	ssym['%']=geq;	ssym[';']=semicolon;

Name Table

The auxiliary name table (an array) can be define to keep the identifier names.

Example on creating tokens:

if x > 7 then a := w + 64;



ASCII Character Set

The ordinal number of a character *ch* is computed from its coordinates (X,Y) in the table as:

$$\text{ord}(ch) = 16 * X + Y$$

Y

Example:

$$\text{ord}('A') = 16 * 4 + 1 = 65$$

X

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
10(A)	LF	SUB	*	:	J	Z	j	z
11(B)	VT	ESC	+	;	K	[k	{
12(C)	FF	FS	,	<	L	\	l	
13(D)	CR	GS	-	=	M]	m	}
14(E)	SO	RS	.	>	N	^	n	~
15(F)	SI	US	/	?	O	_	o	DEL

Regular expressions

An **alphabet** is any finite set of symbols and usually the greek letter ***sigma*** (Σ) is used to denote it.

For example:

$\Sigma = \{0,1\}$ \rightarrow the binary alphabet

Note: **ASCII** is an important example of an alphabet; it is used in many software systems

A **string** (string = sentence = word) over an alphabet is a finite sequence of symbols drawn from that alphabet.

For example:

$\Sigma = \{0,1\}$ $s = 1011 \rightarrow$ denotes a string called **s**

Note: any sequence of **0** and **1** is a string over the alphabet $\Sigma = \{0,1\}$

Regular expressions

Example 2:

Alphabet

$\Sigma = \{a, b, c, \dots, z\}$

Strings

while, for, const.

The **length** of a string **s**, usually written $|s|$, is the number of occurrences of symbols in s.

For example:

If $s = \mathbf{while}$ the value of $|s| = 5$

Note: the empty string, denoted ϵ (**epsilon**), is the string of length zero.

$|\epsilon| = 0$

Regular expressions

A *language* is any countable set of strings over some fixed alphabet.

For example:

Let **L** be the alphabet of letters and **D** be the alphabet of digits:

$$\mathbf{L} = \{ A, B, \dots, Z, a, b, \dots, z \} \text{ and } \mathbf{D} = \{ 0, 1, 2, 3, \dots, 8, 9 \}$$

Note: **L** and **D** are languages. All strings in **L** and **D** happen to be of length one. Therefore, and equivalent definition is:

L is the alphabet of uppercase and lowercase letters.

D is the alphabet of digits.

Regular expressions

Other languages that can be constructed from **L** and **D** are:

- 1) **L U D** → the language with 62 strings of length one.
- 2) **L D** → is the set of 520 strings of length two each containing a letter followed by a digit.
- 3) **L³** → is the set of all 3-letter strings.
- 4) **L*** → is the set of all strings (of any length) of letters, including ϵ the empty string. Formally this is called Kleene closure of L.

The star means “zero or more occurrences”.

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$$

Regular expressions

5) $D^+ \rightarrow$ is the set of all strings of one or more digits.

$$D^+ = D \quad D^* = D^1 \cup D^2 \cup D^3 \cup \dots$$

6) $L (L \cup D)^* \rightarrow$ is the set of all strings of letters and digits beginning with a letter.

For example: **while, for, salary, intel486**

Definition: A **Regular Expressions** is a notation for describing all valid strings (of a language) that can be built from an alphabet. (or a set of characters that specify a pattern)

Regular expressions

Each regular expression r denotes a language $L(r)$

Rules that define a regular expression:

- 1) ϵ (*epsilon*) is a regular expression denoting the language $L(\epsilon) = \{ \epsilon \}$.
- 2) Every element in Σ (*sigma*) is a regular expression. If a is a symbol in Σ , then a is a regular expression, and $L(a) = \{a\}$.
- 3) Given two regular expressions r and s , rs is a regular expression denoting the language $L(r) L(s)$.
- 4) Given two regular expressions r and s , $r \cup s$ is a regular expression denoting the language $L(r) \cup L(s)$.
- 5) Given a regular expression r , r^* is a regular expression.
- 6) Given a regular expression r , r^+ is a regular expression.
- 7) Given a regular expression r , (r) is a regular expression

Regular expressions

For example, given the alphabet:

$$\Sigma = \{ A, B, \dots, Z, a, b, \dots, z, 0, 1, 2, 3, \dots, 8, 9 \}$$

ϵ is a regular expression denoting $\{ \epsilon \}$, the empty string.

a is a regular expression denoting $\{ a \}$.

Any symbol from Σ is a regular expression.

If a and b are regular expressions, then:

$a \mid b$ denotes the language $\{ a, b \}$. \leftarrow choice among alternatives

For example:

$(a \mid b)(a \mid b)$ denotes $\{ aa, ab, ba, bb \}$

The language of all strings of length two over the alphabet Σ .

Regular expressions

a . b denotes the regular expression { **ab** }. ← concatenation

The language (L^2) consisting of the string { **ab** }.
(we will use the notation **ab** instead of **a . b**)

a* denotes the language consisting of all strings of zero or more **a**'s,
that is:

{ ϵ , **a**, **aa**, **aaa**, **aaaa**, ... }

(**a | b**)^{*} denotes the set of all strings consisting of zero or more
instances of **a** or **b**.

For example:

{ ϵ , **a**, **b**, **aa**, **ab**, **ba**, **bb**, **aaa**, ... }

Regular expressions

What is the language denoted by $a \mid a^* b$?

$\{ a, b, ab, aab, aaab, \dots \}$

There are different notations to describe a language. For example,

$L^2 = \{ aa, ab, ba, bb \}$

Or using the regular expression:

$L^2 \rightarrow aa \mid ab \mid ba \mid bb$

This will allow us to describe identifiers in PL/0 as:

letter $\rightarrow A \mid B \mid C \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$

digit $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

id $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

Regular expressions

Remember !

A **language** is any countable set of strings over some fixed alphabet.

Each string from the language is called a **word** or **sentence**.

Given the following alphabet $\Sigma = \{a, b\}$, each one of the following sets is a language over the fixed alphabet $\{a, b\}$:

$$L = \{a, b, ab\} \quad M = \{a, b, ab, aab, aaab, \dots\}$$

Language **L** can be defined by explicit enumeration but **M** can not.

A **regular expression** is a type of grammar that specifies a set of strings and can be used to denote a language over an alphabet .

(i.e., The regular expression $a | a^* b$ denotes the language **M** over Σ)

Regular expressions

Extensions of regular expressions notation:

1) One or more repetitions: “+” .

For example: $(a \mid b)^+ = (a \mid b) (a \mid b)^*$

2) Zero or one instance: “?”

For example: $(+ \mid -)^? (\text{digit})^+ = (\text{digit})^+ \mid + (\text{digit})^+ \mid - (\text{digit})^+$

3) A range of characters: “[... - ...]”

For example: $a \mid b \mid c \mid \dots \mid z = [a - z]$

Example: letter $\rightarrow [A - Za - z]$
 digit $\rightarrow [0 - 9]$
 id $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

Lexemes, Patterns and Tokens

A **Lexeme** is the sequence of input characters in the source program that matches the pattern for a token (the sequence of input characters that the token represents).

A **Pattern** is a description of the form that the lexemes of a token may take.

A **Token** is the internal representation of a lexeme. Some tokens may consist only of a name (internal representation) while others may also have some associated values (attributes) to give information about a particular instance of a token.

Example:

<u>Lexeme</u>	<u>Pattern</u>	<u>Token</u>	<u>Attribute</u>
Any identifier	letter(letter digit)*	idsym	pointer to symbol table
If	if	ifsym	--
>=	< <= > >= = <>	relopsym	GE

Transition Diagrams

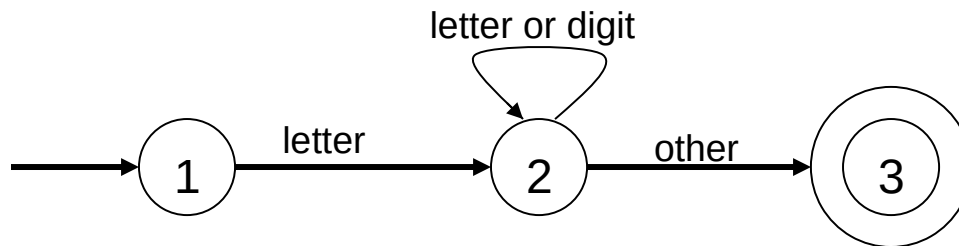
Transition diagrams or transition graphs are used to attempt to match a lexeme to a pattern.

Each Transition diagram has:

- States → represented by circles.
- Actions → represented by arrows between the states.
- Start state → represented by an arrowhead (beginning of a pattern)
- Final state → represented by two concentric circles (end of pattern).

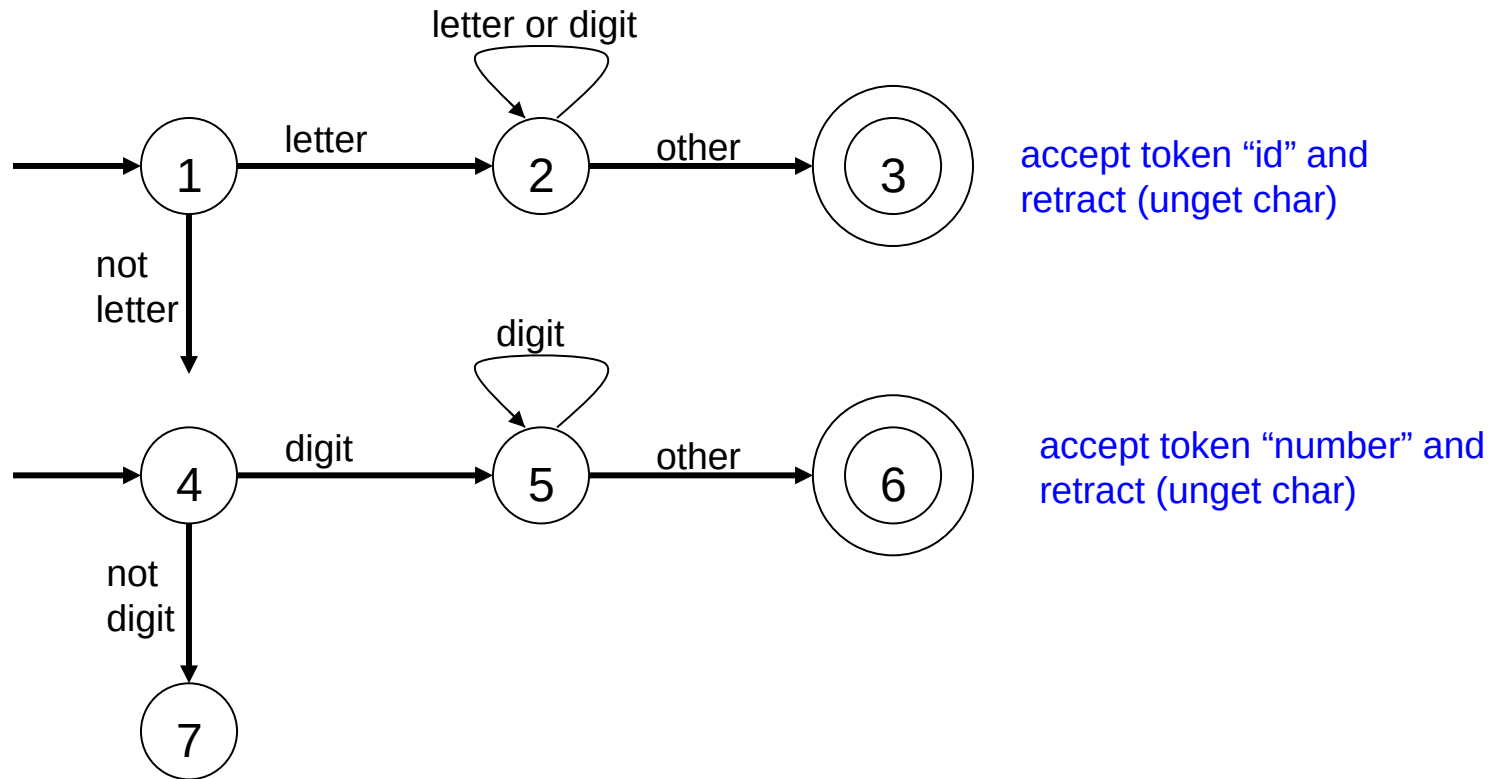
All transition diagrams are deterministic, which means that there is no need to choose between two different actions for a given input.

Example:



Transition Diagrams

The following state diagrams recognize identifiers and numbers (integers)



Transition Diagrams

This will be the translation of the transition diagrams to a programming language notation:

```
{state 1}      ch = getchar  
                If isletter (ch) then {
```

```
{state 2}      while isletter(ch) or isdigit(ch) do{  
                ch := getchar;  
                }
```

```
{state 3}      retract /* we have scanned  
                  /* one character too far  
                token := (id, index in name table)}  
                accept  
                return(token)  
                }  
                else {  
                Fail /* look for a different token  
                }
```

```
{state 4}      ch = getchar  
                if isdigit(ch) then {  
                value := convert (ch)
```

```
{state 5}      ch = getchar  
                while isdigit (ch) do{  
                value := 10 * value + conver (ch)  
                ch := getchar  
                }
```

```
{state 6}      retract  
                token := (int, value)  
                accept  
                return (token)  
                }
```

```
{state 7}      else{  
                Fail /* look for a different token  
                }
```

Transition Diagrams

Convert() turns a character representation of a digit into an integer in the range 0 -9.

Example:

Value := 10 * value + ch - '0';

or

Value := 10 * value + (ord(5) - ord(0))

↓ ↓

53 48 ← ASCII values for five and zero

```
ch = getchar
while isdigit (ch) do
    value := 10 * value + conver (ch)
    ch := getchar
endwhile
```

ASCII Character Set

The ordinal number of a character *ch* is computed from its coordinates (X,Y) in the table as:

$$\text{ord}(ch) = 16 * X + Y$$

Example:

$$\text{ord}('5') = 16 * 4 + 1 = 53$$

Y

X

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEI	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
10(A)	LF	SUB	*	:	J	Z	j	z
11(B)	VT	ESC	+	;	K	[k	{
12(C)	FF	FS	,	<	L	\	l	
13(D)	CR	GS	-	=	M]	m	}
14(E)	SO	RS	.	>	N	^	n	~
15(F)	SI	US	/	?	O	_	o	DEL

Transition Diagrams

“Transitions diagrams” are an implementation of a formal model called **Finite Automata (FA)** or **Finite State Machine (FSM)**.

Any language that can be denoted by a regular expression can be recognized by a Finite State Machine (FSM)

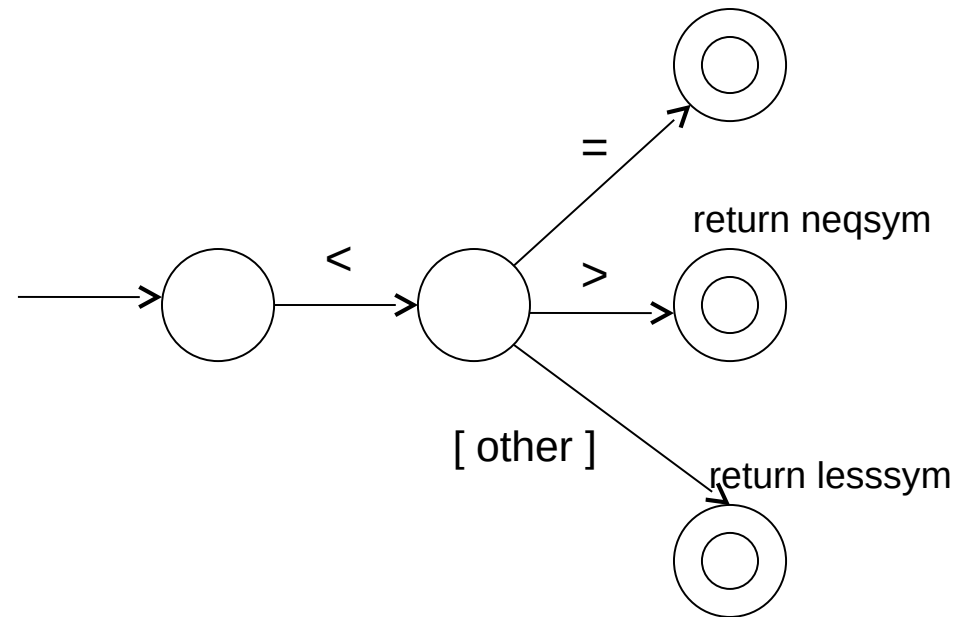
A transition diagram is equivalent to Deterministic Finite Automata (DFA)

Lexical Analyzer

Example of a lexical analyzer implementation:

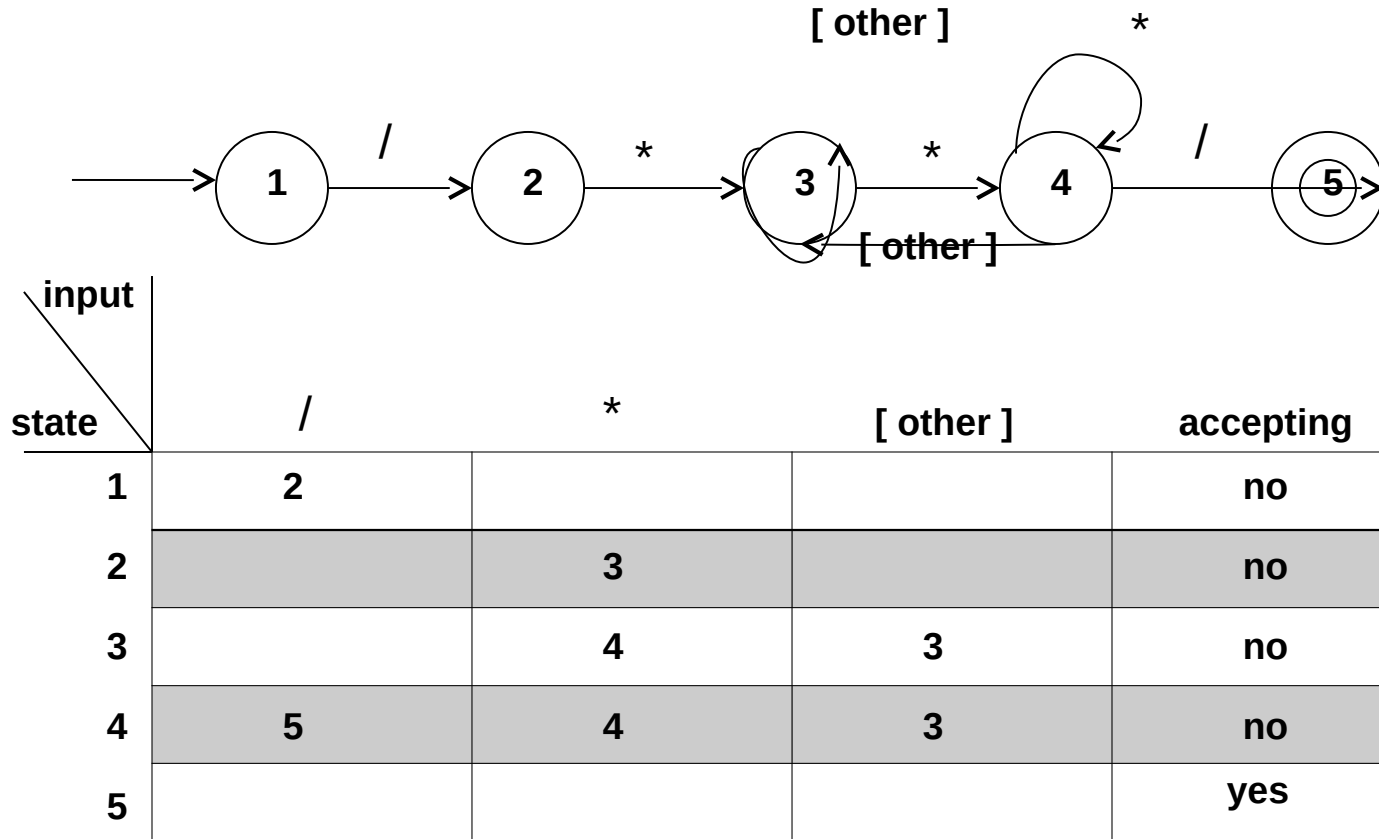
In this example we show you the algorithm to recognize the symbols "<", "<=", and "<>" equivalent to "!=":

```
ch := getch;  
If ch = '<' then  
  begin  
    ch := getch;  
    if ch = '=' then  
      begin  
        token := leqsym;  
        ch := getch  
      end  
    else  
      if ch = '>' then  
        begin  
          token := neqsym;  
          ch := getch  
        end  
      else token := lessym  
    end;  
end;
```



Lexical Analyzer

Transition diagram for C comments.



Transition table

Lexical Analyzer

THE END