Assignment No.5

AIM:

Implement the recognizer for given transition diagram.

THEORY:

Recognition of token

Consider the following grammar fragment:

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| Ɛ

expr → *term* **relop** *term*
| *term*

term → **num**
| **id**

where the terminals if, then, else, relop, id, and num generate sets of strings given by the following regular definitions:

if → if
then → then
else → else
relop→ < | <= | < > | = | > | >=
id → **letter** (**letter** | **digit**)*
num → **digit**+ (. **digit**+)? (**E** (+ | -)? **digit** +) ?

For this language fragment, the lexical analyzer will recognize the keywords if, then, else, as well as the lexeme denoted by relop, id, and mum. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines. Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition ws, below.

delim → blank | tab | newline

ws → delim+

If a match for ws is found, the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space and returns that to the parser. Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value, using the translation table given in Fig. The attribute-values for the relational operators are given by the symblic constants LT, LE, EQ, NE, GT, GE.

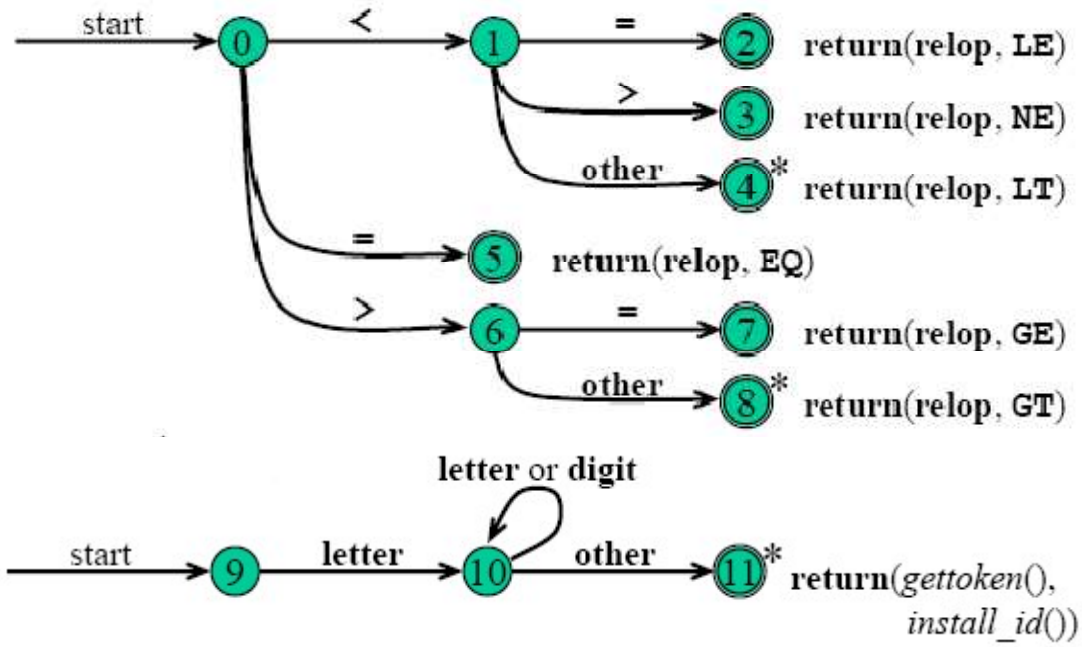
Regular expression	Token	Attribute-value
ws	-	-
if	if	-
then	then	-
else	else	-
num	num	Pointer to table entry
id	id	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

1. Transition diagram

Transition diagrams depict the actions that take place when a lexical analyzer is called by the parser to get the next token. Positions in a transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. Edges leaving state s have labels indicating the input characters that can next appear after the transition diagram has reached state S. The label other refers to any character that is not indicated by any of the other edges leaving s.

One state is labeled the start state; it is the initial state of the transition diagram where control resides when we begin to recognize a token. Certain states may have actions that are executed when the flow of control reaches that state. On entering a state we read the next input character. If there is an edge from the current state

whose label matches this input character, we then go to the state pointed to by the edge. Otherwise, we indicate failure.



2. Implementing the transition diagram

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the diagrams. We adopt a systematic approach that works for all transition diagrams and constructs programs whose size is proportional to the number of states and edges in the diagrams.

Each state gets a segment of code. If there are edges leaving a state, then its code reads a character and selects an edge to follow, if possible. A function nextchar() is used to read the next character from the input buffer, advance the forward pointer at each call, and return the character read. If there is an edge labelled by the character read, or labelled by a character class containing the character read, then control is transferred to the code for the state pointed to by that edge. If there is no such edge, and the current state is not one that indicates a token has been found, then a routine fail() is invoked to retract the forward pointer to the position of the beginning pointer and to initiate a search for a token specified by the next transition diagram. If there are no other transition diagrams to try, fail() calls an errorrecovery routine.

To return tokens we use a global variable lexical_value, which is assigned the pointers returned by functions install_id() and install_num() when an identifier or number, respectively, is found. The token class is returned by the main procedure of the lexical analyzer, called nexttoken().

```
int fail()
{ forward = token_beginning;
  swith (start) {
  case  0: start =  9; break;
  case  9: start = 12; break;
  case 12: start = 20; break;
  case 20: start = 25; break;
  case 25: recover(); break;
  default: /* error */
  }
  return start;
}
```

```
token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
       if (c==blank || c==tab || c==newline) {
         state = 0;
         lexeme_beginning++;
       }
       else if (c=='<') state = 1;
       else if (c=='=') state = 5;
       else if (c=='>') state = 6;
       else state = fail();
       break;
     case 1:
       ...
     case 9: c = nextchar();
       if (isletter(c)) state = 10;
       else state = fail();
       break;
     case 10: c = nextchar();
       if (isletter(c)) state = 10;
       else if (isdigit(c)) state = 10;
       else state = 11;
       break;
     ...
```

PROGRAM:

INPUT AND OUTPUT:

Enter input string :< =

return (RELOP,LE)

Do you wish to continue?(y/n):y

Enter input string: +

Invalid Input

Do you wish to continue?(y/n):y

Enter input string: abc

return(abc,1)

Do you wish to continue?(y/n):y

Enter input string:123

return(123,0)

Do you wish to continue?(y/n):n

CONCLUSION:

Thus the recognizer for given transition diagram is implemented.

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)